FOUR Js
The Power of Simplicity

Four Js Genero Business Development Language User Guide

Contents

Genero Business Development Language User Guide..... 15

What's new in Genero Business Development Language, v 3.00..... 18

General..... 22

Documentation conventions..... 22
- Syntaxes..... 22
- Warnings..... 22
- Code examples..... 23
- Enhancement references..... 23

General terms used in this documentation..... 23
Introduction to Genero BDL programming..... 24
- Overview of Genero BDL..... 25
- Genero BDL concepts..... 26

Installation..... 33

Documentation resources for upgrades..... 33
Software requirements..... 33
- Supported operating systems..... 33
- Database client software..... 33
- C compiler for C extensions..... 34
- Java™ runtime environment..... 34
- IPv6 support with Genero..... 35

Installing Genero BDL..... 35
Upgrading Genero BDL..... 35
Platform specific notes..... 36
- HP-UX configuration notes..... 36
- IBM® AIX® configuration notes..... 36
- Mac OS X configuration notes..... 37
- Microsoft™ Windows™ configuration notes..... 38
- Web Services platform notes..... 39

Upgrading..... 40

New features of Genero BDL..... 40
- What's new in Genero Business Development Language, v 3.00 (Maintenance Releases)... 41
- What's new in Genero Business Development Language, v 3.00..... 41
- What's new in Genero Business Development Language, v 2.51..... 45
- What's new in Genero Business Development Language, v 2.50..... 48
- What's new in Genero Business Development Language, v 2.41..... 51
- What's new in Genero Business Development Language, v 2.40..... 51
- What's new in Genero Business Development Language, v 2.32..... 55
- What's new in Genero Business Development Language, v 2.30..... 56
- What's new in Genero Business Development Language, v 2.21..... 58
- What's new in Genero Business Development Language, v 2.20..... 62
- What's new in Genero Business Development Language, v 2.11..... 65

What's new in Genero Business Development Language, v 2.10 ... 68
What's new in Genero Business Development Language, v 2.02 ... 71
What's new in Genero Business Development Language, v 2.01 ... 72
What's new in Genero Business Development Language, v 2.00 ... 72
What's new in Genero Business Development Language, v 1.33 ... 76
What's new in Genero Business Development Language, v 1.32 ... 76
What's new in Genero Business Development Language, v 1.31 ... 77
What's new in Genero Business Development Language, v 1.30 ... 78
What's new in Genero Business Development Language, v 1.20 ... 82
What's new in Genero Business Development Language, v 1.10 ... 84
Frequently asked questions ... 85
FAQ001: Why do I have a different display with Genero than with BDS V3? ... 85
FAQ002: Why does an empty window always appear? ... 86
FAQ003: Why do some COMMAND KEY buttons no longer appear? ... 86
FAQ004: Why aren't the elements of my forms aligned properly? ... 87
FAQ005: Why doesn't the ESC key validate my input? ... 88
FAQ006: Why doesn't the Ctrl-C key cancel my input? ... 88
FAQ007: Why do the gui.* FGLPROFILE entries have no effect? ... 89
FAQ008: Why do I get invalid characters in my form? ... 89
FAQ009: Why do large static arrays raise a stack overflow? ... 89
FAQ010: Why do I get error -6366 "Could not load database driver *drivername*"? ... 90
Upgrade Guides for Genero BDL ... 90
General upgrade guide ... 90
3.00 upgrade guide ... 91
2.51 upgrade guide ... 101
2.50 upgrade guide ... 106
2.40 upgrade guide ... 109
2.32 upgrade guide ... 112
2.30 upgrade guide ... 113
2.21 upgrade guide ... 116
2.20 upgrade guide ... 117
2.11 upgrade guide ... 122
2.10 upgrade guide ... 123
2.02 upgrade guide ... 123
2.01 upgrade guide ... 123
2.00 upgrade guide ... 123
1.33 upgrade guide ... 129
1.32 upgrade guide ... 129
1.31 upgrade guide ... 130
1.30 upgrade guide ... 130
Planned desupport ... 139
Migrating from IBM® Informix® 4gl to Genero BDL ... 139
Introduction to I4GL migration ... 140
Installation and setup topics ... 140
User interface topics ... 141
4GL programming topics ... 147
Migrating from Four Js BDS to Genero BDL ... 151
Installation and setup topics ... 152
User interface topics ... 153
4GL Programming topics ... 161

Configuration ... 164
The FGLPROFILE file ... 164
Understanding FGLPROFILE ... 164
FGLPROFILE entry syntax ... 165

List of FGLPROFILE entries..... 166
Environment variables..... 169
Setting environment variables on UNIX™..... 169
Setting environment variables on Windows™..... 170
Setting environment variables in FGLPROFILE (mobile)..... 170
Operating system environment variables..... 171
Database client environment variables..... 173
Genero environment variables..... 173
Configuring the front-end connection..... 186
Configuring the database server connections..... 186

Language basics..... 188
Syntax features..... 188
Lettercase insensitivity..... 188
Whitespace separators..... 189
Quotation marks..... 189
Escape symbol..... 189
Statement terminator..... 190
Comments..... 190
Identifiers..... 191
Preprocessor directives..... 191
Data types..... 191
BIGINT..... 192
BYTE..... 193
BOOLEAN..... 195
CHAR(size)..... 195
DATE..... 197
DATETIME qual1 TO qual2..... 198
DECIMAL(p,s)..... 200
FLOAT..... 202
INTEGER..... 202
INTERVAL qual1 TO qual2..... 203
MONEY(p,s)..... 204
SMALLFLOAT..... 205
SMALLINT..... 205
STRING..... 206
TINYINT..... 207
TEXT..... 208
VARCHAR(size)..... 210
Type conversions..... 211
When does type conversion occur?..... 211
Data type conversion reference..... 212
Handling type conversion errors..... 216
Formatting numeric values..... 217
Formatting DATE values..... 220
Formatting DATETIME values..... 221
Formatting INTERVAL values..... 223
Literals..... 225
Integer literals..... 225
Numeric literals..... 225
Text literals..... 226
Datetime literals..... 227
Interval literals..... 228
Expressions..... 229
Understanding expressions..... 229

Boolean expressions... 230
Integer expressions... 231
Numeric expressions... 231
String expressions... 232
Date expressions... 232
Datetime expressions... 233
Interval expressions... 233
Operators... 234
Order of precedence... 234
General warnings regarding expressions... 236
List of expression elements... 236
Flow control... 267
CALL... 268
RETURN... 270
CASE... 270
CONTINUE *block-name*... 272
EXIT *block-name*... 273
FOR... 274
GOTO... 275
IF... 276
LABEL... 276
SLEEP... 277
WHILE... 277
Functions... 278
Understanding functions... 278
FUNCTION blocks... 278
Using functions in programs... 279
Examples... 280
Variables... 281
Understanding variables... 281
DEFINE... 281
Declaration context... 282
Structured types... 283
Database column types... 283
User defined types... 284
Variable initialization values... 284
INITIALIZE... 285
LOCATE (for TEXT/BYTE)... 286
FREE (for TEXT/BYTE)... 287
LET... 288
VALIDATE... 288
THRU operator... 289
Examples... 290
Constants... 291
Understanding constants... 291
CONSTANT... 291
Examples... 293
Records... 294
Understanding records... 294
DEFINE ... RECORD... 294
Examples... 296
Arrays... 296
Understanding arrays... 297
DEFINE ... ARRAY... 297
Static arrays... 298
Dynamic arrays... 300

Array methods..... 302
Copying complete arrays.....302
Examples..... 302
Types..... 303
Understanding type definition.....303
TYPE.....303
Using types in programs..... 304
Examples..... 305

Advanced features.....307

Localization.....307
Application locale.....307
Localized strings.....327
Runtime stack.....336
Passing simple typed values as parameter..... 336
Passing a record as parameter.....337
Passing a static array as parameter..... 337
Passing a dynamic array as parameter..... 338
Passing objects as parameter.....338
Passing a TEXT/BYTE as parameter.....339
Returning simple typed values from functions..... 339
Returning dynamic arrays from functions.....339
Returning TEXT/BYTE values from functions..... 340
Implicit data type conversion on the stack.....340
Exceptions..... 340
Understanding exceptions..... 341
Exception classes.....341
Exception actions.....341
WHENEVER instruction.....342
TRY - CATCH block.....344
Tracing exceptions.....345
Default exception handling..... 346
Non-trappable errors.....346
Examples..... 346
OOP support.....349
Understanding classes and objects.....349
DEFINE ... *package.class*.....349
Distinguish class and object methods.....349
Working with objects.....350
What class packages exist?.....351
XML support..... 351
DOM and SAX standards.....351
DOM and SAX built-in classes.....351
Limitations of XML built-in classes.....352
Exception handling with XML classes.....352
Controlling the user interface with XML classes.....352
Globals.....353
Understanding global blocks..... 353
GLOBALS..... 353
Rules for globals usage.....353
Database schema in globals.....354
Content of a globals file.....354
Examples..... 354
Database schema.....355
Understanding database schemas.....356

SCHEMA....356
Structure of database schema files....357
Database schema extractor options....364
Programs....368
Structure of a program....368
Structure of a module....368
The MAIN block....370
Importing modules....371
Predefined constants....376
Configuration options....378
Program registers....387
Program execution....390
Executing programs....390
RUN....391
EXIT PROGRAM....394
BREAKPOINT....394
Front calls....395
Understanding front calls....395
ui.Interface.frontCall....395
User-defined front calls....397

SQL support....398
SQL programming....398
SQL basics....398
SQL security....410
SQL portability....412
SQL performance....452
Database connections....457
Understanding database connections....457
Opening a database connection....458
Database client environment....459
Connection parameters....461
Connection parameters in database specification....464
Direct database specification method....465
Indirect database specification method....466
IBM® Informix® emulation parameters in FGLPROFILE....466
Database vendor specific parameters in FGLPROFILE....469
Database user authentication....473
Unique session mode connection instructions....476
Multi-session mode connection instructions....477
Miscellaneous SQL statements....480
Database transactions....480
Understanding database transactions....481
BEGIN WORK....482
SAVEPOINT....483
COMMIT WORK....483
ROLLBACK WORK....484
RELEASE SAVEPOINT....484
SET ISOLATION....485
SET LOCK MODE....486
Static SQL statements....486
Understanding static SQL statements....487
Using program variables in static SQL....487
Table and column names in static SQL....488
SQL texts generated by the compiler....488

INSERT ... 489
UPDATE ... 490
DELETE ... 492
SELECT ... 493
SQL ... END SQL ... 495
CREATE SEQUENCE ... 496
ALTER SEQUENCE ... 496
DROP SEQUENCE ... 497
CREATE TABLE ... 497
ALTER TABLE ... 497
DROP TABLE ... 498
CREATE INDEX ... 498
ALTER INDEX ... 499
DROP INDEX ... 499
CREATE VIEW ... 499
DROP VIEW ... 499
CREATE SYNONYM ... 499
DROP SYNONYM ... 499
RENAME ... 500
Dynamic SQL management ... 500
Understanding dynamic SQL ... 500
PREPARE (SQL statement) ... 501
EXECUTE (SQL statement) ... 502
FREE (SQL statement) ... 503
EXECUTE IMMEDIATE ... 504
Result set processing ... 504
Understanding database result sets ... 504
DECLARE (result set cursor) ... 506
OPEN (result set cursor) ... 509
FETCH (result set cursor) ... 510
CLOSE (result set cursor) ... 511
FREE (result set cursor) ... 512
FOREACH (result set cursor) ... 512
Positioned updates/deletes ... 514
Understanding positioned update or delete ... 514
DECLARE (SELECT ... FOR UPDATE) ... 515
UPDATE ... WHERE CURRENT OF ... 516
DELETE ... WHERE CURRENT OF ... 517
Examples ... 517
SQL insert cursors ... 517
Understanding SQL insert cursors ... 518
DECLARE (insert cursor) ... 520
OPEN (insert cursor) ... 521
PUT (insert cursor) ... 521
FLUSH (insert cursor) ... 521
CLOSE (insert cursor) ... 522
FREE (insert cursor) ... 522
Examples ... 522
SQL load and unload ... 524
LOAD ... 524
UNLOAD ... 527
SQL adaptation guides ... 529
SQL guide for IBM® Informix® database servers 5.x, 7.x, 8.x, 9.x, 10.x, 11.x ... 529
SQL adaptation guide for IBM® DB2® UDB 10.x ... 540
SQL adaptation guide for IBM® Netezza® 6.x ... 572
SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 ... 592

SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x....625
SQL adaptation guide for Oracle Database 11, 12....643
SQL adaptation guide for PostgreSQL 9.x....683
SQL adaptation guide for SQLite 3.x....709
SQL adaptation guide for SAP Sybase ASE 16.x....723

User interface....747
User interface basics....747
The dynamic user interface....747
The abstract user interface tree....749
Genero user interface modes....752
Establish a GUI front-end connection....755
Special user interface features....759
Configuring a text terminal....762
Form definitions....769
Windows and forms....769
Using images....782
Accessibility guidelines....791
Message files....794
Action defaults files....796
Presentation styles....799
Form specification files....853
Form rendering....1002
Toolbars....1021
Topmenus....1027
Dialog instructions....1034
Static display (DISPLAY/ERROR/MESSAGE/CLEAR)....1034
Prompt for values (PROMPT)....1042
Ring menus (MENU)....1048
Record input (INPUT)....1060
Read-only record list (DISPLAY ARRAY)....1075
Editable record list (INPUT ARRAY)....1098
Query by example (CONSTRUCT)....1128
Multiple dialogs (DIALOG)....1144
Parallel dialogs (START DIALOG)....1199
User interface programming....1249
Dialog programming basics....1249
Input fields....1260
Dialog actions....1276
Table views....1345
Tree views....1384
Split views....1395
Drag & drop....1411
Web components....1416
Canvases....1448
Start menus....1454
Window containers (WCI)....1458

Reports....1460
Understanding reports....1460
XML output for reports....1461
Writing an XML report driver and routine....1461
Structure of XML report output....1462
Conditional statement output in XML reports....1462

The report driver....1464
START REPORT....1465
OUTPUT TO REPORT....1467
FINISH REPORT....1468
TERMINATE REPORT....1468
The report routine....1469
The report prototype....1471
DEFINE section in REPORT....1471
OUTPUT section in REPORT....1472
ORDER BY section in REPORT....1473
FORMAT section in REPORT....1474
Prohibited report routine statements....1480
Two-pass reports....1480
Report instructions....1480
EXIT REPORT....1480
PRINT....1481
PRINTX....1483
NEED....1484
PAUSE....1485
SKIP....1485
Report operators....1486
LINENO....1486
PAGENO....1486
SPACES....1487
WORDWRAP....1487
Report aggregate functions....1489
COUNT....1489
PERCENT....1490
SUM....1490
AVG....1490
MIN....1491
MAX....1491
Report engine configuration....1492

Programming tools....1493
Command line tools....1493
fglrun....1493
fglform....1495
fgl2p....1496
fglcomp....1497
fgllink....1499
fglmkmsg....1500
fglmkext....1500
fgldb....1501
fgldbsch....1502
fglmkstr....1503
fglwsdl....1503
fglpass....1506
fglWrt....1507
Compiling source files....1508
Compiling form files....1508
Compiling message files....1509
Compiling string files....1510
Compiling source code....1510
Importing modules....1511

Linking libraries....1511
Linking programs....1512
Using makefiles....1515
Module build information....1515
Source code edition....1516
Choosing the correct locale....1516
Avoid Tab characters in screen layouts....1517
Code completion and syntax highlighting with VIM....1517
Source documentation....1517
Understanding source code documentation....1518
Prerequisites for source documentation generation....1518
Documentation structure....1518
Adding comments to sources....1519
Run the documentation generator....1521
The preprocessor....1522
Understanding the preprocessor....1522
Compilers command line options....1522
File inclusion....1523
Simple macro definition....1525
Function macro definition....1527
Stringification operator....1529
Concatenation operator....1529
Predefined macros....1530
Undefining a macro....1530
Conditional compilation....1530
The debugger....1531
Understanding the debugger....1531
Prerequisites to run the debugger....1532
Starting fglrun in debug mode....1532
Attaching to a running program....1533
Debugging on a mobile device....1534
Stack frames in the debugger....1535
Setting a breakpoint programmatically....1536
Expressions in debugger commands....1536
Debugger commands....1537
The profiler....1556
Syntax of the program profiler....1556
Usage....1556
Example....1558
Optimization....1559
Runtime system basics....1560
Check runtime system memory leaks....1561
Optimize your programs....1561
Logging options....1563

Extending the language....1564
The Java™ interface....1564
Prerequisites and installation....1565
Getting started with the Java™ interface....1566
Advanced programming....1568
Examples....1592
C-Extensions....1597
Understanding C-Extensions....1597
Header files for ESQL/C typedefs....1598
Creating C-Extensions....1598

Creating Informix® ESQL/C Extensions....1599
The C interface file....1600
Linking programs using C-Extensions....1601
Loading C-Extensions at runtime....1601
Runtime stack functions....1602
C-Extension data types and structures....1606
Calling C functions from programs....1609
Calling program functions from C....1610
Sharing global variables....1611
Simple C-Extension example....1612
Implementing C-Extensions for GMI....1613
User-defined front calls....1615
Implement front call modules for GDC....1615
Implement front call modules for GMA....1620
Implement front call modules for GMI....1624
Implement front call modules for GWC - HTML5 theme....1631
Implement front call modules for GWC - JavaScript....1632
Web Components....1636

Library reference....1637
Built-in functions....1637
Built-in functions....1637
List of desupported built-in functions....1666
The key code table....1666
Utility functions....1667
Common dialog utility functions (IMPORT FGL fgldialog)....1668
Database utility functions (IMPORT FGL fgldbutl)....1672
Front-end dialog utility functions (IMPORT FGL fglwinexec)....1676
vCard utility functions (IMPORT FGL VCard)....1679
Built-in packages....1687
BDL data types package....1687
The base package....1703
The ui package....1755
The om package....1833
Built-in front calls....1881
Built-in front calls....1881
Extension packages....1947
The util package....1947
The os package....1990
The com package....2009
The xml package....2103
The security package....2278
File extensions....2296
Genero BDL errors....2297

Web services....2400
General....2400
Introduction to Web Services....2400
SOAP Web Services basics....2404
RESTful Web Services basics....2416
Getting started and examples....2416
Debugging....2416
Platform-specific notes....2416
Known issues....2419

Legal Notices.....2419
Concepts.....2419
High-level and low-level web services APIs.....2420
SOAP features.....2420
Stateful web services.....2422
Encryption, base64 and password agent with fglpass tool.....2429
HTTP compression.....2432
SOAP multipart style requests in GWS.....2434
Security.....2435
Encryption and authentication.....2435
Accessing secured services.....2438
HTTPS configuration.....2440
Certificates in practice.....2441
Examining certificates.....2443
Troubleshoot common issues.....2446
The Diffie-Hellman key agreement algorithm.....2447
SOAP Web Services.....2450
Writing a Web Services client application.....2450
Writing a Web Services server application.....2468
How To's.....2484
RESTful Web Services.....2505
Deploy a Web Service.....2506
Web services server program deployment.....2506
Configuring the apache web server for HTTPS.....2507
Reference.....2509
Web services configuration.....2509
Attributes to customize XML serialization.....2517
Error handling in GWS calls (STATUS).....2546
Interruption handling in GWS calls (INT_FLAG).....2546
Server API functions - version 1.3 only.....2546
Configuration API functions - version 1.3 only.....2552
Using fglwsdl to generate code from WSDL or XSD schemas.....2555

Mobile applications.....2557
Types of Genero Mobile apps.....2557
Language limitations.....2560
Environment variables on mobile.....2560
App localization.....2560
Apps user interface.....2561
Action rendering.....2561
Images and icons.....2562
Keyboard type.....2563
List views.....2563
Split views.....2564
Toolbars.....2565
Topmenus.....2565
Front call support.....2566
Color and theming.....2566
Database support on mobile devices.....2567
Using SQLite database in mobile apps.....2567
Accessing device functions.....2569
Web Services on mobile devices.....2569
Debugging a mobile app.....2570
Deploying mobile apps.....2572
Deploying mobile apps on Android™ devices.....2572

Deploying mobile apps on iOS devices..... 2584
Running mobile apps on an application server..... 2595
Push notifications..... 2599
Google Cloud Messaging (GCM)..... 2600
Apple Push Notification Service (APNs)..... 2605
Implementing a token maintainer..... 2611
Handling notifications in the mobile app..... 2617

Genero Business Development Language User Guide

Manual organization at a glance.

General on page 22	**Installation on page 33**	**Upgrading on page 40**	**Configuration on page 164**
• What's new in Genero Business Development Language, v 3.00 on page 18 • Documentation conventions on page 22 • General terms used in this documentation on page 23 • Introduction to Genero BDL programming on page 24	• Documentation resources for upgrades on page 33 • Software requirements on page 33 • Installing Genero BDL on page 35 • Upgrading Genero BDL on page 35 • Platform specific notes on page 36	• New features of Genero BDL on page 40 • Frequently asked questions on page 85 • Upgrade Guides for Genero BDL on page 90 • Planned desupport on page 139 • Migrating from IBM Informix 4gl to Genero BDL on page 139 • Migrating from Four Js BDS to Genero BDL on page 151	• The FGLPROFILE file on page 164 • Environment variables on page 169 • Configuring the front-end connection on page 186 • Configuring the database server connections on page 186
Language basics on page 188	**Advanced features on page 307**	**SQL support on page 398**	**SQL adaptation guides on page 529**
• Syntax features on page 188 • Data types on page 191 • Type conversions on page 211 • Literals on page 225 • Expressions on page 229 • Operators on page 234 • Flow control on page 267 • Functions on page 278 • Variables on page 281 • Constants on page 291 • Records on page 294 • Arrays on page 296	• Localization on page 307 • Runtime stack on page 336 • Exceptions on page 340 • OOP support on page 349 • XML support on page 351 • Globals on page 353 • Database schema on page 355 • Programs on page 368 • Program execution on page 390	• SQL programming on page 398 • Database connections on page 457 • Database transactions on page 480 • Static SQL statements on page 486 • Dynamic SQL management on page 500 • Result set processing on page 504 • Positioned updates/deletes on page 514 • SQL insert cursors on page 517 • SQL load and unload on page 524	• IBM Informix • IBM DB2 UDB • IBM Netezza • SQL Server • Oracle MySQL • Oracle database • PostgreSQL • SQLite • Sybase ASE

• Types on page 303			
User Interface: User interface basics on page 747	**User Interface: Form definitions on page 769**	**User Interface: Dialog instructions on page 1034**	**User Interface: User interface programming on page 1249**
• The dynamic user interface on page 747 • The abstract user interface tree on page 749 • Genero user interface modes on page 752 • Establish a GUI front-end connection on page 755 • Special user interface features on page 759 • Configuring a text terminal on page 762	• Windows and forms on page 769 • Using images on page 782 • Accessibility guidelines on page 791 • Message files on page 794 • Action defaults files on page 796 • Presentation styles on page 799 • Form specification files on page 853 • Form rendering on page 1002 • Toolbars on page 1021 • Topmenus on page 1027	• Static display (DISPLAY/ERROR/ MESSAGE/CLEAR) on page 1034 • Prompt for values (PROMPT) on page 1042 • Ring menus (MENU) on page 1048 • Record input (INPUT) on page 1060 • Read-only record list (DISPLAY ARRAY) on page 1075 • Editable record list (INPUT ARRAY) on page 1098 • Query by example (CONSTRUCT) on page 1128 • Multiple dialogs (DIALOG) on page 1144 • Parallel dialogs (START DIALOG) on page 1199	• Dialog programming basics on page 1249 • Input fields on page 1260 • Dialog actions on page 1276 • Table views on page 1345 • Tree views on page 1384 • Split views on page 1395 • Drag & drop on page 1411 • Front calls on page 395 • Web components on page 1416 • Canvases on page 1448 • Start menus on page 1454 • Window containers (WCI) on page 1458
Reports on page 1460	**Programming tools on page 1493**	**Extending the language on page 1564**	**Library reference on page 1637**
• Understanding reports on page 1460 • XML output for reports on page 1461 • The report driver on page 1464 • The report routine on page 1469 • Two-pass reports on page 1480 • Report instructions on page 1480 • Report operators on page 1486	• Command line tools on page 1493 • Compiling source files on page 1508 • Source code edition on page 1516 • Source documentation on page 1517 • The preprocessor on page 1522 • The debugger on page 1531 • The profiler on page 1556 • Optimization on page 1559	• The Java interface on page 1564 • C-Extensions on page 1597	• Built-in functions on page 1637 • Utility functions on page 1667 • Built-in packages on page 1687 • Extension packages on page 1947 • File extensions on page 2296 • Genero BDL errors on page 2297

• Report aggregate functions on page 1489 • Report engine configuration on page 1492	• Logging options on page 1563		
Web services on page 2400	**Mobile applications on page 2557**		
• General on page 2400 • Concepts on page 2419 • Security on page 2435 • Writing a Web Services client application on page 2450 • Writing a Web Services server application on page 2468 • How To's • Reference on page 2509	• Types of Genero Mobile apps on page 2557 • Language limitations on page 2560 • Environment variables on mobile on page 2560 • Database support on mobile devices on page 2567 • Web Services on mobile devices on page 2569 • Front call support on page 2566 • List views on page 2563 • Split views on page 2564 • Toolbars on page 2565 • Topmenus on page 2565 • Debugging a mobile app on page 2570 • Deploying mobile apps on page 2572 • Push notifications on page 2599		

What's new in Genero Business Development Language, v 3.00

This topic lists features added for the 3.00 GA release of the Genero Business Development Language.

Important: Please read also:

- What's new in Genero Business Development Language, v 3.00 (Maintenance Releases) on page 41, for a list of features that were introduced with the Genero BDL 3.00 Maintenance Releases.
- What's new in Genero Business Development Language, v 2.51 on page 45, for a list of features that were introduced with the Genero Mobile 1.0 release.

Table 1: Core language

Overview	Reference
The fglmkext command line tool can build your C Extension library.	See fglmkext on page 1500.
New fglcomp warning for invalid NULL usage in expressions like var==NULL.	See Compiler warning -6636.
C Extension runtime stack introspection (parameter type and actual string value size in bytes).	See Runtime stack functions on page 1602.
Temporary file name creation with `os.Path.makeTempName()`.	See os.Path.makeTempName on page 2002.
Attach the debugger to a running program with `fgldb -p` *`process-id`*.	See Attaching to a running program on page 1533.
Improved compilation time (`fglcomp` and `fglform`)	See Improved compilation time on page 98.
Datetime-related utility methods.	See util.Datetime.getCurrentAsUTC on page 1951, util.Datetime.format on page 1949, util.Datetime.parse on page 1951.
Date-related utility methods.	See util.Date methods on page 1947.
Interval-related utility methods.	See util.Interval methods on page 1955.

Table 2: User interface

Overview	Reference
Autocompletion in text edit fields with the COMPLETER attribute.	See Enabling autocompletion on page 1274.
Centralization of icon definitions with the FGLIMAGEPATH environment variable.	See Providing the image resource on page 784, FGLIMAGEPATH on page 182, Built-in front-end icons desupport on page 97.
Defining an action for `IMAGE` form items (clickable images).	See Defining action views in forms on page 1276, Defining actions on list columns with images on page

Overview	Reference
	1355, IMAGE item type on page 888.
Resizable `SCROLLGRID` containers (`WANTFIXEDPAGESIZE=NO`).	See WANTFIXEDPAGESIZE attribute on page 994.
Detect window resizing or device orientation change with the `windowresized` predefined action.	See Adapting to viewport changes on page 1003.
Dialog methods to convert the program array row index to the visual index, and the opposite.	See ui.Dialog.arrayToVisualIndex on page 1796, ui.Dialog.visualToArrayIndex on page 1815.
The `ON SORT` dialog control block can be used to execute code when the record list is re-ordered by the user.	See List ordering on page 1356, Populating a DISPLAY ARRAY on page 1372, ON SORT block on page 1091, ui.Dialog.getSortKey on page 1802, ui.Dialog.getSortReverse on page 1802.
`ON TIMER` trigger in dialogs, to execute a block of code at regular intervals.	See Get program control on a regular (timed) basis on page 1255.
Dynamic dialog creation.	See Implementing dynamic dialogs on page 1255.
Providing application image resources to Web Components with `ui.Interface.filenameToURI()`.	See Using image resources with the gICAPI web component on page 1430, ui.Interface.filenameToURI on page 1759.
Binding structured ARRAYs in DISPLAY ARRAY and INPUT ARRAY.	See Structured ARRAYs in list dialogs on page 100.

Table 3: SQL databases

Overview	Reference
Support for PostgreSQL 9.4.	See Database driver specification (driver) on page 462.
Support for Sybase ASE 16.x.	See Database driver specification (driver) on page 462.
Support for SQL Server 2008, 2012 and 2014 with FreeTDS driver (using FreeTDS 0.95)	See FreeTDS driver supports SQL Server 2008, 2012, 2014 on page 97.
SQL interruption is now supported with MySQL.	See SQL interruption on page 405.
MySQL VARCHAR(N) can be used when N is greater as 255.	See MySQL VARCHAR size limit on page 95.
MySQL DATETIME can store fractional seconds.	See MySQL DATETIME fractional seconds on page 96.

Overview	Reference
Maria DB support (V5.5 and V10): Use the dbmmys driver.	See MariaDB support on page 97.
Dynamic cursor built-in class `base.SqlHandle`.	See The SqlHandle class on page 1725.
Native Oracle NUMBER type (without precision/scale) can be extracted by fgldbsch.	See Oracle DB NUMBER type on page 95.
Serial emulation based on triggers and sequences with SQL Server 2012 and +.	See SERIAL data types on page 606.
PostgreSQL connection string option specification in the `source` parameter.	See Database source specification (source) on page 461, Prepare the runtime environment - connecting to the database on page 684.

Table 4: Web Services

Overview	Reference
Flushing immediately the response of a web service operation with `com.WebServicesEngine.flush`.	See com.WebServiceEngine.Flush on page 2026.
Base64 / Hexadecimal / Digest methods using a specific character set for string data.	See security.Base64.FromStringWithCharset on page 2282, security.Base64.ToStringWithCharset on page 2285, security.HexBinary.FromStringWithCharset on page 2288, security.HexBinary.ToStringWithCharset on page 2290, security.Digest.AddStringDataWithCharset on page 2293.
`com.WebServiceEngine` option `server_readwritetimeout` to define a server socket read/write timeout.	See Web Services changes on page 93, WebServiceEngine options on page 2032.
IPv6 support for Web Services clients.	See Configure a WS client to use IPv6 on page 2464.
Specific APIs for Apple Push Notification Service support.	See The APNS class on page 2095, com.TCPRequest.setKeepConnection on page 2091, com.TCPRequest.doDataRequest on page 2087, com.TCPResponse.getDataResponse on page 2093, Push notifications on page 2599.
Methods to perform RESTful requests using files on disk.	See com.HTTPServiceRequest.readFileRequest on page 2046, com.HTTPServiceRequest.sendFileResponse on page 2048,

Overview	Reference
	com.HTTPRequest.doFileRequest on page 2058, com.HTTPResponse.getFileResponse on page 2073, com.HTTPPart.getAttachment on page 2080, com.HTTPPart.CreateAttachment on page 2080.
FGLPROFILE entries to define XML Signature and XML Encrypted data prefix: `xml.signature.prefix` and `xml.encrypted.prefix`.	See XML configuration on page 2514.
SOAP fault handling works now when HTTP error 200 is returned by the server.	See SOAP fault handling in client stub on page 93.
Client stub multipart supports now optional parts.	See Optional multipart handling in client stub on page 94

Table 5: Mobile apps

Overview	Reference
Starting remote applications from a mobile device with the `runOnServer` front call.	See Running mobile apps on an application server on page 2595.
Extended feInfo front call options for mobile devices (deviceModel, deviceId, freeStorageSpace, iccid, imei, ppi, windowSize, and so on).	See feInfo on page 1893.
New `materialFABType` and `materialFABActionList` style attributes for Window class, to control the FAB button on devices following material design guidelines.	See Floating action button on Android devices on page 1286.
Front call to display a box controlling debug settings on GMA.	See showSettings (Android) on page 1941.
Push notification APIs for Google Cloud Messaging (GMA) and Apple Push Notification Service (GMI), with new predefined actions (`notificationpushed`).	See Push notifications on page 2599.
Command line tools to build mobile apps.	See Building Android apps with Genero on page 2574, Building iOS apps with Genero on page 2586.
Automatic FGLAPPDIR environment variable (defining the path to the *appdir*), and automatic FGLDIR environment variable, when executing on mobile devices.	See FGLAPPDIR on page 181, FGLDIR on page 181, Setting environment variables in FGLPROFILE (mobile) on page 170.
Front calls to take or choose videos on mobile devices.	See chooseVideo on page 1927, takeVideo on page 1939 front calls.

Table 6: Experimental features

Overview	Reference
Stacked form definition in .per files with the new STACK container, for mobile programming.	See Stack-based layout on page 1017, STACK container on page 914.

General

These topics provide an introduction to the Genero Business Development Language

- Documentation conventions on page 22
- General terms used in this documentation on page 23
- Introduction to Genero BDL programming on page 24
- Frequently asked questions on page 85

Documentation conventions

Learn about documentation conventions regarding syntax, warnings, code examples, and enhancement references.

- Syntaxes on page 22
- Warnings on page 22
- Code examples on page 23
- Enhancement references on page 23

Syntaxes

The term *syntax* is global and indicates the way to use a product function.

For example, it can be used to describe a language instruction or a command-line tool:

```
CALL function ( [ parameter [,...] ]  )
   [ RETURNING variable [,...] ]
```

Language keywords are written in uppercase.

Variable elements in a syntax definition are written in *italic*.

Wildcard characters in syntax definitions are marked with an underscore:

Table 7: Wildcard characters

Wildcards	Description
[*e*]	Square braces indicate an optional element in the syntax.
[*e1* | *e2* ...]	Square braces with pipe indicate an optional element to be selected from the list.
{ *e1* | *e2* ... }	Curly braces with pipe separator indicate a mandatory element the be selected from the list.
[...]	Indicates that the previous element can appear more than once.
[,...]	Previous element can appear more than once, and must be separated by a comma.

Warnings

Warnings are noticeable technical remarks, describing special behavior of the product function you must be aware of.

> **Important:** When a DATE, DATETIME or INTERVAL constant cannot be initialized correctly, it is set to NULL.

Some Genero features are not supported on all back-end or front-end platforms. The following note will warn you about the limitation:

Important: This feature is not supported on mobile platforms.

Important: This feature is only for mobile platforms.

Important: This feature is experimental, the syntax/name and semantics/behavior may change in a future version.

Code examples

Code examples contain code that can be copied as-is.

Code examples appear in the documentation as follows:

```
MAIN
  DEFINE a1 ARRAY[100] OF INTEGER,
         a2 ARRAY[10,20] OF RECORD
         id INTEGER,
...
```

Enhancement references

In some parts of the documentation you can find enhancement reference notes with a number identifying the request in our internal database.

Enhancement reference: BZ#1827

General terms used in this documentation

This documentation uses several terms that must be clarified for a good understanding.

Application	The *application* defines all software components that compose the information system managing a given domain. Usually, the domains covered by programs written in BDL are business oriented.
End user	The *end user* is the person that uses the application; that person works on hardware called the workstation.
Programs	The *programs* are the software components that are developed and distributed by the supplier of the application. Programs typically implement business logic. Programs are executed by the runtime system. Program components are typically p-code modules, forms and additional resource files.
Developer	The *developer* is the person in charge of the conception and implementation of the application components.
Application data	*Application data* defines the data manipulated by the application. It is typically managed by one or more database systems. The application data has a volatile state when loaded in the runtime system, and it has a static state when stored in the database system.

Database	The *database* is a logical entity regrouping the application data. It is managed by the database system.
Database system	The *database system* is the software that manages data storage and searching; it is usually installed on the database server machine and is supported by a tier software vendor.
Development database	The *development database* is the database used in the application development environment.
Production database	The *production database* is the database used on production sites.
Front end	The *front end* is the software that manages the display and input of the user interface on the workstation machine. This component is historically called "the client". It is the software handling the presentation. There are different sort of front-ends available, for desktop workstations (GDC), for web-browsers (GAS), and on mobile devices (GMA/GMI).
Runtime system	The *runtime system* is the software that manages the execution of the programs, where the business logic is processed. The runtime system is also known as the *Dynamic Virtual Machine* (DVM - fglrun).
User interface	The *user interface* defines the parts of the programs that interact with the end user, including interactive elements like windows, screens, input fields, buttons and menus. It is managed by the front-end.
Graphical user interface	The *graphical user interface* (GUI) mode identifies the user interface displayed on a remote machine via a front end. The GUI mode is active when the FGLGUI environment variable is set to 1 or when not set (GUI mode is the default).
Text user interface	The *text user interface* (TUI) mode identifies the user interface displayed on dumb terminals (TTY on UNIX™ or Console Window on MS Windows™). The TUI mode is active when the FGLGUI environment variable is set to 0.
Workstation	The *workstation* identifies the hardware used by the end user to interact with the front end. It can be an dumb terminal, a computer, or mobile device, as long as a front end is available on the hardware.

Introduction to Genero BDL programming

Understand the basics about programming, compiling and deploying an application.

- Overview of Genero BDL on page 25
- Genero BDL concepts on page 26

Overview of Genero BDL

Genero Business Development Language (BDL) is a program language designed to write an *interactive database application*.

A Genero BDL application is a set of programs that handle the interaction between a user and a database. Programs communicate with the database server with Structured Query Language (SQL), and execute interactive instruction controlling application forms, to manage user input.

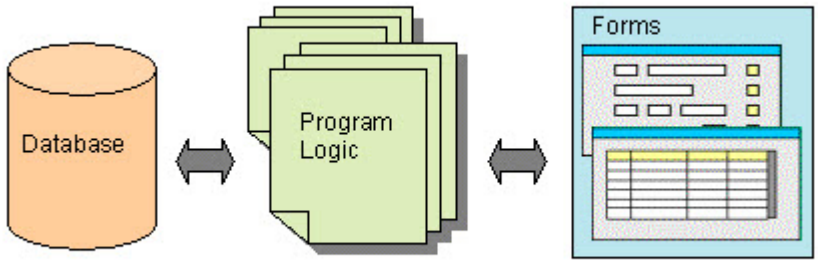


Figure 1: Interactive database applications with Genero

An important feature of the language is the ease with which you can design applications that allow the user to access and modify data in a database. The language syntax includes a set of SQL statements to manipulate the database, powerful interactive instructions that provide simple record input, read-only and read-write record list handling, as well as database query to search the database, by using forms supporting a large variety of graphical widgets.

The program sources are compiled to *p-code* modules, which can be interpreted on different platforms by the *Dynamic Virtual Machine* (the Runtime system).

- Separation of business logic and user interface on page 25
- Portability - write once, deploy anywhere on page 25

Separation of business logic and user interface

Genero BDL separates business logic and the user interface to provide maximum flexibility.

- Intensive use of *XML standards* ensures that user interface is well separated from the program logic.
- *Forms* define the user interface are designed in a simple-to-understand and simple-to-read syntax.
- The *business logic* is written in .4gl source code modules.
- High-level interactive instructions called *dialogs* let you write form controllers in a few lines of code.
- *Action views* (buttons, menu items, toolbar icons) in the form definition can trigger *actions* defined in the business logic.
- The user interface can be *manipulated at runtime*, for example to enable/disable fields and action views dynamically.

Portability - write once, deploy anywhere

Genero application can be deployed for different kinds of display devices, operating systems and database servers, by using the same source code.

Application forms can be displayed with a graphical front-end device based on native desktop frameworks, in web browsers, as well as on simple dumb terminals. Genero programs can be executed on major Operating Systems such as UNIX™, Linux™, Windows™ and Mac OS X® SQL can be performed by IBM® Informix®, or any other major database server such as Oracle DB, IBM® DB2®, Microsoft™ SQL Server, PostgreSQL, Oracle MySQL, Sybase ASE.

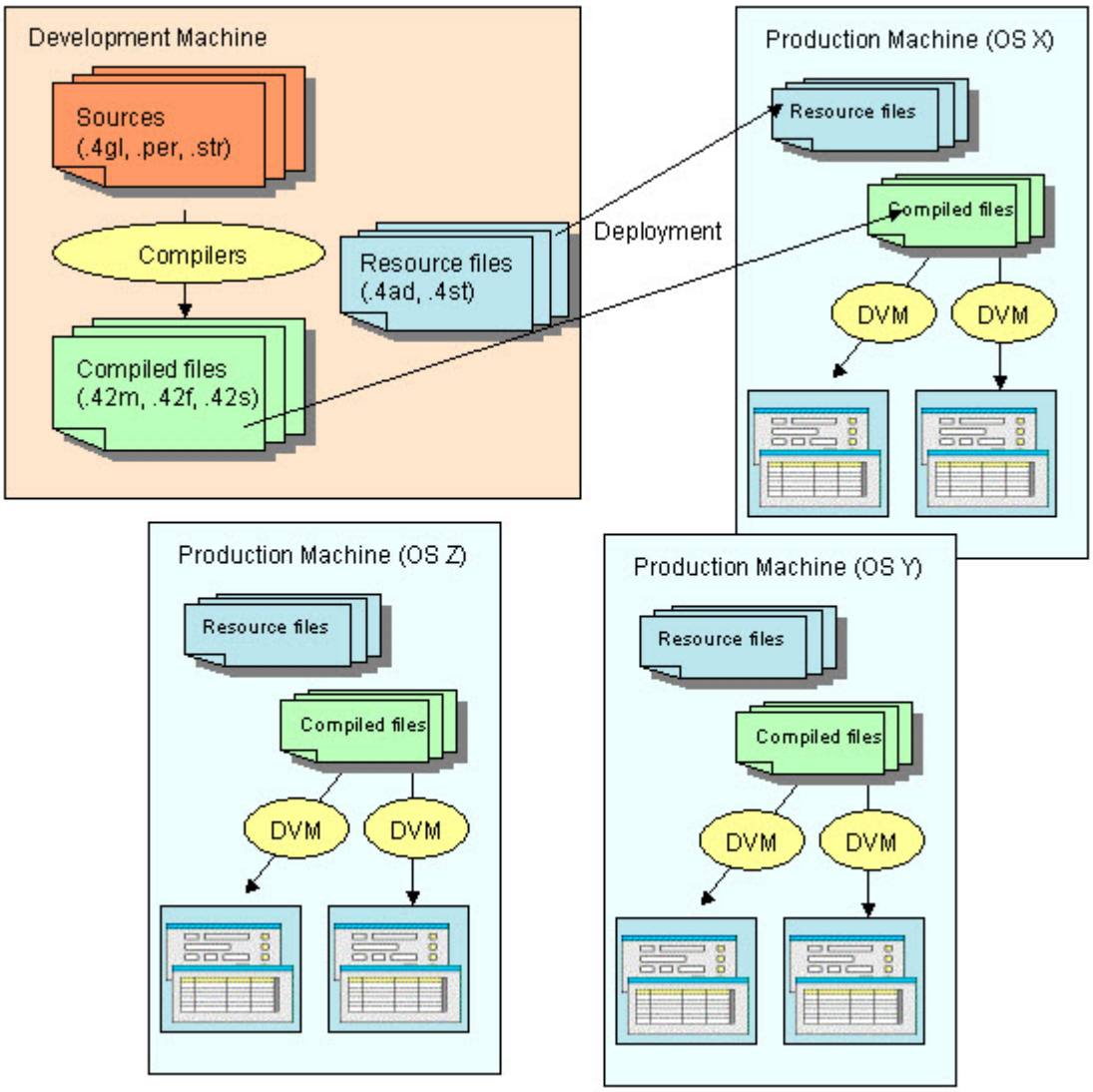


Figure 2: Genero portability

Genero BDL concepts

This section describes basic Genero language concepts.

- Genero programs on page 27
- Integrated SQL support on page 27
- The user interface on page 27
- Language library on page 28
- Windows and forms on page 28
- Interactive instructions on page 28
- Responding to user actions on page 29
- Producing reports on page 29
- Internationalization on page 29
- Web services support on page 29
- Extending the language on page 30
- Programming tools on page 30

Genero programs

Genero Business Development Language (BDL) is a programming language based on simple and readable syntax.

The program logic in written in text files with the `.4gl` file extension, called *program source modules*. Module sources are compiled (fglcomp) into p-code modules with the `.42m` file extension, that can be executed by the *runtime system* (fglrun). Application programs are build with a group of `.42m` modules.

Integrated SQL support

A set of SQL statements are part of the language syntax and can be used directly in the source code, as a normal procedural instruction.

The static SQL statements are parsed and validated at compile time. At runtime, these SQL statements are automatically prepared and executed. Program variables are detected by the compiler and handled as SQL parameters. Common SQL statements such as SELECT, INSERT, UPDATE or DELETE can be directly written in the source code, as part of the language syntax:

```
MAIN
  DEFINE n INTEGER, s CHAR(20)
  DATABASE stores
  LET s = "Sansino"
  SELECT COUNT(*) INTO n FROM customer WHERE custname = s
  DISPLAY "Rows found: " || n
END MAIN
```

Dynamic SQL management allows you to execute SQL statements that are constructed at runtime. The SQL statement can use SQL parameters:

```
MAIN
  DEFINE txt CHAR(20)
  DATABASE stores
  LET txt = "SET DATE_FORMAT = YMD"
  PREPARE sh FROM txt
  EXECUTE sh
END MAIN
```

Through the database drivers, the same program can open database connections to any of the supported databases.

XML support

The language provides XML support through different classes, according to the needs.

Genero XML support is provided in two forms:

- For basic XML tasks related to the user interface, use the built-in XML classes.
- For complex XML tasks, and Web Services functions, use the full-featured XML classes provided in the web services extension.

The user interface

The Genero user interface technology is based on the sharing of an abstract representation between the runtime system and the front-end.

When a program starts, the runtime system creates the *abstract user interface* (AUI) tree and passes this tree to the front-end. The front-end renders the abstract element as real graphical objects on the workstation.

When an interaction statement takes control of the application, the tree on the front-end is automatically synchronized with the runtime system tree. Runtime system and front-ends communicate with the *front end protocol*, through the computer network. The AUI tree and the protocol are using XML standards.

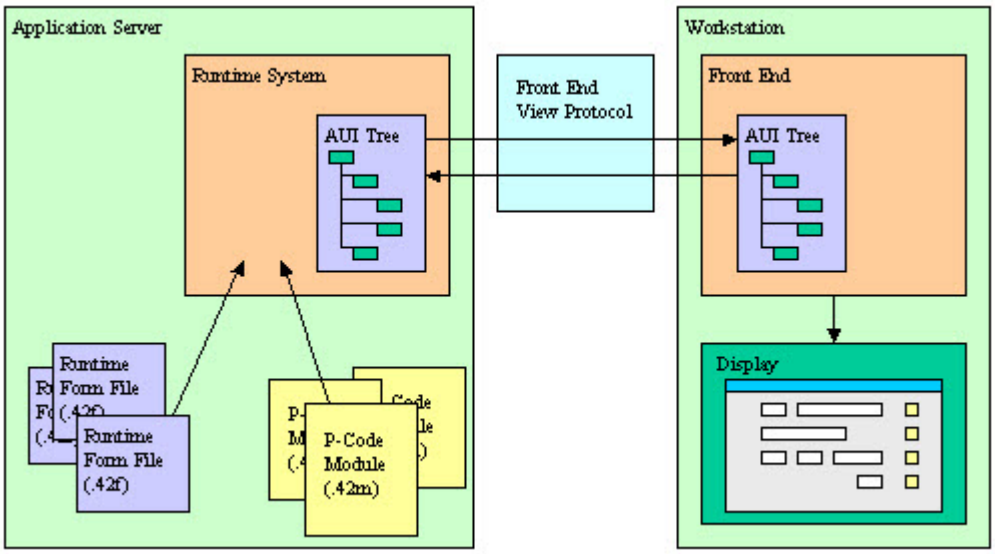


Figure 3: AUI tree synchronization

Resource files describe the appearance (decoration) of some of the graphic objects. Default resource files (`default.4ad`, `default.4st`) are provided and can be customized, or replaced with your own versions.

The elements of the AUI tree can be manipulated at runtime with built-in utilities.

Language library
Several utility packages are provided to ease programming in different domains.

Utility functions and classes are available in different forms, including built-in classes, built-in functions, and loadable modules.

Windows and forms
Programs manipulate window and form objects to define display areas controlled by interactive statements.

The forms are defined in text-based *form specification files* (.per). These form files are transformed by the fglform compiler to produce the *runtime form files* (.42f) that are deployed in production environments. The resulting (.42f) files are XML documents that describe the form elements, enabling portability across display devices. The XML file can also be written directly, or it can be generated or modified from your program at runtime with XML utilities.

Interactive instructions
Control application forms with interactive instructions that perform field input and action handling.

These *interactive instructions* allow the program to respond to user actions and data input. For example the `INPUT BY NAME` block controls a set of form fields where the user can enter data:

```
DEFINE cust_rec RECORD LIKE customer.*
INPUT BY NAME cust_rec.*
    ...
    BEFORE FIELD cust_name
        ...
    ON ACTION print
        ...
END INPUT
```

Interactive instructions can be implemented as modal or parallel dialogs. *Modal dialogs* control a given window, and that window closes when the dialog is accepted or canceled. The window displays on the top of any existing windows which are not accessible while the modal dialog executes. *Parallel dialogs* allow access to several windows simultaneously; the user can switch from on window to the other.

Responding to user actions
Clicking a form button or pressing a key triggers *actions* that can invoke the execution of program of code called *action handlers*. Form elements that can trigger actions are called *action views*.

Action handlers are defined in interactive statements with the `ON ACTION` clause. The code defined in action handler blocks is executed when an action is fired. Action objects are created and linked to action views when such `ON ACTION` handlers are seen by the runtime system. Common action handlers, such as `accept` (dialog validation) and `cancel` (dialog cancellation), are created automatically in accordance with the interactive instruction.

By configuring *action defaults*, you define the default decoration attributes (text, image) and functional attributes (accelerator keys, context menu display) for the action views associated with actions.

Producing reports
The language allows you to implement reports easily, producing different sort of output formats.

Page headers and footers, with page numbers, can be defined. Data can be grouped together, with group totals and subtotals shown. The output from a report can be sent to the screen, to a printer, to a file, or (through a pipe) to another program, and report output can even be redirected to an SAX filter in order to write XML data, that can be transformed into HTML, PDF or any other document format that can be generated from an XML source.

Internationalization
The language supports single-byte and multi-byte internationalization.

The language supports single-byte such as ISO-8859-1, as well as multibyte character sets such as BIG5 or UTF-8.

Length semantics to define variables and manipulate character string data can be based on byte or character units.

Labels and messages can be separated from programs and forms, to customize your application for specific subsets of for the user population, whether it is for a particular language or a particular business segment.

The source files (`4gl`, `per`, `4ad`, and so on) can be written in a specific encoding, however, we recommend you to keep sources in ASCII, and store locale-dependent strings in external strings files (`str`).

Web services support
The Genero Web service library allows to implement web service clients and servers.

Web services are a standard way of communicating between applications over an intranet or Internet. They define how to communicate between two entities:

- A server that exposes services
- A client that consumes services

The Genero Web Services Extension (GWS) is an extension to the Genero Business Development Language. It installs within the Genero Business Development Language directory. The fglgws package includes both Genero Business Development Language and Genero Web Services.

The Genero Application Server is required to manage your Web Services in a deployment environment. It is not required for Web services development, unless you are interested in testing deployment issues.

Extending the language

You can extend the language using C or Java™.

Using C

The language can be completed with *C extensions*. This allows you to implement specific function libraries in C, callable from the program modules. C extension libraries are typically used to interface with specific devices, such as barcode scanners or biometric identification devices.

Using Java™

You can instantiate Java™ objects from your programs by using the *Java™ interface*. This allows you to take benefit of the huge class library of Java™.

Programming tools

Genero BDL includes several programming tools in addition to compilers.

A set of useful programming tools is provided, to help you in the application development process, for debugging, optimization and source documentation production.

Compiling a program

You need to compile the source files in order to run the application.

A program can consist of a single source code module, but generally it will be organized in multiple modules, will involve form specification files and perhaps localized string files.

Database schema files are required when you define program data types and variables in the terms of an existing database column or table, by using the `DEFINE ... LIKE` statement.

Before running your application with the runtime system, you need to use compilation tools in order to build the various runtime files.

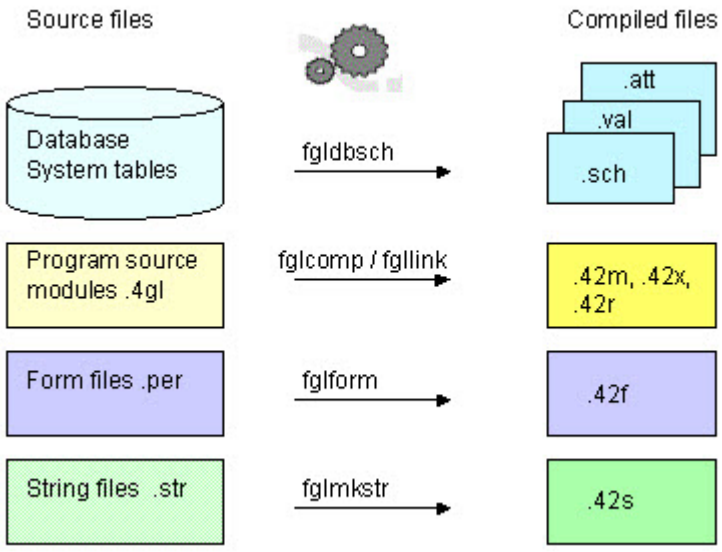


Figure 4: Genero compilation tools

The compiled source code modules can be linked into a .42r program that can be executed by the Runtime System. Compiled modules can also be grouped together into a.42x library that can then be used to build .42r programs.

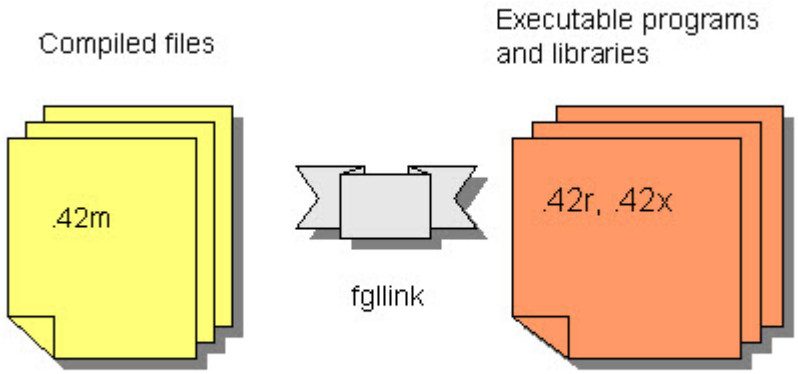


Figure 5: Linking of compiled modules

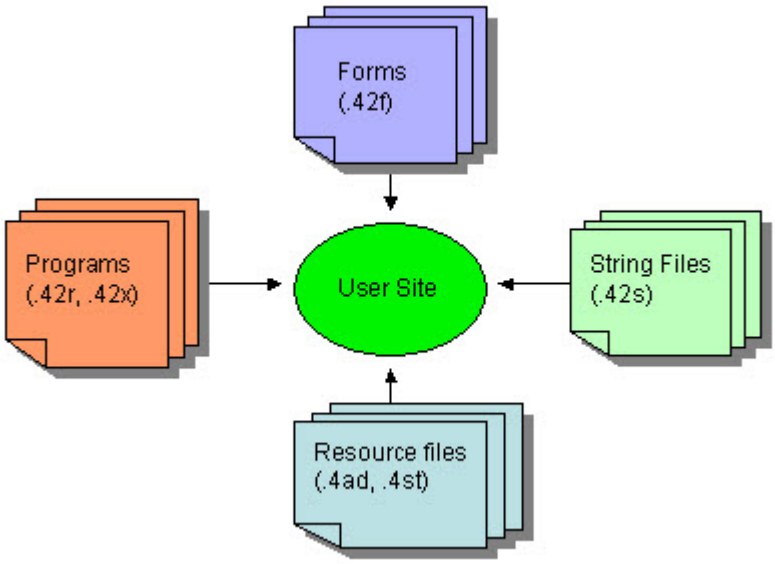
It is also possible to declare what modules are needed by the current module with the `IMPORT FGL` instruction, in order to define the dependency between .4gl modules. When using this language feature, it is no longer required to link modules together to build a program.

```
IMPORT FGL cust_module
MAIN
    DATABASE stores
    CALL cust_module.input_customer()
END MAIN
```

Importing modules is the preferred solution.

Deploying an application
To deploy an application, you must deploy all of the required runtime and resource files. Many (but not all) of these files are compiled from the source files.

Figure 6: Deployment files



These program files must be deployed at the user site:

- `.42r`, `.42x`, `.42m` - Executable programs and libraries, compiled modules
- `.42f` - Runtime form files
- `.42s` - compiled localized string files, if used in your applications
- `.4sm` - your custom Start Menu XML file, if created
- `.4ad`, `.4st` - these default XML files, provided with Genero, must be distributed with the runtime system files; if you have customized these files, or created your own versions, your versions must be deployed instead.

Runtime environment settings

The fglprofile configuration file and environment variables can be used to change the behavior of programs.

Installation

This chapter contains installation and setup instructions.

- Documentation resources for upgrades on page 33
- Software requirements on page 33
  - Supported operating systems on page 33
  - Database client software on page 33
  - C compiler for C extensions on page 34
  - Java runtime environment on page 34
- Installing Genero BDL on page 35
- Upgrading Genero BDL on page 35
- Platform specific notes on page 36
  - HP-UX configuration notes on page 36
  - IBM AIX configuration notes on page 36
  - Mac OS X configuration notes on page 37
  - Microsoft Windows configuration notes on page 38

Documentation resources for upgrades

Version-specific upgrade guides describe potential compatibility issues with new product releases.

Product improvements can be found in the new features section of this documentation.

Contact you support channel to get the list of corrected defects in the new version.

Software requirements

Before installing, ensure that your system meets the minimum system requirements and additional software.

Supported operating systems

Details of the supported operating systems for the Genero Business Development Language.

Genero Business Development Language is supported on a large brand of operating systems, such as Linux™, IBM® AIX®, HP-UX, SUN Solaris, Mac OS X® and Microsoft™ Windows™.

You must install the software package corresponding to the operating system that you use. For the detailed list of supported operating systems, refer to the relevant installation guide or contact your support center.

Database client software

To connect to a database server, the database client software must be installed on the system where you run the Genero BDL programs.

The Genero runtime system uses database drivers to connect to database servers, as a database client program. Database vendor-specific client software needs to be installed on the system where you run the Genero programs.

Example of database client software:

- IBM® Informix® Client SDK (with ESQL/C)
- IBM® okDB2 Connect® (with CLI)

- Oracle® Client (with OCI)
- Oracle® MySQL client (libmysqlclient)
- Microsoft™ SQL Server Native Client(with ODBC driver)
- PostgreSQL client (libpq)
- FreeTDS ODBC client (libtdsodbc)
- Easysoft™ ODBC client for SQL Server (libessqlsrv)
- SQLite 3.x (libsqlite3)
- SAP Sybase ASE™ OCS client

Database drivers are shipped as shared libraries and require a database client software shared library. The database driver to be selected must correspond to the database client type and version.

C compiler for C extensions

Ensure you have a C compiler and linker to compile your C-Extensions.

Applications using C extensions, need a C compiler and linker to build the C extension library that will be loaded by the runtime system.

C compiler On UNIX™ platforms

On UNIX platforms, you need a cc compiler on the system where you create the C extension libraries. Some systems may not have a C compiler by default. Make sure you have a C compiler on the system.

C compiler On Microsoft™ Windows™ platforms

On Windows platforms, it is mandatory to install Microsoft Visual C++ 2010 (also known as Visual C++ 10.0) on the system where you create the C extension libraries.

C compiler On Mac OS X™ platforms

On Mac OS X platforms, it is mandatory to install XCode 6.1, on the system where you create the C extension libraries.

Java™ runtime environment

Software requirements when using the Java™ Interface

In order to use the Java™ Interface in your application programs, you need the Java software installed and properly configured.

- Install a Java™ Development Kit on development sites (if you need to compile your own Java classes)
- Install a Java™ Runtime Environment on production sites (on the server where your programs are running)

The Java™ classes defined by Genero (`com.fourjs.fgl.lang.*`) are compiled with `javac -source 1.5 -target 1.5`, to be Java™ **1.5+** compatible. Therefore the minimum theoretical Java™ version is **1.5**. However, according to the platform, the minimum required version is **Java™ 1.6** or **1.7**.

The version of the installed Java software can be shown with the command:

```
java --version
```

In order to execute Java byte code, the Genero runtime system uses the JNI interface. The JVM is loaded as a shared library and its binary format must match the binary format of the Genero runtime system. For example, a 64-bit Genero package requires a 64-bit JVM.

When implementing Java classes for Genero Mobile for Android (GMA), check the JDK version required by the Android™ SDK. For more information, see the Android Studio web site.

IPv6 support with Genero

Network interface configuration for IPv6 support

IPv6 basics

IPV6 is the successor for IPv4, to increase the possible number of nodes of a computer network.

IPv6 support for WS clients

A Web Services client program can by default access to a WS server using IPv6. For more details, see Configure a WS client to use IPv6 on page 2464.

> **Note:** Web Services server programs work only in IPv4 to communicate with the GAS (since there is no need for IPv6 on a localhost). It's up to the web server to support IPv6 for the internet access of WS clients.

Installing Genero BDL

Different forms of installation programs are provided, as individual package or bundeled with other Genero components. Refer to the appropriate installation guide for a detailed description of the installation procedure. Do not hesitate to contact your support if you need help.

After installing a package, you should:

1. Set the FGLDIR environment variable to the installation directory.
2. Set the PATH environment variable to (FGLDIR)/bin in order to run compilers and runtime system tools from the command line.
3. Set the database client software environment (INFORMIXDIR, ORACLE_HOME, DB2DIR, SYBASE, PGDIR, LD_LIBRARY_PATH, etc)
4. Set access path to database client software DLLs (PATH), or Unix shared libraries (LD_LIBRARY_PATH, SHLIB_PATH, LIBPATH)
5. According to the database server you want to connect to, set up the correct database driver in FGLPROFILE. The default database driver is Informix®.
6. Depending what rendering mode you want to use (text mode or graphical mode), you will have to set environment variables such as FGLGUI, FGLSERVER, TERM, INFORMIXTERM.
7. If your application uses C-Extensions, a C compiler is required and you must recompile your C-Extensions as shared libraries.

Upgrading Genero BDL

After upgrading to a newer version, follow these next steps:

1. If the new version is a major upgrade (for example, from 2.20 to 2.21), recompile the sources and form files. While recompilation is not needed when migrating to maintenance release versions (for example, from 2.21.01 to 2.21.02), it is recommended to benefit from potential p-code optimizations.
2. If required, you may need to recreate the C-Extension libraries. C extension libraries must be provided as dynamically loadable modules and thus should not required a rebuild. However, if the C-Extension API header files have changed, consider recompiling your C sources. Check *`FGLDIR`*`/include/f2c` for C Extension API header file changes.

Platform specific notes

HP-UX configuration notes

Thread Local Storage in shared libraries

On HP-UX, the shared library loader cannot load libraries using Thread Local Storage (TLS), like Oracle `libclntsh`. In order to use shared libraries with TLS, you must use the LD_PRELOAD_ONCE environment variable. For more details, search for "shl_load + Thread Local Storage" on the HP support site.

PostgreSQL on HP-UX LP64

On HP-UX LP64, the PostgreSQL database driver should be linked with the `libxnet.sl` library if you want to use networking. You can force the usage of libxnet by setting the LD_PRELOAD_ONCE environment variable to `/lib/pa20_64/libxnet.sl`.

Java™ Interface

When using the Java™ Interface with the HotSpot JVM on HP/UX:

If you get an error when fglcomp or fglrun try to load the libjvm library, use the LD_PRELOAD environment variable:

```
$ LD_PRELOAD=libjvm.sl
$ export LD_PRELOAD
```

Using LD_PREPLOAD can make other applications fail. LD_PRELOAD should only be set for the runtime system. If you need to run other applications in the same environment as your application programs, you can set the LD_PRELOAD_ONCE or JAVA_PRELOAD_ONCE variable in the shell scripts found in FGLDIR/bin.

IBM® AIX® configuration notes

LIBPATH environment variable

The LIBPATH environment variable defines the search path for shared libraries. Make sure LIBPATH contains all required library directories, including the system library path `/lib` and `/usr/lib`.

Shared libraries archives

On AIX®, shared libraries are usually provided in .a archives containing the shared object(s). For example, the DB2® client library `libdb2.a` contains both the 32-bit (`shr.o`) and the 64-bit (`shr_64.o`) versions of the shared library. Not all products follow this rule; for example Oracle 9.2 provides `libclntsh.a` with `shr.o` on 64-bit platforms, and Informix® provides both `.a` archives with static objects and `.so` shared libraries as on other platforms.

The runtime system database drivers are created with the library archives or with the .so shared objects, according to the database type and version. No particular manipulation is needed to use any supported database client libraries on this platform.

The dump command

On IBM® AIX®, you can check the library dependencies with the `dump` command:

```
$ dump -Hv -X64 libstckp.so
```

Unloading shared libraries from memory

In production environments, AIX® loads shared libraries into the system shared library segment in order to improve program load time. Once a shared library is loaded, other programs using the same library are attached to that memory segment.

Once a shared library is loaded by the system, you cannot copy the executable file unless you unload the library from the system memory. This problem will occur when installing a new version of the software, even if it is installed in a different directory. Since shared libraries will have the same name, AIX® will not allow multiple versions of the same library to load. Therefore, before installing a new version, make sure all shared libraries are unloaded from memory.

The `genkld` command prints the list of shared libraries currently loaded into memory. The `slibclean` command unloads a shared library from the system shared library segment.

POSIX Threads and shared libraries

When using a thread-enabled shared library like Oracle's `libclntsh`, the program using the shared object must be linked with thread support, otherwise you can experience problems (like segmentation fault when the runner program ends). IBM® recommends using the xlc_r compiler to link a program with pthread support.

By default, the runtime system provided for AIX® platforms is linked with pthread support.

Java™ Interface

When using the Java™ Interface with the IBM® Java™ VM (J9VM) on AIX®, you may need to set the path to native shared libraries in the LIBPATH environment variable, if you get `java.lang.UnsatisfiedLinkError` exceptions:

```
$ LIBPATH=$JAVA_HOME/jre/bin:$JAVA_HOME/jre/bin/j9vm:$JAVA_HOME/jre/lib/
ppc64:$LIBPATH
$ export LIBPATH
```

This is required when using Java code that needs to access native code supplied as part of the JRE. For example, without setting LIBPATH to the appropriate path, the JVM cannot find the shared library `libnet.so`.

Using the `-Djava.library.path=`*`path-to-native-library`* java VM option does not seem to help.

Mac OS X configuration notes

DYLD_LIBRARY_PATH denied in OS X 10.11

Starting with Mac OS X 10.11 (El Capitan), if the System Integrity Protection (SIP) is enabled, the DYLD_LIBRARY_PATH environment variable is no longer exported in sub processes. This variable could be used to define the shared library search path for software components used by the Genero runtime system. This was required especially for database client libraries installed in directories other than `/usr/lib` and `/usr/local/lib` (the default location for shared libraries).

As DYLD_LIBRARY_PATH cannot be used, the proper workaround is to install all required shared libraries in `/usr/local/lib`. A good practice is to create the installation directory of the software component in `/usr/local/`*`product/version`*, and create symbolic links to the required shared libraries in `/usr/local/lib`.

> **Important:** You might need to install several shared libraries in `/usr/local/lib`: To make sure that all required libraries are available, check the dependencies with the `otool -L shared-library.dylib` command.

For example:

```
$ mkdir /usr/local/postgresql
$ mkdir /usr/local/postgresql/9.5.1

... install PostgreSQL in /usr/local/postgresql/9.5.1 ...

$ cd /usr/local/lib
$ ln -s /usr/local/postgresql/9.5.1/lib/libpq.5.dylib libpq.5.dylib
$ otool -L /usr/local/lib/libpq.5.dylib
...
```

Java™ Interface

When using the Java Interface, the runtime system is able to find automatically the `libjvm.dylib` according to the JAVA_HOME environment variable.

For more details, see Platform-specific notes for the JVM on page 1566

Microsoft™ Windows™ configuration notes

Microsoft™ Visual C++ version

When using C-Extensions, you need Microsoft™ Visual C++ compiler to compile and link your C sources. Make sure you have installed the software package corresponding to the MSVC version installed on your system. The MSVC version is identified in the software package name.

Checking binary dependencies

Microsoft™ Visual C++ provides the **dumpbin** utility to extract information from a binary file.

Use the **/dependents** option to check for DLL dependencies:

```
C:\ dumpbin /dependents mylib.dll
Microsoft (R) COFF/PE Dumper Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file mylib.dll

File Type: EXECUTABLE IMAGE

Image has the following dependencies:

isqlt09a.dll
MSVCR71.dll
KERNEL32.dll

Summary

1000 .data
1000 .rdata
1000 .text
```

Changing the stack size of fglrun

On Windows™ platforms, the fglrun.exe binary has a predefined C stack size. In some rare cases (for example, when programs do deep recursion), the stack size of fglrun.exe binary needs to be changed to avoid a stack overflow. The stack size of fglrun can be changed this permanently by patching the EXE

file with the Microsoft™ Visual C++ **editbin** utility. Check the stack size by running the **dumpbin** utility on fglrun.exe as follows:

```
C:\ dumpbin /headers %FGLDIR%\bin\fglrun.exe
```

Search for the line containing "stack reserve" words in the OPTIONAL HEADER VALUES section:

```
OPTIONAL HEADER VALUES
    ...
    100000 size of stack reserve
```

The stack size is displayed in hexadecimal value. So for example, a value 100,000 means1,048,567 bytes = 1MB.

In order to modify the stack size of fglrun.exe, run the **editbin** utility on fglrun.exe with the /stack option:

```
C:\ editbin /stack:1000000 %FGLDIR%\bin\fglrun.exe
```

See Microsoft™ Visual C++ documentation for more details.

Web Services platform notes

Genero Web Services reference documentation contains a list of platform specific notes to consider. For more details, see Web Services platform specific notes.

Upgrading

These topics talk about what steps you need to take to upgrade to the next release of Genero Business Development Language, and allows you to identify which features were added for a specific version.

- New features of Genero BDL on page 40
- Frequently asked questions on page 85
- Upgrade Guides for Genero BDL on page 90
- Planned desupport on page 139
- Migrating from IBM Informix 4gl to Genero BDL on page 139
- Migrating from Four Js BDS to Genero BDL on page 151

New features of Genero BDL

These topics provide an look back at the new features introduced with each release of the Genero Business Development Language.

- **Product line 3.0x**
  - What's new in Genero Business Development Language, v 3.00 (Maintenance Releases) on page 41
  - What's new in Genero Business Development Language, v 3.00 on page 18
- **Product line 2.5x**
  - What's new in Genero Business Development Language, v 2.51 on page 45
  - What's new in Genero Business Development Language, v 2.50 on page 48
- **Product line 2.4x**
  - What's new in Genero Business Development Language, v 2.41 on page 51
  - What's new in Genero Business Development Language, v 2.40 on page 51
- **Product line 2.3x**
  - What's new in Genero Business Development Language, v 2.32 on page 55
  - What's new in Genero Business Development Language, v 2.30 on page 56
- **Product line 2.2x**
  - What's new in Genero Business Development Language, v 2.21 on page 58
  - What's new in Genero Business Development Language, v 2.20 on page 62
- **Product line 2.1x:**
  - What's new in Genero Business Development Language, v 2.11 on page 65
  - What's new in Genero Business Development Language, v 2.10 on page 68
- **Product line 2.0x:**
  - What's new in Genero Business Development Language, v 2.02 on page 71
  - What's new in Genero Business Development Language, v 2.01 on page 72
  - What's new in Genero Business Development Language, v 2.00 on page 72
- **Product line 1.3x:**
  - What's new in Genero Business Development Language, v 1.33 on page 76
  - What's new in Genero Business Development Language, v 1.32 on page 76
  - What's new in Genero Business Development Language, v 1.31 on page 77
  - What's new in Genero Business Development Language, v 1.30 on page 78
- **Product line 1.2x:**

- What's new in Genero Business Development Language, v 1.20 on page 82

- **Product line 1.1x:**
  - What's new in Genero Business Development Language, v 1.10 on page 84

What's new in Genero Business Development Language, v 3.00 (Maintenance Releases)

This topic lists features added for 3.00 MRs of the Genero Business Development Language.

Important: Please read What's new in Genero Business Development Language, v 3.00 on page 18, for a list of features that were introduced with Genero 3.00 General Availability release.

Table 8: User interface

Overview	Reference
The `standard.openFile` frontcall is now supported with GWC-JS.	See standard frontcall support matrix.
The `dictionariesDirectory` parameter for the `standard.feInfo` frontcall can be used to get the directory where spell checker dictionary files can be uploaded.	See `standard.feInfo` frontcall.

Table 9: Mobile apps

Overview	Reference
Front call to ask user for Android permissions.	See askForPermission (Android) on page 1940 front call.
GMA buildtool clean option to cleanup the scaffold directory in case of interruption or failure in prior build.	See Building Android apps with Genero on page 2574.

Note: The new features listed in this topic are available in the lastest version of the related products. Contact your support channel for more details.

What's new in Genero Business Development Language, v 3.00

This topic lists features added for the 3.00 GA release of the Genero Business Development Language.

Important: Please read also:

- What's new in Genero Business Development Language, v 3.00 (Maintenance Releases) on page 41, for a list of features that were introduced with the Genero BDL 3.00 Maintenance Releases.
- What's new in Genero Business Development Language, v 2.51 on page 45, for a list of features that were introduced with the Genero Mobile 1.0 release.

Table 10: Core language

Overview	Reference
The fglmkext command line tool can build your C Extension library.	See fglmkext on page 1500.
New fglcomp warning for invalid NULL usage in expressions like var==NULL.	See Compiler warning -6636.
C Extension runtime stack introspection (parameter type and actual string value size in bytes).	See Runtime stack functions on page 1602.

Overview	Reference
Temporary file name creation with `os.Path.makeTempName()`.	See os.Path.makeTempName on page 2002.
Attach the debugger to a running program with `fgldb -p process-id`.	See Attaching to a running program on page 1533.
Improved compilation time (`fglcomp` and `fglform`)	See Improved compilation time on page 98.
Datetime-related utility methods.	See util.Datetime.getCurrentAsUTC on page 1951, util.Datetime.format on page 1949, util.Datetime.parse on page 1951.
Date-related utility methods.	See util.Date methods on page 1947.
Interval-related utility methods.	See util.Interval methods on page 1955.

Table 11: User interface

Overview	Reference
Autocompletion in text edit fields with the COMPLETER attribute.	See Enabling autocompletion on page 1274.
Centralization of icon definitions with the FGLIMAGEPATH environment variable.	See Providing the image resource on page 784, FGLIMAGEPATH on page 182, Built-in front-end icons desupport on page 97.
Defining an action for `IMAGE` form items (clickable images).	See Defining action views in forms on page 1276, Defining actions on list columns with images on page 1355, IMAGE item type on page 888.
Resizable `SCROLLGRID` containers (`WANTFIXEDPAGESIZE=NO`).	See WANTFIXEDPAGESIZE attribute on page 994.
Detect window resizing or device orientation change with the `windowresized` predefined action.	See Adapting to viewport changes on page 1003.
Dialog methods to convert the program array row index to the visual index, and the opposite.	See ui.Dialog.arrayToVisualIndex on page 1796, ui.Dialog.visualToArrayIndex on page 1815.
The `ON SORT` dialog control block can be used to execute code when the record list is re-ordered by the user.	See List ordering on page 1356, Populating a DISPLAY ARRAY on page 1372, ON SORT block on page 1091, ui.Dialog.getSortKey on page 1802, ui.Dialog.getSortReverse on page 1802.
`ON TIMER` trigger in dialogs, to execute a block of code at regular intervals.	See Get program control on a regular (timed) basis on page 1255.

Overview	Reference
Dynamic dialog creation.	See Implementing dynamic dialogs on page 1255.
Providing application image resources to Web Components with `ui.Interface.filenameToURI()`.	See Using image resources with the gICAPI web component on page 1430, ui.Interface.filenameToURI on page 1759.
Binding structured ARRAYs in DISPLAY ARRAY and INPUT ARRAY.	See Structured ARRAYs in list dialogs on page 100.

Table 12: SQL databases

Overview	Reference
Support for PostgreSQL 9.4.	See Database driver specification (driver) on page 462.
Support for Sybase ASE 16.x.	See Database driver specification (driver) on page 462.
Support for SQL Server 2008, 2012 and 2014 with FreeTDS driver (using FreeTDS 0.95)	See FreeTDS driver supports SQL Server 2008, 2012, 2014 on page 97.
SQL interruption is now supported with MySQL.	See SQL interruption on page 405.
MySQL VARCHAR(N) can be used when N is greater as 255.	See MySQL VARCHAR size limit on page 95.
MySQL DATETIME can store fractional seconds.	See MySQL DATETIME fractional seconds on page 96.
Maria DB support (V5.5 and V10): Use the dbmmys driver.	See MariaDB support on page 97.
Dynamic cursor built-in class `base.SqlHandle`.	See The SqlHandle class on page 1725.
Native Oracle NUMBER type (without precision/scale) can be extracted by fgldbsch.	See Oracle DB NUMBER type on page 95.
Serial emulation based on triggers and sequences with SQL Server 2012 and +.	See SERIAL data types on page 606.
PostgreSQL connection string option specification in the `source` parameter.	See Database source specification (source) on page 461, Prepare the runtime environment - connecting to the database on page 684.

Table 13: Web Services

Overview	Reference
Flushing immediately the response of a web service operation with `com.WebServicesEngine.flush`.	See com.WebServiceEngine.Flush on page 2026.

Overview	Reference
Base64 / Hexadecimal / Digest methods using a specific character set for string data.	See security.Base64.FromStringWithCharset on page 2282, security.Base64.ToStringWithCharset on page 2285, security.HexBinary.FromStringWithCharset on page 2288, security.HexBinary.ToStringWithCharset on page 2290, security.Digest.AddStringDataWithCharset on page 2293.
`com.WebServiceEngine` option `server_readwritetimeout` to define a server socket read/write timeout.	See Web Services changes on page 93, WebServiceEngine options on page 2032.
IPv6 support for Web Services clients.	See Configure a WS client to use IPv6 on page 2464.
Specific APIs for Apple Push Notification Service support.	See The APNS class on page 2095, com.TCPRequest.setKeepConnection on page 2091, com.TCPRequest.doDataRequest on page 2087, com.TCPResponse.getDataResponse on page 2093, Push notifications on page 2599.
Methods to perform RESTful requests using files on disk.	See com.HTTPServiceRequest.readFileRequest on page 2046, com.HTTPServiceRequest.sendFileResponse on page 2048, com.HTTPRequest.doFileRequest on page 2058, com.HTTPResponse.getFileResponse on page 2073, com.HTTPPart.getAttachment on page 2080, com.HTTPPart.CreateAttachment on page 2080.
FGLPROFILE entries to define XML Signature and XML Encrypted data prefix: `xml.signature.prefix` and `xml.encrypted.prefix`.	See XML configuration on page 2514.
SOAP fault handling works now when HTTP error 200 is returned by the server.	See SOAP fault handling in client stub on page 93.
Client stub multipart supports now optional parts.	See Optional multipart handling in client stub on page 94

Table 14: Mobile apps

Overview	Reference
Starting remote applications from a mobile device with the `runOnServer` front call.	See Running mobile apps on an application server on page 2595.

Overview	Reference
Extended feInfo front call options for mobile devices (deviceModel, deviceId, freeStorageSpace, iccid, imei, ppi, windowSize, and so on).	See feInfo on page 1893.
New `materialFABType` and `materialFABActionList` style attributes for Window class, to control the FAB button on devices following material design guidelines.	See Floating action button on Android devices on page 1286.
Front call to display a box controlling debug settings on GMA.	See showSettings (Android) on page 1941.
Push notification APIs for Google Cloud Messaging (GMA) and Apple Push Notification Service (GMI), with new predefined actions (`notificationpushed`).	See Push notifications on page 2599.
Command line tools to build mobile apps.	See Building Android apps with Genero on page 2574, Building iOS apps with Genero on page 2586.
Automatic FGLAPPDIR environment variable (defining the path to the *appdir*), and automatic FGLDIR environment variable, when executing on mobile devices.	See FGLAPPDIR on page 181, FGLDIR on page 181, Setting environment variables in FGLPROFILE (mobile) on page 170.
Front calls to take or choose videos on mobile devices.	See chooseVideo on page 1927, takeVideo on page 1939 front calls.

Table 15: Experimental features

Overview	Reference
Stacked form definition in .per files with the new STACK container, for mobile programming.	See Stack-based layout on page 1017, STACK container on page 914.

What's new in Genero Business Development Language, v 2.51

This topic lists features added for the 2.51 release of the Genero Business Development Language.

> **Important:** Most of the new features of BDL 2.51 have been added for Genero Mobile. The features designed for Genero Mobile may not be supported by desktop and web-browser front ends in the coming releases.

Genero Mobile V 1.0 (FGL 2.51.06)

Table 16: Core language

Overview	Reference
The channel methods `openServerSocket()` and `readOctets()`	See base.Channel.openServerSocket on page 1713, base.Channel.readOctets on page 1715.
The `sort()` method of ARRAY variables.	See DYNAMIC ARRAY.sort on page 1700.

Overview	Reference
Remote debugging through network TCP socket	See Debugging on a mobile device on page 1534.
Datetime-related utility methods.	See util.Datetime methods on page 1949.
String-related utility methods.	See util.Strings methods on page 1956.
Write to stdout with `om.XmlWriter.createFileWriter(NULL)`.	See om.XmlWriter.createFileWriter on page 1878.

Table 17: Core language (mobile apps)

Overview	Reference
FGLPROFILE settings to define environment variables	See Setting environment variables in FGLPROFILE (mobile) on page 170.
The method `base.Application.isMobile()`	See base.Application.isMobile on page 1706.
FGL Java class to access Android™ JVM context	See Standard Java and Android library usage on page 1587.
VCard utility functions.	See vCard utility functions (IMPORT FGL VCard) on page 1679.

Table 18: User interface

Overview	Reference
Dialog-level action attribute definitions with `ON ACTION` *name* `ATTRIBUTES()`.	See Configuring actions on page 1318.
URL-based Web Components	See Using a URL-based web component on page 1419.
The `DATETIMEEDIT` form item type	See DATETIMEEDIT item type on page 884.
New `ON SELECTION CHANGE` control block.	See Multiple row selection on page 1381.

Table 19: User interface (mobile apps)

Overview	Reference
`START DIALOG` / `TERMINATE DIALOG` / `fgl_eventLoop()`	See Understanding parallel dialogs on page 1199.
Window `TYPE` attribute in `OPEN WINDOW` instruction.	See Window types on page 777.
`DISPLAY ARRAY` attributes for list views handling: `ACCESSORYTYPE`, `DETAILACTION`, `DOUBLECLICK`.	See Using tables on mobile devices on page 1362.
The `DISCLOSUREINDICATOR` action attribute.	See DISCLOSUREINDICATOR action attribute on page 1328

Overview	Reference
The `ROWBOUND` action attribute.	See ROWBOUND action attribute on page 1329
The `KEYBOARDHINT` form field attribute.	See KEYBOARDHINT attribute on page 973.
List filter with `DISPLAY ARRAY` dialog.	See Reduce filter on page 1358.
Method `ui.Interface.getFrontEndName()` can now return `GMI` or `GMA`	See ui.Interface.getFrontEndName on page 1761.
Front-end functions for Genero Mobile (GMA / GMI)	See Genero Mobile common front calls on page 1925, Genero Mobile Android front calls on page 1940, Genero Mobile iOS front calls on page 1945.
Toolbar style attribute `iosStretchSeparator`, to stretch item separators on iOS device toolbars.	See Toolbar style attributes on page 838.
Navigation bar button colors and background colors for iOS device (`iosTintColor` , `iosNavigationBarTintColor`, `iosToolBarTintColor`, `iosTabBarTintColor`) - provided as Window class style attributes.	See Window style attributes on page 839.

Table 20: SQL databases

Overview	Reference
Simplified database driver specification	See New database driver name specification on page 102.
Support for SQL Server 2014	See Database driver specification (driver) on page 462.
Support for Oracle Database 12c	See Database driver specification (driver) on page 462.
Support for PostgreSQL 9.3	See Database driver specification (driver) on page 462.
Better support for DATETIME types with SQLite	See DATETIME types with SQLite on page 103.
`STRING` typed variables can be used in SQL statements.	See STRING on page 206.

Genero Mobile V 1.1 (FGL 2.51.07)

Table 21: Core language (mobile apps)

Overview	Reference
Implementing C-Extensions on iOS / GMI.	See Implementing C-Extensions for GMI on page 1613.
Using Java interface for Android / GMA.	See Executing Java code with GMA on page 1587.
Implementing customer front calls for GMA.	See Implement front call modules for GMA on page 1620.

Overview	Reference
Presentation styles are now supported by mobile front-ends.	See Style attributes reference on page 818.
GMA bundles zxing for Android.	See scanBarCode on page 1937.

Table 22: Web Services (mobile apps)

Overview	Reference
Complete support of Web Services on mobile devices. (WS were partially supported in GM v1.0)	See Web services on page 2400.

Table 23: User interface (mobile apps)

Overview	Reference
Presentation styles are now supported by mobile front-ends.	See Style attributes reference on page 818.
GMA bundles zxing for Android.	See scanBarCode on page 1937.

What's new in Genero Business Development Language, v 2.50

This topic lists features added for the 2.50 release of the Genero Business Development Language.

Table 24: Core language

Overview	Reference
Support for character length semantics to simplify UTF-8 programming.	See Length semantics settings on page 314.
The UTF-8 character set can be used on Microsoft™ Windows™ platforms by setting the LANG environment variable to `.fglutf8`.	See Language and character set settings on page 313.
JSON (JavaScript™ Object Notation) utility classes.	See The util.JSON class on page 1966, The util.JSONObject class on page 1970, The util.JSONArray class on page 1979.
String to `DATETIME` conversion now accepts ISO 8601 format sub-set.	See Data type conversion reference on page 212.
The `base.Channel` method `dataAvailable()`, to check for channel readability.	See base.Channel.dataAvailable on page 1709.
With `IMPORT FGL`, fglcomp now automatically compiles imported modules when needed. To avoid implicit compilation, use the `--implicit=none` option of fglcomp.	See IMPORT FGL module on page 372.
The `--resolve-calls` or `-W implicit` fglcomp compiler options can be used to detected unresolved symbols.	See IMPORT FGL module on page 372.
The fglrun option `--print-imports` can be used to find modules dependencies and use `IMPORT FGL` instead of traditional linking.	See IMPORT FGL module on page 372.

Table 25: User interface

Overview	Reference
Dialog modularization. Declarative `DIALOG` blocks can be defined as module elements and reused with the `SUBDIALOG` keyword of procedural `DIALOG` blocks.	See Structure of a procedural DIALOG block on page 1153.
Form modularization. Use the new `FORM` layout keyword to include a sub-form in the current form specification file.	See Form file structure on page 901.
`CLEAR SCREEN ARRAY` instruction clears the values of all the rows of a form list (`TABLE`, `TREE`, `SCROLLGRID`).	See CLEAR SCREEN ARRAY.
`AUTONEXT` attribute is allowed in `DATEEDIT`, `SPINEDIT` and `TIMEEDIT` fields.	See DATEEDIT, SPINEDIT, TIMEEDIT.
`BUTTONEDIT` item type attribute `NOTEDITABLE`, to disable the field editor.	See NOTEDITABLE attribute on page 976.
`ON CHANGE` fired when selecting a date in `DATEEDIT` calendar or when changing the value of a `TIMEEDIT` widget.	See ON CHANGE block on page 1069.
Presentation style attributes `ringMenuButtonTextHidden` and `actionPanelButtonTextHidden` added to customize the default action view panels.	See actionPanelButtonTextHidden, ringMenuButtonTextHidden.
Presentation style attribute `thinScrollbarDisplayTime` to define the display time of the thin scrollbar when scrolling in fixed screen arrays.	See thinScrollbarDisplayTime
Presentation style attribute `customWidget` to define the widget to be used for a `TEXTEDIT` (and `CHECKBOX` - removed in V3.00).	See TEXTEDIT customWidget.
fglrun options `--start-guilog` and `--run-guilog`, to generate and replay a GUI protocol exchange.	See Front-end protocol logging on page 759.

Table 26: SQL databases

Overview	Reference
The SQLite driver `dbmsqt3xx` is now statically linked with the SQLite library, except on platforms where the SQLite library is usually present such as Linux™ and Mac OS-X™.	See the SQLite adaptation guide.
Database driver for PostgreSQL 9.2: `dbmpgs92x`. This driver is similar to the prior PGS 9.x drivers, it is supported for strict binary compatibility with the PostgreSQL 9.2 client library and is compiled with the 9.2 libpq headers.	See Database driver dbmpgs92x.
Database driver for IBM® DB2® UDB version 10: `dbmdb2Ax`. This driver is similar to the prior DB2® 9.x driver, it is supported for strict binary compatibility with the DB2® 10.x client library and is compiled with the 10.x CLI headers.	See Database driver dbmdb2Ax.
Support for the Oracle `RAW` data type, in order to use the `SYS_GUID()` values generator.	See The RAW data type on page 662.
`FGLPROFILE` entry for MySQL to specify the `my.cnf` client configuration file: `dbi.database.`*`dbname`*`.mys.config`.	See Oracle MySQL / MariaDB specific FGLPROFILE parameters on page 471.

Table 27: Web Services

Overview	Reference
New security library provides classes and methods to support basic cryptographic features. Although added for Genero Web Services, can be used for any Genero application.	See The security package on page 2278.
New signature methods in `xml.Signature` class: `signString()` and `verifyString()`.	See xml.Signature methods on page 2235.
New methods in `xml.CryptoKey`: `loadPublicFromString()` and `savePublicToString()`.	See xml.CryptoKey methods on page 2209.
Support of Diffie-Hellman key-agreement algorithm. It allows two peers to agree on the same symmetric key, the shared secret, without exchanging confidential data.	See The Diffie-Hellman key agreement algorithm on page 2447, Supported kind of keys on page 2221 and Computing the shared secret with Diffie-Hellman on page 2227
HTTP compression support has been added for Genero Web Services.	See HTTP compression on page 2432.
The `com.HTTPRequest.setAutoReply()` method now works for HTTP HEAD method as well as the GET method.	See com.HTTPRequest methods on page 2053.
DOM features: The Genero XML DOM library has been enhanced with new features that can be set with the setFeature() method or retrieved with the getFeature() method. • `load-save-base64-string` - loads and saves an XML document from/to a base64 string • `auto-id-attribute` - sets at document loading all unqualified attributes named ID, id, Id or iD of type ID • `auto-id-qualified-attribute` - sets at document loading all qualified attributes named ID, id, Id or iD of type ID • `enable-html-compliancy` - allows HTML document parsing and modification using the xml.DomDocument API.	See The DomDocument class on page 2104.
Binary support on HTTP layer: The Genero COM library has been enhanced to support transport of binary data via the Genero BYTE data type. On the client side, it is now possible to send and read binary data to/from a server with the following two methods: • `doRequest()` - sends binary data from a BYTE to a HTTP server • `getDataResponse()` - reads binary data from a HTTP server into a BYTE On the server side, it is possible to read and write binary data to a client with following two methods: • `readDataRequest()` - reads binary data from a HTTP client into a BYTE • `sendDataResponse()` - sends binary data from a BYTE to a HTTP client	See The com package on page 2009.

Overview	Reference
Access the HTTP headers request and response in high level web services.	See how to modify your server or use fglwsdl generated global end point at runtime.
The standard API is enhanced with few new methods and a new class called `HTTPPart` to handle the different part in a HTTP request or response at client and server side.	See The HTTPPart class on page 2077, The HTTPRequest class on page 2053, The HTTPResponse class on page 2070, and The HTTPServiceRequest class on page 2036.
The client side is able to generate stubs to support multiple part with Genero Web Services. Support for the server side is not yet provided. **Note:** Starting with version 2.50.25, when generating client stubs managing multipart, you will get extra input and/or output variables called `AnyInputParts` and `AnyOutputParts`, defined as a `DYNAMIC ARRAY` of `com.HTTPPart` objects. These arrays may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program, to handle those dynamic arrays in any functions calling such stubs. See Client stubs managing multipart changes on page 109.	See Multipart in the client stub on page 2460 and SOAP multipart style requests in GWS on page 2434.
FGLPROFILE HTTPS configuration details no longer needed to perform HTTPS communication. A default SSL configuration is now generated automatically.	See HTTPS configuration on page 2440.
Creating URL base that applies to multiple server applications by using a wildcard in the URL, allowing for a shared server configuration (such as authentication and HTTPS).	See Wildcards in the URL base on page 2517.

What's new in Genero Business Development Language, v 2.41

This topic lists features added for the 2.41 release of the Genero Business Development Language.

Table 28: User interface

Overview	Reference
The `datatypeHint` style attribute (for Edit item types) and `nativeLook` style attribute (for CheckBox item types) have been added for use by the GWC for HTML5 front end. **Important:** In 2.50, the `nativeLook` attribute is renamed `customWidget`.	See Edit style attributes on page 826 and CheckBox style attributes on page 822.

What's new in Genero Business Development Language, v 2.40

This topic lists features added for the 2.40 release of the Genero Business Development Language.

Table 29: Core language

Overview	Reference
The NVL() operator allows you to write the equivalent of an IF *expr* IS NOT NULL THEN RETURN *expr* ELSE RETURN *default* END IF statement in a single scalar expression.	See NVL() on page 242.

Overview	Reference
The IIF() allows you to write the equivalent of an IF bool-expr THEN RETURN *true-value* ELSE RETURN *false-value* END IF statement in a single scalar expression.	See IIF() on page 243.
A new global program option has been added, OPTIONS SHORT CIRCUIT, to instruct the runtime system to evaluate Boolean expressions by using the *short-circuit evaluation* (also called *minimal evaluation*) method.	See OPTIONS (Compilation) on page 378 and Controlling semantics of AND / OR operators on page 379.

Table 30: User interface

Overview	Reference
New ON INSERT, ON APPEND, ON UPDATE and ON DELETE interaction blocks are now allowed in DISPLAY ARRAY dialogs to implement list modification, as an alternative to the traditional INPUT ARRAY dialog. These new triggers simplify the programming of modifiable record lists.	See DISPLAY ARRAY modification triggers on page 1380.
The new find and findnext actions of DISPLAY ARRAY and INPUT ARRAY can be used by the user to search rows where a field value matches the value entered in the find dialog box.	See Find function on page 1357.
The DISPLAY ARRAY dialog now supports a built-in seek feature to quickly find rows where a field value starts with the character typed by the user.	See Keyboard seek on page 1357.
It is now possible to define a summary line for TABLEs by using AGGREGATE form fields. Values can be automatically computed or can be calculated and displayed by program	See AGGREGATE item definition on page 933.
You can now use the terminfo database for text terminal mode (FGLGUI=0) by setting INFORMIXTERM=terminfo.	See Configuring a text terminal on page 762.

Table 31: SQL databases

Overview	Reference
New database drivers are provided.	List of new database drivers: • **dbmntz6x** for IBM® Netezza® ODBC client • **dbmsncB0** for SQL Server 2012 Native client • **dbmesmB0** for SQL Server 2012, with Easysoft ODBC driver • **dbmpgs91x** for PostgreSQL 9.1.x client
The fglcomp compiler now supports SQL ... END SQL blocks for compliance with IBM® Informix® 4GL.	See SQL ... END SQL on page 495.
The Static SQL syntax has been extended to allow the FIRST, LIMIT, SKIP and MIDDLE SELECT projection clause options.	See Static SQL statements on page 486.
The CASE operator is now allowed in Static SQL statements.	See Static SQL statements on page 486.

Overview	Reference
The syntax of DDL (Data Definition Language) statements in Static SQL now allows the IF NOT EXISTS and IF EXISTS clauses.	See Static SQL statements on page 486.
The transaction instruction set has been completed with SAVEPOINT and ROLLBACK WORK TO SAVEPOINT.	See SAVEPOINT on page 483 and ROLLBACK WORK on page 484.
Control shadow column extraction with fgldbsch.	See fgldbsch on page 1502.
A new FGLPROFILE entry parameter has been added to control the ORACLE DATE fetch into CHAR/VARCHAR variables.	See DATE and DATETIME data types on page 657.
Support for the ROWVERSION data type of SQL Server (2008 and +) has been added.	See SQL Server ROWVERSION data type on page 611.

Table 32: Web Services

Overview	Reference
The Genero Web Service engine has been enhanced to support a part of SOAP 1.2 protocol, restricted to the SOAP POST feature only. It does not support the SOAP 1.2 encoding feature, as it is prohibited by the WS-I Basic Profile 2.0. To allow the SOAP 1.2 protocol in your Genero Web service application, call the `setFeature()` method of your web service to enable SOAP 1.2 support. The same Web service can provide both the SOAP 1.1 and SOAP 1.2 protocol. You can also specify the SOAP role of your Genero application if you pass the new `SoapModuleURI` option to the WebServiceEngine `setOption()` method in order to identify the headers the SOAP engine has to understand.	See com.WebService.setFeature on page 2017, com.WebServiceEngine.SetOption on page 2030, and WebServiceEngine options on page 2032.
The Genero Web Service engine has been enhanced to support the WS-Addressing 1.0 specification. To enable WS-Addressing 1.0 specification in your Genero Web service application, call the `setFeature()` method of your web service with "`TRUE`" or "`REQUIRED`" as a parameter.	See com.WebService.setFeature on page 2017.
The Genero Web Service engine has been enhanced to support stateful services. There are two kinds of stateful services: • Based on WS-Addressing: independent from the transport protocol used to to convey the state between the client and the server. • Based on HTTP cookies: depends on the transport protocol to convey the state between the client and the server. To create a stateful web service, call `com.WebService.createStatefulWebService()` with a simple BDL variable or a dedicated W3CEndpointReference record to handle the service state. You can also take a look at WS-Addressing and at the following links for additional information: JAX-WS, Oracle and Stateful based on cookies.	See com.WebService.CreateStatefulWebService on page 2012 and Stateful web services on page 2422.
The Genero Web Service engine has been enhanced to support SOAP faults in RPC and Document style services.	See The com package on page 2009 (com.WebService.createFault

Overview	Reference
On the server side, you can define BDL variables that will be thrown as SOAP faults to a web service client using the SOAP 1.1 or SOAP 1.2 protocol. The `fglwsdl` tool has also been enhanced to generate client and server stubs according to the SOAP fault described in the WSDL. • Method `createFault()` • Method `addFault()` • Method `SetFaultDetail()` • Tool `fglwsdl`	on page 2011, com.WebOperation.addFault on page 2020, com.WebServiceEngine.SetFaultDetail on page 2030) and fglwsdl on page 1503.
The Genero fglwsdl tool generates a new Endpoint record per service in the client stub to configure the client behavior at runtime without the need to modify the generated code. This feature requires regeneration of the client stub and modification of the server location assignment if used in your application (See migration note).	See Change WS client behavior at runtime on page 2453.
The Genero fglwsdl tool has been enhanced to support WS-Addressing 1.0, the SOAP 1.2 protocol and to handle operation faults in SOAP 1.1 and SOAP 1.2. The generated client and server stub will handle WS-Addressing 1.0, SOAP 1.2 protocol and manage soap faults as defined in the WSDL. The following options have been added: Options related to SOAP: • `-soap11` : Generate only client and server stubs supporting the SOAP 1.1 protocol. • `-soap12` : Generate only client and server stubs supporting the SOAP 1.2 protocol. • `-ignoreFaults` : Do not generate soap faults. Options related to WS-Addressing: • `-wsa <yes|no>` : Force support of WS-Addressing 1.0. if `yes`, disable support of WS-Addressing 1.0, if `no`, otherwise support WS-Addressing 1.0 according to the definition in the WSDL. Other options: • `-alias` : Generate FGLPROFILE Logical names in place of URLs for all client stubs. • `-extDir` : Add all schema files located in a directory and ending with .xsd as external schemas. • `-CA` : Validate HTTPS certificate against a certificate authority list.	See fglwsdl on page 1503.
The XML-Signature and XML Encryption API of the XML library have been enhanced with new built-in methods to ease compatibility with the WS-Security specification: • Method `getSignatureMethod()` • Method `getThumbprintSHA1()` • Method `getSHA1()`	See XML security classes on page 2208.

Overview	Reference
The XML library has been enhanced to support XML parsing from PIPE and saving to PIPE: • Method `loadFromPipe()` • Method `saveToPipe()` • Method `readFromPipe()` • Method `writeToPipe()`	See The xml package on page 2103 (xml.DomDocument.loadFromPipe on page 2124, xml.DomDocument.saveToPipe on page 2126, xml.StaxReader.readFromPipe on page 2198, xml.StaxWriter.writeToPipe on page 2182)
The Genero Web Services service library has been enhanced to support global SSL security configuration in `FGLPROFILE` for HTTPS communication. You can now define the SSL certificate and private key to be used for all secured connections with the following entries and still use a dedicated SSL configuration if needed for a particular server. • Entry `security.global.certificate` • Entry `security.global.privatekey` • Entry `security.global.keysubject` (Windows™ only) • Entry `security.global.protocol`	See Web services configuration on page 2509.
A universal unique identifier function, `CreateUUIDString()`, has been added to the COM library. This funciton generates a universal unique identifier in BDL.	This method is desupported since 3.00, use security.RandomGenerator.CreateUUIDString on page 2280 as replacement.
The Genero Web services library has been enhanced with two new serializers: • `xml.Serializer.DomToStax()` converts a Dom node to a Stax writer • `xml.Serializer.StaxToDom()` converts a Stax reader to a Dom node	See xml.Serializer.DomToStax on page 2204 and xml.Serializer.StaxToDom on page 2206.

What's new in Genero Business Development Language, v 2.32

This topic lists features added for the 2.32 release of the Genero Business Development Language.

Table 33: Web Services

Overview	Reference
The COM library enables to intercept high-level web services operation on server side. You can now define three BDL functions via methods of the web service class. They will be executed at different steps of a web service request processing in order to modify the SOAP request, response or the generated WSDL document before or after the SOAP engine has processed it. This helps handle **WS-*** specifications not supported in the web service API. • Method `registerWSDLHandler()` • Method `registerInputRequestHandler()` • Method `registerOutputRequestHandler()`	See The WebService class on page 2009.

Overview	Reference

All three kinds of BDL callback functions must conform to the following prototype:
`FUNCTION CallbackHandler( doc xml.DomDocument )`
`  RETURNING xml.DomDocument`	

What's new in Genero Business Development Language, v 2.30

This topic lists features added for the 2.30 release of the Genero Business Development Language.

Table 34: Core language

Overview	Reference
Genero is now available on Mac OS-X™. You need at least Mac OS X version 10.5. The Operating System code for Mac OS X 10.5 64-bit is m64x105.	See Supported operating systems on page 33.
Platform identifier is now displayed when using the -V option with command-line tools.	See fglrun on page 1493.
The FGLPROFILE environment variable now accepts multiple file specification with an operating-system-specific path separator.	See The FGLPROFILE file on page 164.
The LOAD, UNLOAD and base.Channel class support the "CSV" delimiter specification to read/write files in Comma Separated Value format.	See LOAD on page 524, UNLOAD on page 527 and The Channel class on page 1707
Version **2.30.04** supports now the `fglrun.arrayIgnoreRangeError` entry which can be set to true to force the runtime system to return the first element of an array when the array index is out of bounds.	See Arrays on page 296.
The version **2.30.04** introduces the new `fglrun.mapAnyErrorToError` FGLPROFILE entry. This configuration parameter can be set to true to map the default action of the WHENEVER ANY ERROR exceptions to the action defined for the WHENEVER ERROR exception type.	See Exceptions on page 340.

Table 35: User interface

Overview	Reference
Drag & Drop support in DISPLAY ARRAY for tables or tree views.	See The DragDrop class on page 1827.
A new form item type called WEBCOMPONENT is provided to integrate external Java-Script-based widgets in your forms.	See WEBCOMPONENT item type on page 900.
New ui.Form class method to make a specific form field visible, showing the parent containers automatically. This method can also be used to bring a given folder page to the front, even if the field is not active (i.e. not driven by a dialog).	See ui.Form.ensureFieldVisible on page 1777 and ui.Form.ensureElementVisible on page 1776.
The ERROR and MESSAGE instructions get an additional STYLE attribute, to reference a presentation style and define the rendering with font, color, and position.	See MESSAGE on page 1035.

Overview	Reference
New style for TOOLBAR and TOPMENU elements. See Front-End documentation for more details about possible decoration attributes.	See Toolbars on page 1021 and Topmenus on page 1027.
As with COMBOBOX, the items of a RADIOGROUP are now filled with the values of the INCLUDE attribute, if specified.	See RADIOGROUP item definition on page 943
Identify the last clicked CANVAS item with the drawGetClickedItemId() function of fgldraw.4gl.	See Step by step canvas example on page 1452
The FIELD_TOUCHED() operator and ui.Dialog.getFieldTouched() method accept now a simple star as parameter, in order to check all fields used by the dialog.	See FIELD_TOUCHED() on page 266 and ui.Dialog.getFieldTouched on page 1800.
The JUSTIFY attribute is now supported for all form item types, in order to let you specify both the data justification in the field/cell and the alignment of the table column header.	See JUSTIFY attribute on page 972.
The ui.Dialog.setFieldActive() method takes now a list of fields as parameter, with the "dot-asterisk" notation, like the setFieldTouched() method.	See ui.Dialog.setFieldActive on page 1812.
This new feature is part of the fix for bug **#18224**. When modifying a tree during the dialog execution (for example, when implementing dynamic trees with ON EXPAND / ON COLLAPSE triggers), if you use the ui.Dialog.insertRow(), ui.Dialog.deleteRow() or ui.Dialog.deleteAllRows() methods to modify the node list, the internal tree structure was corrupted. You could safely modify directly the program array with array methods, but multi-range selection flags and cell attributes are not synchronized when doing this. Starting with **2.30.02**, you can now use the ui.Dialog.insertNode(), ui.Dialog.appendNode() and ui.Dialog.deleteNode() methods to manipulate the node list and get additional data like row selection flags and cell attributes synchronized.	See The Dialog class on page 1784.

Table 36: SQL databases

Overview	Reference
New database drivers	List of new database drivers: • **dbmase0Fx** for Sybase ASE 15.x (2.30.01) • **dbmmys55x** for a Mysql 5.5.x client (2.30.01) • **dbmpgs90x** for a PostgreSQL 9.0.x client (2.30.02)
Informix® SMALLFLOAT and FLOAT can now be stored in Oracle native BINARY_FLOAT / BINARY_DOUBLE types.	See SQL adaptation guide for Oracle Database 11, 12 on page 643.
The LOAD, UNLOAD and base.Channel class support the "CSV" delimiter specification to read/write files in Comma Separated Value format.	See LOAD on page 524, UNLOAD on page 527 and The Channel class on page 1707
Use the fgl_db_driver_type() built-in function to identify the target database type.	See fgl_db_driver_type() on page 1646.

Overview	Reference
In order to identify the reason why a database driver cannot be loaded, when setting FGLSQLDEBUG you now get an additional debug message that contains the operating system error message (dlerror())	See .
The fgldbsch tool can now extract database schema from SQLite. However, pay attention to the data types used in SQLite (V 3.6): This database supports some standard type names in the SQL syntax but in reality the types used to store data are very limited. For example, a DATE will be stored as an integer or string (i.e. there is no native DATE type). See SQLite documentation for more details. The fgldbsch tool will extract the schema according to the original type names used to create the table.	See fgldbsch on page 1502.

What's new in Genero Business Development Language, v 2.21

This topic lists features added for the 2.21 release of the Genero Business Development Language.

Table 37: Core language

Overview	Reference
Program module dependency specification with IMPORT FGL instruction.	See The IMPORT FGL instruction

Support for C1 Ming Guo date format modifier: Enable the digit-based Ming Guo date format by adding the C1 modifier at the end of the value set for the DBDATE environment variable:
`$ DBDATE="Y3MD/C1"`
`$ export DBDATE`
Note:
• When using C1, the possible values for the Yn specifier are Y4, Y3, Y2.
• The MDY() function is sensitive to the C1 modifier usage in DBDATE.
• The USING operator supports the c1 modifier as well.
• The C2 modifier to use Era names is not supported.
• Unlike Informix® 4gl, when using negative years, the minus sign is placed over the left-most zero of the year.
• Front-ends may not support the Ming Guo calendar for widgets like DATEEDIT.
	See DBDATE on page 174

Table 38: User interface

Overview	Reference
VALUEMIN/VALUEMAX attributes for the SPINEDIT widget.	See SPINEDIT
New presentation styles attributes for `Window` nodes.	See actionPanelButtonTextAlign, ringMenuButtonTextAlign
New presentation styles attributes for `Image` nodes.	See alignment

Overview	Reference
Numeric keypad decimal separator: The decimal separator defined by DBMONEY or DBFORMAT will be used when pressing the dot key of the numeric keypad.	See DBMONEY and DBFORMAT.
Automatic display of BYTE images: Image data contained in a BYTE variable are now displayed automatically when using a simple DISPLAY BY NAME, DISPLAY TO or when the BYTE variable is used by a dialog instruction. The BYTE data must be located in a file (LOCATE IN FILE "path") or temp file (LOCATE IN FILE).	See IMAGE item definition on page 941.
Paged DISPLAY ARRAY supports undefined initial row count: With this feature, when using a Paged DISPLAY ARRAY, it was mandatory to provide the total number of rows in the result set, which required a `SELECT COUNT(*)` before executing the dialog instruction. The dialog now supports an undefined number of rows, with value -1 in the COUNT dialog attribute.	See Read-only record list (DISPLAY ARRAY) on page 1075.
New ui.Interface.setSize() method to to let you define the initial size of the WCI container window.	See The Interface class on page 1755.
New `formScroll` presentation style attribute for windows.	See Window style attributes on page 839.

Table 39: SQL databases

Overview	Reference
New database drivers	List of new database drivers: • **dbmesmA0** for an EasySoft 1.2.3 client • **dbmpgs84x** for a PostgreSQL 8.4.x client • **dbmoraB2x** for Oracle 11g release 2 (11.2)
New EasySoft driver to connect from UNIX™ to SQL Server. This driver is based on the EasySoft SQL Server ODBC client.	See SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 on page 592.
New PostgreSQL 8.4 driver with INTERVAL support: **dbmpgs84x**. This driver converts Informix-style INTERVALs to native PostgreSQL INTERVALs.	See SQL adaptation guide for PostgreSQL 9.x on page 683.

Static SQL column definition supports DEFAULT clause: The syntax of the `CREATE TABLE` and `ALTER TABLE` Static SQL statements allows the `DEFAULT` clause in column definitions.
`CREATE TABLE item (`
`num SERIAL,`
`name VARCHAR(50)`
`DEFAULT '<undefined>'`
`NOT NULL )`
	See Static SQL statements on page 486.

PostgreSQL database driver supports now TEXT/BYTE.	See Large OBject (LOB) types on page 696.

Overview	Reference

New Static SQL syntax for the INSERT statement, which removes the record member defined as SERIAL, SERIAL8 or BIGSERIAL in the schema file:
`SCHEMA mydb`
`...`
`DEFINE record RECORD LIKE table.*`
`...`
`INSERT INTO table VALUES record.*`
	See INSERT on page 489.

The LOAD can now raise error -846 when the input file has a corrupted line (missing or invalid field separator, invalid character set, UNIX/DOS line terminators). You can now easily find the invalid line by setting the FGLSQLDEBUG on page 185 environment variable. The runtime system will display such debug messages with the line number:
`| DBI: LOAD: Corrupted data file,`
`  check line #12345.`
	See LOAD on page 524.

ODBC Character type control with SNC driver is now possible by using simple char or wide-char character strings for ODBC, with the following FGLPROFILE entry:
`dbi.database.<dbname>.snc.widechar =`
`true/false`
	See SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 on page 592.

Table 40: Web Services

Overview	Reference
The `fglwsdl` tool supports HTTPS request to retrieve WSDL or XSD on the network. You must specify the X509 certificate and private key using these options: • `-cert` *`filename`* : The *filename* of the X509 PEM-encoded certificate. • `-key` *`filename`* : The *filename* of the X509 PEM-encoded private key associatedto the above certificate. • `-wCert` *`name`* : The *name* of the X509 certificate and its associated private key in the Windows™ key store (Windows™ Only)	See fglwsdl on page 1503.
The `fglwsdl` tool allows http authentication and proxy authentication when requesting a WSDL or an XSD on the network, and supports basic and digest authentication. Two options have been added for authentication. • `-pAuth` *`login password`* : The *login* and the *password* to be used for proxy authentication. • `-hAuth` *`login password`* : The *login* and the *password* to be used for http or https authentication.	See fglwsdl on page 1503.
The `fglwsdl` tool provides a new option that generates: • a client stub entirely based on the DOM API • calls to a request, response and fault callback function per service	See WS client stubs and handlers on page 2456.

Overview	Reference
This option is especially useful when you have to communicate with another web service that requires additional information on the XML request, or when it returns additional information that was not specified in the WSDL. For instance, this is the case if you have to communicate with web services using WS-Security. You can manipulate the XML document in the generated client stub using the XML-Signature or XML-Encryption API to perform the security part by hand before it is sent on the network. The following option has been added for that purpose: • `-domHandler` : Generate function calls to a request, response and fault callback handler, and force the use of DOM in the client stub.	
The COM library is enhanced by a new function called `HandleRequest` to allow low-level and high-level web services on the same server.	See com.WebServiceEngine.HandleRequest on page 2028
The COM library is enhanced to perform automatic reply on HTTP GET request when the server requires HTTP authentication, proxy authentication, or returns an HTTP redirect.	See com.HTTPRequest.setAutoReply on page 2064.
The XML library supports a new option, `xml_useutctime`, to serialize any BDL `DATE` and `DATETIME` using the UTC format requested in most WS-Security exchanges.	See Serialization option flags on page 2207.
The XML library has been enhanced with two APIs in the `CryptoKey` class. Due to security issues, the usage of a direct shared symmetric or HMAC key is not recommended; most secured operations should use a key derived from a common shared key instead. The XML library has been enhanced with two APIs in the CryptoKey class: • Constructor `CreateDerivedKey()` • Method `deriveKey()`	See Derived keys on page 2224.
The COM library has been enhanced with two helper APIs in a new `Util` class. In most Web Service security exchanges, the application must be able to compute digest passwords and use random binary data to detect reply attacks (for instance). The COM library has therefore been enhanced with two helper APIs in a new `Util` class: • Static method `CreateDigestString()` • Static method `CreateRandomString()`	These methods are desupported since 3.00, use security.Digest.CreateDigestString on page 2294 and security.RandomGenerator.CreateRandomString on page 2279
The StAX reader and writer classes have been enhanced with two new methods to set up the XML stream on a TEXT lob. It enables parsing of an XML document in StAX directly from a TEXT with the `readFromText()` method, and creating a new XML document saved directly as TEXT with the `writeToText()` method.	See The StaxWriter class on page 2170 and The StaxReader class on page 2183.
The Genero Web Services library has been enhanced to support XML wildcard attributes. Such wildcard attribute can be set in a XML schema or in a WSDL via the `anyAttribute` tag. It allows additional attributes belonging to other XML schemas in a main XML schema. The additional attributes are not necessarily known by the main schema. The `fglwsdl` tool has been enhanced to recognize the additional attribute and to generate a one-dimensional dynamic array with a	See Attributes to customize XML serialization on page 2517 and The Serializer class on page 2202.

Overview	Reference
new `XMLAnyAttribute` attribute, and the XML Serializer has been enhanced to handle the new `XMLAnyAttribute` during the serialization and deserialization process. A new option called `xs_processcontents` is supported by the XML Serializer to generate the XML schema of such wildcard attributes with a `processContents` tag that defines the way a validator will handle them.	
The package contains a new demo called SimplePKI that demonstrates the usage of XML-Encryption in Genero. It allows several clients to register to a centralized PKI (Public Key Infrastructure) service that generates a unique RSA key-pair per user. The private key is returned to the user during the registration or login, using a derived symmetric key based on the user's password to make it secure. Then any client is able to retrieve the public key of the registered users, and to encrypt XML data only readable by that user. **Note:** This demo could easily be adapted in a real-world application if (for instance) all key-pair are stored in a database for persistence. You can find the demo in the demo/WebServices/simplepki subdirectory or by running the demo application in your installation directory.	N/A.

What's new in Genero Business Development Language, v 2.20

This topic lists features added for the 2.20 release of the Genero Business Development Language.

Table 41: Core language

Overview	Reference
The Java™ Interface allows your programs to use the Java™ library.	See Java™ Interface.
New TINYINT, BIGINT and BOOLEAN data types.	See TINYINT on page 207, BIGINT on page 192, BOOLEAN on page 195.
Private functions: It is now possible to hide a function (or report) to the other modules with the new PRIVATE keyword.	See Understanding functions on page 278.
Automatic source documentation generator.	See Source documentation on page 1517.
The fglcomp compiler has been extended with a new option (`--timestamp`) to write the compilation timestamp to the generated 42m p-code module. If present, the timestamp will be printed when using `fglrun -b`. Use compilation timestamps only if really needed; every new compiled .42m module will be different, even if the source code has not changed.	See fglcomp on page 1497.
The FGLRESOURCEPATH environment variable to define search paths for program resource files like forms.	See FGLRESOURCEPATH on page 184.
New precision math built-in functions for DECIMAL data.	See fgl_decimal_truncate() on page 1647, fgl_decimal_sqrt() on page 1647, fgl_decimal_exp() on page 1647, fgl_decimal_logn() on page

Overview	Reference
	1647, fgl_decimal_power() on page 1648.
Automatic Code Completion with VIM: If you have Vim 7 installed, you can now use .per and .4gl code completion.	See Source code edition on page 1516.

Table 42: Reports

Overview	Reference
The START REPORT instruction now allows to specify the XML SAX Document Handler to process XML output with the TO XML HANDLER clause.	See TO XML HANDLER syntax.
Report definition file generation with fglcomp --build-rdd option.	See See fglcomp on page 1497.

Table 43: User interface

Overview	Reference
Support for typical Tree-View widgets with the new TREE container.	See Tree views on page 1384.
The traditional user interface mode: To simplify migration from Informix® 4GL or Four Js BDS, you can now run applications in traditional mode to render windows as simple boxes, as in the WTK front-end.	See Traditional GUI mode on page 753.
Phantom form fields can be used to define the screen-record or screen-array, but are not used in the LAYOUT section of the form. Phantom fields are especially useful when implementing a TREE container.	See Phantom fields on page 861.
Multi-row selection allows end users to highlight several rows in a list of records.	See Syntax of DISPLAY ARRAY instruction on page 1076.
Built-in sort works now in INPUT ARRAY.	See List ordering on page 1356.
New `contextMenu` action default attribute to allow you to specify whether the menu option is visible in the default context menu. The default value is "yes" - the option is visible whenever the action is visible.	See Action defaults files on page 796.
New `integratedSearch` presentation style attribute for TEXTEDIT fields to enable text search.	See TextEdit style attributes on page 834.
FOLDER elements can now use a "`position`" style attribute to define the position (top, left, right, bottom) of folder tabs.	See Folder style attributes on page 827.
BUTTON form items get a new "`buttonType`" attribute to define the rendering of the button.	See Button style attributes on page 821.
MENU object created with the popup option can be placed with the "`position`" style attribute.	See Menu style attributes on page 829.
Window Menu and Action panel decoration can be customized using the new "`ringMenuDecoration`", "`actionPanelDecoration`" style attributes.	See Window style attributes on page 839.
The new "`tabbedContainer`", "`tabbedContainerCloseMethod`" style attributes can be used to turn on and customize tabbed WCI containers.	See Window style attributes on page 839.
TABLE elements can use the new "`tableType`" attribute to render data in different ways. The new "`resizeFillsEmptySpace`" attribute can be used to define how the last column is resized when the table is resized.	See Table style attributes on page 831.

Overview	Reference
All items with an IMAGE attribute can use the new "`imageCache`" attribute to define if the picture can be cached locally on the front-end.	See Common style attributes on page 818.
New Front-End Functions "`getWindowId`", "`feInfo`", "`launchURL`".	See Standard front calls on page 1889.
Front-End protocol compression can now be disabled with a new FGLPROFILE entry. This is especially useful in fast networks to save processor time.	See GUI protocol compression on page 758.
New built-in functions are now available to control the part of the text that is selected in the current field.	See fgl_dialog_getselectionend() on page 1652, fgl_dialog_setselection() on page 1653.
New IMAGE attribute in form LAYOUT element: The LAYOUT section of a form definition can now use the IMAGE attribute to define the icon to be used for the parent Window. This is especially useful in a Container-based application, to distinguish child programs inside the WCI container.	See LAYOUT section on page 907.
Use the new INFIELD clause in ON ACTION interactive block to automatically enable/disable the action when entering/leaving the specified field.	See Field-specific actions (INFIELD clause) on page 1335.
Getting the current active dialog with ui.Dialog.getCurrent().	See ui.Dialog.getCurrent on page 1792.

Table 44: SQL databases

Overview	Reference
New database drivers.	List of new database drivers: • **dbmsqt3xx** for an SQLite 3 library (2.20.01)
MySQL Driver supports TEXT/BYTE data types.	See SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x on page 625.
To work around conflicts with the Informix® database path specification in DBPATH, use the FGLRESOURCEPATH environment variable.	See FGLRESOURCEPATH on page 184.
Database user authentication callback function can be used to specify a database user and password when the DATABASE instruction cannot be replaced by CONNECT TO.	See User authentication callback function on page 474.
FGLSQLDEBUG output is improved to display and SQL command header with SQL command name and source/line information before executing the underlying ODI driver code. If the driver code crashes or stops the process with an assertion, you can easily identify the last SQL instruction that was executed.	See FGLSQLDEBUG on page 185.

Table 45: Web Services

Overview	Reference
The Genero Web Services XML Library has been improved to support the *XML-Signature* and *XML-Encryption* specifications defined by the W3C (also known as *XML-Security*).	See XML security classes on page 2208.

Overview	Reference
The library enables BDL applications to handle public, private, symmetric or hmac keys and X509 certificates in order to sign XML documents or document fragments, and verify a XML signature against a certificate or key. It also enables the applications to encrypt XML nodes using symmetric keys, and decrypt them back using DOM manipulation. Combined with the COM library, any BDL application can now exchange any XML documents over the Internet in a completely secured manner. The library provides classes for: • Manipulating cryptography keys • Handling X509 certificates for identification • Encrypting and decrypting XML documents, document fragments, or symmetric keys • Signing XML documents, document fragments, or any kind of data, and validating them againstXML signatures	
The Genero Web Services XML library provides APIs to encrypt and decrypt strings with symmetric or RSA public/private keys. These APIs can be used to encrypt/decrypt passwords directly in BDL applications.	See The Encryption class on page 2262 and fglpass on page 1506.
The Genero Web Services provides support for the new `BOOLEAN`, `TINYINT` and `BIGINT` data types. You can use these data types when writing your web service or to customize your BDL RECORDs for XML serialization. The `fglwsdl` tool has been enhanced to generate these new data types automatically when encountered in WSDL files or XML schemas. **Note:** For compatibility issues, the `fglwsdl` tool allows code generation without these new data types by using the option '`-legacyTypes`'.	See Attributes to customize XML serialization on page 2517 and fglwsdl on page 1503.

What's new in Genero Business Development Language, v 2.11

This topic lists features added for the 2.11 release of the Genero Business Development Language.

Table 46: Core language

Overview	Reference
New `-p noln` preprocessor option to remove line number information to get a readable output: `fglcomp -E -p noln mymodule.4gl`	See The preprocessor on page 1522
The `-b` option of `fglrun` has been extended to recognize headers of p-code modules compiled with older versions of Genero.	See Module build information on page 1515
The `fglform` compiler now writes build information in the `.42f` files, to identify on the production site what version was used to compile forms.	See Compiling form files on page 1508

Table 47: User interface

Overview	Reference
The `ui.ComboBox` class has been extended with new methods: `getTextOf()` and `getIndexOf()`.	See The ComboBox class on page 1820
A new FGLPROFILE entry has been added to force the current row to be shown automatically after a sort in a table: `Dialog.currentRowVisibleAfterSort = 1` By default, the offset does not change and the current row may disappear from the window. When this new parameter is used, the current row will always be visible.	See Dialog configuration with FGLPROFILE on page 1251

Table 48: SQL databases

Overview	Reference

Static SQL syntax now supports derived tables and derived column lists in the FROM clause. For example:
`SELECT * FROM`
`(SELECT * FROM customer`
`ORDER BY cust_num)`
`AS t(c1,c2,c3,...)`
See database server documentation for more details about this SQL feature.
Informix® 11 does not support the full ANSI SQL 92 specification for derived columns, while other databases like DB2® do. For this reason, fglcomp allows the ANSI standard syntax.
	See SELECT on page 493

The `SET ISOLATION` statement now supports the newInformix® 11 clauses for the `COMMITTED READ` option:
`SET ISOLATION TO COMMITTED READ`
`[LAST COMMITTED]`
`[RETAIN UPDATE LOCKS]`
When connecting to a non-Informix database, the `LAST COMMITTED` and `RETAIN UPDATE LOCKS` are ignored; other databases do not support these options, and have the same behavior as when these options are used with Informix® 11.
	See SET ISOLATION on page 485

The `CAST` operator can now be used in static SQL statements: `CAST ( expression AS sql-data-type )` Only Informix® data types are supported after the AS keyword.	See Static SQL statements on page 486

In order to execute database administration tasks, you can now connect to Oracle as SYSDBA or SYSOPER with the `CONNECT` instruction:
`CONNECT TO "dbname"`
`USER "scott/SYSDBA"`
	See CONNECT TO on page 477

Overview	Reference
`USING "tiger"`	

Table 49: Web Services - Version 2.11.00

Overview	Reference
The Genero Web Services com library provides the HTTPServiceRequest class to perform low-level XML and TEXT over HTTP communication on the server side. This allows communication at a very low-level layer, to write your own type of web services.	See The HTTPServiceRequest class on page 2036.
XML facet constraints attributes: the Genero Web Services XML library provides 12 new XML attributes to map to simple BDL variables. These attributes restrict the acceptable value-space for each variable in different ways such as: • a minimum or a maximum number of XML characters or bytes. • a strict number of XML characters or bytes. • a minimum inclusive or exclusive value depending on the data type. • a maximum inclusive or exclusive value depending on the data type, • a enumeration of authorized values. • a number of digits and fraction digits. • how white spaces have to be handled. • a regular expression to match. (See Section F of XML Schema Part 2)	See Attributes to customize XML serialization on page 2517.
The fglwsdl tool has been enhanced with the following three new options : • `-disk` : to retrieve locally a WSDL or an XSD with all its dependencies from an URL on the disk • `-noFacets` : to avoid the generation of the new facet constrain attributes (for compatibility) • `-regex` : to validate a value against a regular expression as described in the XML Schema specification	See fglwsdl on page 1503.

Table 50: Web Services - Version 2.11.04

Overview	Reference
The Genero Web Services library provides two new methods in the `WebOperation` class to create One-Way operations in services. A One-Way operation means that the server accepts an incoming request, but doesn't return any response back to the client. There is one method called `CreateOneWayRPCStyle` to create an RPC Style operation, and another one called `CreateOneWayDOCStyle` to create a Document Style operation. For instance, a One-Way operation can be used as a logger service, where a client sends a message to the server, but doesn't care about what the server is doing with it.	See The WebOperation class on page 2018.
The `fglwsdl` tool has been enhanced with the following new options: • `-b`: Generate code from a WSDL using the binding section instead of the service section	See fglwsdl on page 1503.

Overview	Reference
• `-autoNsPrefix`: Determine the prefix for variables and types according to the XML namespace they belong to • `-nsPrefix`: Set the prefix for a variable or a type belonging to the given XML namespace The following options have been changed: • `-o`: If there are several services in one WSDL, they will be generated in the same file with the given base name instead of returning an error • `-disk`: Retrieves and displays all dependencies to the current directory but there are no sub directories any longer. • `-prefix`: Accepts patterns %s, %f and %p	
The Genero Web Services library has been enhanced to support WSDL with circular references. The Genero language doesn't provide a way to define variables or types that refer to themselves. However, to provide better interoperability and a way to handle such circular data, the `fglwsdl` tool now generates variables or types of `xml.DomDocument` type when circular references are detected during the processing of WSDL files. This gives the user the ability to manipulate the circular data by hand, using the XML DOM API.	See The xml package on page 2103.

What's new in Genero Business Development Language, v 2.10

This topic lists features added for the 2.10 release of the Genero Business Development Language.

Table 51: Core language

Overview	Reference
The `TRY/CATCH` block can handle exceptions raised by the runtime system.	See TRY - CATCH block on page 344
`WHENEVER ... RAISE` instructs the runtime system that an uncaught exception will be handled by the caller of the function.	See WHENEVER instruction on page 342

NULL point exceptions can now be trapped as other exceptions: Error -8083 will be raised if you try to call an object method with a variable that does not reference an object (that contains NULL):
`DEFINE x ui.Dialog`
`-- x is NULL`
`CALL x.setFieldActive("fieldname",FALSE)`
`  -- raises -8083`
In previous versions, the above code raised a fatal NULL pointer error.
	See OOP support on page 349

The `base.Channel` class now provides a method to establish a client socket connection to a server, with the new `openClientSocket()` method.	See base.Channel.openClientSocket on page 1710
For debugging purpose, get the stack trace of the program with the `base.Application.getStackTrace()` method.	See base.Application.getStackTrace on page 1706
Before version 2.10, it was only possible to assign a `TEXT` to a `TEXT` variable. It is now possible to assign `STRING`, `CHAR` and `VARCHAR` values to a `TEXT` variable.	See Type conversions on page 211

Overview	Reference
The fglrun `-e` option now supports a comma-separated list of extensions, and `-e` can be specified multiple times: `fglrun -e ext1,ext2,ext3 -e ext4,ext5 myprogram`	See Loading C-Extensions at runtime on page 1601
Get an action event when the user modifies the value of a field, with the predefined `dialogtouched` action, to detect first user modifications.	See Immediate detection of user changes on page 1267
The `parse()` and `toString()` methods are now available for a `om.DomNode` object.	See The DomNode class on page 1839
A `om.DomDocument` object can be created with `createFromString()`.	See The DomDocument class on page 1833
The `TEXT` and `BYTE` data types now support the methods `readFile(fileName)` and `writeFile(fileName)`.	See BYTE on page 193, TEXT on page 208

Table 52: User interface

Overview	Reference
The new `DIALOG` instruction handles different parts of a form simultaneously.	See Multiple dialogs (DIALOG) on page 1144
`HBox` and `VBox` containers can now have a splitter.	See SPLITTER attribute on page 985
The new `DOUBLECLICK` table allows to configure the action to be sent when the user double-clicks on a row.	See DOUBLECLICK attribute on page 962
Define a timeout delay for front-end connections with the following FGLPROFILE entry: `gui.connection.timeout = seconds`	See Configure the GUI connection timeout on page 757
Before version 2.10, it was only possible to assign a `TEXT` to a `TEXT` variable. It is now possible to assign `STRING`, `CHAR` and `VARCHAR` values to a `TEXT` variable.	See Type conversions on page 211
Presentation styles have been extended: • The style attribute "`position`" for Windows™ can be set to "`previous`". • `TEXTEDIT` now has the "`textSyntaxHighlight`" attribute (value can be "`per`", more to come...). • All widgets can now use the "`localAccelerators`" global style attribute to interpret standard navigation and editor keys (like Home/ End) without firing an action that uses the same keys as accelerators.	See Presentation styles on page 799
Get an action event when the user modifies the value of a field, with the predefined `dialogtouched` action, to detect first user modifications.	See Immediate detection of user changes on page 1267
Use the `validate="no"` action default attribute to prevent data validation when executing an action.	See Data validation at action invocation on page 1331

Overview	Reference
Define a minimum width and height for forms with the `MINWIDTH`, `MINHEIGHT` attributes.	See MINHEIGHT attribute on page 975, MINWIDTH attribute on page 975
In `INPUT ARRAY`, avoid the automatic creation of a temporary row with the new `AUTO APPEND = FALSE` dialog attribute.	See INPUT ARRAY temporary rows on page 1378

Table 53: SQL databases

Overview	Reference
Support for SQL Server 2005 Native Client is now provided.	See SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 on page 592
The `fgldbsch` tool now supports the `X` conversion code to ignore table columns of a specific type. This is useful for ROWID-like columns such as SQL Server's *uniqueidentifier* columns.	See Data type conversion control on page 365
Before version 2.10, SQL interruption was not supported well for some databases. SQL interruption is now available with all databases providing an API to cancel a long-running query.	See SQL interruption on page 405

Table 54: Web Services

Overview	Reference
The Genero Web Services XML library (`xml`) has been added. This library provides classes and methods to perform: • XML manipulation with a W3C Document Object Model (DOM) API • XML manipulation with a Streaming API for XML (StAX) • Validation of DOM documents against XML Schemas • Serialization of BDL variables in XML • Creation of XML Schemas corresponding to BDL variables	See The xml package on page 2103.
New classes have been added to the Genero Web Services COM library to facilitate low-level XML and TEXT over HTTP and TCP Client communication (`com`). The Genero Web Services `com` library provides two classes, `HTTPRequest` and `HTTPResponse`, to perform low-level XML and TEXT over HTTP communications on the client side. Two more classes, `TCPRequest` and `TCPResponse`, are also provided to perform low-level XML and TEXT over TCP communications on the client side. This allows communication between applications using the core Web technology, taking advantage of the large installed base of tools that can process XML delivered plainly over HTTP or TCP, as well as SOAP over HTTP. Specific streaming methods are also available to improve the communication by sending XML to the network even if the serialization process is not yet finished, as well as for the deserialization process. It is also possible to prevent asynchronous requests from being blocked when waiting for a response, and to perform specific HTTP form encoded requests as specified in HTML4 or XForms1.0.	See The HTTPRequest class on page 2053, The HTTPResponse class on page 2070, The TCPResponse class on page 2092 and The TCPResponse class on page 2092.

Overview	Reference
The `fglwsdl` tool now generates low-level and asynchronous client stubs from the WSDL. The `fglwsdl` tool generates all client stubs with the low-level `HTTPRequest` and `HTTPResponse` classes of the `com` library to perform HTTP communications. The low-level generated stub also takes advantage of the streaming methods, if Document Style or RPC-Literal web services are performed. Streaming is not possible with RPC-Encoded web services, as nodes can have references to other nodes in the XML document, requiring the entire document in memory to perform serialization or deserialization. The `fglwsdl` tool also generates two new BDL functions for each operation of a Web service. These two functions enable you to perform asynchronous web service operation calls by first sending the request, and retrieving the corresponding response later in the application. This allows you to prevent a BDL application from being blocked if the response of a web service operation takes a certain amount of time.	See The HTTPRequest class on page 2053, The HTTPResponse class on page 2070, and fglwsdl on page 1503.
Genero Web Services provides an enhanced `fglwsdl` tool that is able to generate Genero data types from a XML schema. The data types can then be used in your application to be serialized or deserialized in XML. The resulting XML is a valid instance of that XML schema, and validation with a XML validator will succeed.	See fglwsdl on page 1503.

What's new in Genero Business Development Language, v 2.02

This topic lists features added for the 2.02 release of the Genero Business Development Language.

Table 55: Core language

Overview	Reference
Share global variables between the Genero source and the C Extension, by using the `-G` option of `fglcomp`.	See Sharing global variables on page 1611
Customize the runtime system error messages according to the current locale.	See Runtime system messages on page 321
New debugger commands (`ptype`). Avoid switching into debug mode with SIGTRAP (Unix) or CTRL-Break (Windows™) with the new `fglrun.ignoreDebuggerEvent` FGLPROFILE entry.	See The debugger on page 1531

Table 56: User interface

Overview	Reference
Specify a `TABINDEX` of zero to exclude the form item from the tagging list.	See TABINDEX attribute on page 986

Table 57: SQL databases

Overview	Reference
Some common SQL statements have been added to the static SQL syntax, such as `TRUNCATE TABLE`, `RENAME INDEX`, `CREATE` / `ALTER` / `DROP` / `RENAME SEQUENCE`.	See Static SQL statements on page 486
With Oracle, specify the `SELECT` statement producing the unique session identifier which is used for temporary table names.	See Oracle DB specific FGLPROFILE parameters on page 470
To emulate Informix® temporary tables in Oracle, set the `temptables.emulation` parameter to use `GLOBAL TEMPORARY TABLES` instead of permanent tables..	See Using the global temporary table emulation on page 672

What's new in Genero Business Development Language, v 2.01

This topic lists features added for the 2.01 release of the Genero Business Development Language.

Table 58: Core language

Overview	Reference
The `fglcomp` compiler now supports a negative form for `-W` warning arguments.	See Compiling source code on page 1510
When using the `RUN` command, the `ComSpec` environment variable is now used under Windows™ platforms.	See RUN on page 391

Table 59: User interface

Overview	Reference
The layout tag syntax in grids has been extended to support an ending tag to get better control of form layout.	See Layout tags on page 868

Table 60: SQL databases

Overview	Reference
Support for IBM® DB2® V9.x.	See SQL adaptation guide for IBM DB2 UDB 10.x on page 540
Support for PostgreSQL 8.2.x.	See SQL adaptation guide for PostgreSQL 9.x on page 683

What's new in Genero Business Development Language, v 2.00

This topic lists features added for the 2.00 release of the Genero Business Development Language.

Table 61: Core language

Overview	Reference
The runtime system (`fglrun`) now uses shared libraries for database drivers; there is no need to link anymore.	See Database driver specification (driver) on page 462.
The `TYPE` instruction allows to define your own data type structures.	See Types on page 303.

Overview	Reference
File management function library provided as loadable extension.	See The os.Path class on page 1990.
Mathematical function library provided as loadable extension.	See The util.Math class on page 1960.
C extension support has been extended with Informix-like C API functions.	No longer applicable as of Genero 2.51
The runtime system now shares several static elements among all processes, reducing the memory usage. The shared elements are: Data type definitions, string constants and debug information. For example, when a program defines a string containing a long SQL statement, all fglrun processes will share the same string, which is allocated only once.	See Runtime system basics on page 1560.
The `IMPORT` instruction allows to declare a C extension module.	See IMPORT C-Extension on page 371.
New debugger commands (`call`, `ignore`).	See Debugger commands on page 1537.
The `base.Channel` class now has an `isEof()` method to detect end of file.	See Read and write simple lines on page 1720.
Ignoring the CTRL_LOGOFF_EVENT events on Microsoft™ Windows™ platforms.	See Responding to CTRL_LOGOFF_EVENT on page 386.
New built-in function to set an environment variable: `FGL_SETENV()`.	See fgl_setenv() on page 1662.
The XML reader and writer classes have been extended to properly support markup language entities (like HTML's ` `).	See The XmlReader class on page 1871, The XmlWriter class on page 1876.

Table 62: User interface

Overview	Reference
New form item types (i.e. widgets): `SLIDER`, `SPINEDIT`, `TIMEEDIT`.	See ATTRIBUTES section on page 932.
The `WIDTH` and `HEIGHT` attributes can be used for `IMAGE` form items, as a replacement for `PIXELWIDTH`/`PIXELHEIGHT`.	See HEIGHT attribute on page 965, WIDTH attribute on page 999.
New debugger commands (`call`, `ignore`).	See Debugger commands on page 1537.
Presentation styles support now pseudo selectors such as `focus`, `active`, `inactive`, `input`, `display` for fields and `odd` / `even` states for table rows.	See Pseudo selectors on page 802.
New presentation style attributes were added: • `'errorMessagePosition'` can be used for windows to define how the ERROR message must be displayed; • `'highlightTextColor'` for tables allows you to change the color of the selected line; • `'border'` allows you to remove the border of some widgets like button, images;	See Style attributes reference on page 818.

Overview	Reference
• 'firstDayOfWeek' can be used for DateEdit widget to specify the first day of the week in the calendar; • The auto-selection behavior for ComboBoxes and RadioGroup can be changed using 'autoSelectionStart'.	
With X11 or Windows™ TSE environments, you can now automatically start up the front-end with FGLPROFILE entries.	See Automatic front end startup on page 761.
Up to fourth accelerators can now be defined for an action in actions defaults files or in the ACTION DEFAULTS section of form files.	See Defining keyboard accelerators on page 1323.
Specify TTY attributes (COLOR, REVERSE) and conditional TTY attributes (COLOR WHERE) for all type of fields.	See COLOR attribute on page 957, REVERSE attribute on page 981, COLOR WHERE Attribute on page 958.

Table 63: SQL databases

Overview	Reference
Database schema files have been extended to centralize form field definition with the new FIELD item type. **Important:** This feature is deprecated in 2.51 and +.	See FIELD item type.
Call database stored procedures with output parameters with the new IN/OUT keywords.	See EXECUTE (SQL statement) on page 502, Stored procedures on page 441.

Primary key, foreign key and check constraints can be specified in static SQL CREATE TABLE statements:
`CREATE TABLE t1 (`
`col1 INTEGER PRIMARY KEY,`
`col2 CHAR(2),`
`col3 DATE,`
`FOREIGN KEY (col2)`
`REFERENCES t2(col1)`
`)`
	See CREATE TABLE on page 497.

The fgldbsch tool can now extract database tables with LVARCHAR columns. The LVARCHAR type is converted to VARCHAR2(n>255) in the .sch file.	See Data type conversion control on page 365.

Table 64: Web Services

Overview	Reference
You can now choose to use Document Style Service (Doc/Literal) or RPC Literal Style Service (RPC/Literal) with Genero Web Services (GWS), for .NET compatibility and WS-I compatibility (standards defined by the Web Services Interoperability organization). • Document Style Service allows you to exchange complex data structures, such as database tables or word processing documents (MS.Net default)	See Choosing a web services style on page 2483 and Writing a Web server application on page 2473.

Overview	Reference
• RPC Literal Style Service is usually used to execute a function, such as a service that returns a stock option **Note:** RPC/Encoded Style Service (Traditional SOAP section 5) is available for backwards compatibility.	
Genero Web Services now provides a tool, `fglwsdl`, to allow a Genero application that is accessing a Web Service to obtain the WSDL information for the service. It does not matter what language the Web Service is written in. The `fglwsdl` tool is installed in Genero as part of the Genero Web Services package.	See fglwsdl on page 1503.
You no longer need to create a runner that includes the Genero Web Services package. Instead, your applications import the Genero Web Services library named `com`. This library provides classes and methods that allow you to perform tasks associated with creating GWS Servers and Clients, and managing the Web Services.	See The com package on page 2009.
GWS now supports SOAP header management through the `CreateHeader` method in the Web Service class that is part of the Web Services library (`com`).	See The WebService class on page 2009.
HTTPS support has been added on the client side. GWS supports secure communications through the use of encryption and standard X.509 certificates. Based on the OpenSSL engine, new security features allow a Web Services client to communicate with any secured server over HTTP or HTTPS. A new tool is provided, `fglpass`, allowing you to encrypt a password from a standard X.509 certificate, and to decrypt a password you previously encrypted with a certificate. Entries in the `FGLPROFILE` file are used to define the configuration for client security.	See fglpass on page 1506, Encryption, base64 and password agent with fglpass tool on page 2429, and The FGLPROFILE file on page 164.
You can configure a GWS Client to connect via an HTTP proxy by adding an entry in the `FGLPROFILE` file.	See Configure a WS client to connect via an HTTP Proxy on page 2463.
You can define multiple Web Services in a single Genero DVM. When you start the Web Services engine, all registered Web Services are started.	See The WebServiceEngine class on page 2025.
You can remap the location of Genero Web Services using entries in the `FGLPROFILE` file, depending on the network configuration and the access rights management of the deployment site.	See Using logical names for service locations on page 2462.
Serializing Genero data types: you can add optional attributes to the definition of data types. You can use these attributes to map the BDL data types in a Genero Web Services Client or Server application to their corresponding XML data types.	See Attributes to customize XML serialization on page 2517.
The `WSHelper.42m` library file contains internal BDL functions to handle SOAP requests and errors. The file is provided in the `$FGLDIR/lib` directory of the Genero Web Services package, and should be linked into every Genero Web Services Server or Client program.	See Compiling the client application on page 2453 and Compiling GWS server applications on page 2480.

What's new in Genero Business Development Language, v 1.33

This topic lists features added for the 1.33 release of the Genero Business Development Language.

Table 65: Core language

Overview	Reference
New `base.TypeInfo` built-in class to serialize program variables.	See The TypeInfo class on page 1752
The `base.Channel` class now supports a binary mode with the `'b'` option, to control CR/LF translation when using DOS files.	See Line terminators on Windows and UNIX on page 1721

Table 66: User interface

Overview	Reference
Up to three accelerators can now be defined for an action in actions defaults files or in the `ACTION DEFAULTS` section of form files.	See Defining keyboard accelerators on page 1323

Table 67: SQL databases

Overview	Reference
Generic ODBC database driver is now available (code is generic ODBC database driver is now available (code is `odc`).	See Database driver specification (driver) on page 462
MySQL version 5.0.x is now supported.	See SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x on page 625
PostgreSQL version 8.1.x is now supported.	See SQL adaptation guide for PostgreSQL 9.x on page 683
Microsoft™ SQL Server 2005 is now supported.	See SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 on page 592
Pre-fetch rows by block with SQL Server to get better performance. Use the following FGLPROFILE entry to specify the maximum number of rows the driver can pre-fetch: `dbi.database.`*`dbname`*`.msv.prefetch.rows = `*`count`* See "Database vendor specific parameters" in Connections for more details.	See SQL Server (Native Client driver) specific FGLPROFILE parameters on page 471

What's new in Genero Business Development Language, v 1.32

This topic lists features added for the 1.32 release of the Genero Business Development Language.

Table 68: Core language

Overview	Reference
New debugger commands (`watch` with condition, `whatis`).	See Debugger commands on page 1537
The preprocessor is now part of the compilers and is always enabled. Preprocessing directives start with an ampersand character (`&`).	See The preprocessor on page 1522

Table 69: User interface

Overview	Reference
New built-in functions to transfer files from/to the front-end.	See fgl_getfile() on page 1656, fgl_putfile() on page 1660

Table 70: SQL databases

Overview	Reference
PostgreSQL version 8.0 is now supported (8.0.2 and higher).	See SQL adaptation guide for PostgreSQL 9.x on page 683

What's new in Genero Business Development Language, v 1.31

This topic lists features added for the 1.31 release of the Genero Business Development Language.

Table 71: Core language

Overview	Reference
C extensions can be loaded dynamically, no need to re-link runner.	See C-Extensions on page 1597
The `FGL_WIDTH()` built-in function computes the number of print columns needed to represent a single or multi-byte character.	See fgl_width() on page 1663

Table 72: User interface

Overview	Reference
GUI protocol compression for slow networks.	See GUI protocol compression on page 758
Interruption handling with SSH port forwarding - only supported with GDC 1.31!	See User interruption handling on page 1252
New method `ui.Form.setFieldStyle()` to set a style for a field.	See ui.Form.setFieldStyle on page 1782
Improved front-end identification when connecting to GUI client.	See Establish a GUI front-end connection on page 755

Table 73: SQL databases

Overview	Reference
MySQL version 4.1.x is now supported, 3.23 is desupported.	See SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x on page 625
Oracle version 10g is now supported.	See SQL adaptation guide for Oracle Database 11, 12 on page 643

What's new in Genero Business Development Language, v 1.30

This topic lists features added for the 1.30 release of the Genero Business Development Language.

Table 74: Core language

Overview	Reference
First version of integrated preprocessor using # sharp syntax for macros. Version 1.32 uses & instead	See The preprocessor on page 1522
Localization support (multi-byte character sets).	See Localization on page 307

The `fglcomp` compiler now adds build information in 42m modules. Compiler version of a 42m module can be checked on site by using the `fglrun` with the `-b` option (line break added for documentation only):
`$ fglrun -b module.42m`
`2004-05-17 10:42:05 1.30.2a-620.10 /devel/tests/module.4gl`
	See Module build information on page 1515

The `fglmkmsg` tool now has the same behavior as other tools like `fglcomp` and `fglform`: If you give only the source file, the message compiler uses the same file name for the compiled output file, adding the `.iem` extension.	See Compiling message files on page 795
New `BREAKPOINT` instruction to stop a program at a given position when using the debugger. It is ignored when not running in debug mode.	See Setting a breakpoint programmatically on page 1536
New assignment operator `:=` has been added to the language. Assign variables directly within expressions: `IF (i:=(j+1))==2 THEN`	See Assignment (:=) on page 259
New `fglcomp` compiler option to detect non-standard SQL syntax: `fglcomp -W stdsql module.4gl`	See SQL portability on page 412
New method `base.StringBuffer.replace()`, to replace a sub-string in a string: `CALL s.replace("old","new",2)` Replaces two occurrences of "old" with "new"...	See base.StringBuffer.replace on page 1745
New methods to read/write complete lines in `base.Channel` built-in class: `readLine()` and `writeLine()`.	See Read and write simple lines on page 1720
The FGLLDPATH variable is now used during program linking.	See Compiling source files on page 1508
The linker option `-O` (optimize) is de-supported (was ignored before). You now get a warning if you use this option.	See Linking programs on page 1512

The `[]` array sub-script operator now returns the sub-array:
`DEFINE a2 DYNAMIC ARRAY`
`  WITH DIMENSION 2 OF INTEGER`
`LET a2[5,10] = 123`
`DISPLAY a2.getLength() -- displays 5`
`DISPLAY a2[5].getLength() -- displays 10`
	See Arrays on page 296

Table 75: User interface

Overview	Reference
New layout rules and form item attributes provide better control of form design.	See Form rendering on page 1002
Decoration attribute can be defined in a presentation style file to set fonts and colors.	See Presentation styles on page 799
Action defaults can be specified in forms in the `ACTION DEFAULTS` section.	See ACTION DEFAULTS section on page 903
New `ui.Dialog` built-in class to provide better control over interactive instructions.	See The Dialog class on page 1784
`COMBOBOX` fields now support `UPSHIFT` and `DOWNSHIFT` attributes, to force character case when `QUERYEDITABLE` is used.	See QUERYEDITABLE attribute on page 979
New presentation style attribute `highlightCurrentRow` for Tables, to indicate if the current row must be highlighted in a specific mode. By default, the current row is highlighted during a `DISPLAY ARRAY`.	See Table style attributes on page 831
New method `appendElement()` for `ARRAYs`, to append an element at the end of a dynamic array.	See Array methods on page 302
New assignment operator `:=` has been added to the language. Assign variables directly within expressions: `IF (i:=(j+1))==2 THEN`	See Assignment (:=) on page 259
The new method `ui.Dialog.setCellAttributes()` lets you define colors for each cell of a table.	See Cell color attributes on page 1380
The `ui.Window` class provides new methods to create or get a form object.	See ui.Window methods on page 1769
When using a dynamic array in `INPUT ARRAY` or `DISPLAY ARRAY`, the number of rows is defined by the size of the dynamic array. The `SET_COUNT()` or `COUNT` attributes are ignored.	See Controlling the total number of rows on page 1350
The new form field attribute `TITLE` can be used to specify a table column label with a localized string.	See TITLE attribute on page 988
New class method `ui.Dialog.setDefaultUnbuffered()` to set the default for the `UNBUFFERED` mode.	See The buffered and unbuffered modes on page 1262
Action defaults are now applied at element creation by the runtime system. In previous versions this was done dynamically by the front-end. Now, changing an action default node at runtime has no effect on existing elements.	See Configuring actions on page 1318
The `DATEEDIT` field type now supports DBDATE/CENTURY settings and the `FORMAT` attribute.	See FORMAT attribute on page 963
New default action 'close' to control window closing: `ON ACTION close`	See Implementing the close action on page 1337
`INPUT ARRAY` using `TABLE` container now needs `FIELD ORDER FORM` attribute to keep tabbing order consistent with visual order of columns.	See Defining the tabbing order on page 1271
New instructions `ACCEPT INPUT` / `ACCEPT CONSTRUCT` / `ACCEPT DISPLAY` to validate a dialog by program. `ON ACTION doit`	See ACCEPT INPUT instruction on page 1072, ACCEPT DISPLAY instruction on page 1095, ACCEPT

Overview	Reference
`ACCEPT INPUT`	CONSTRUCT instruction on page 1139
New dialog attribute `ACCEPT` / `CANCEL` to avoid creation of default actions 'accept' and 'cancel'.	See INPUT instruction configuration on page 1065
New default action 'append' in `INPUT ARRAY`. Allows you to add a row at the end of the list.	See Default actions in INPUT ARRAY on page 1105
New method `ui.Window.createForm()` to create an empty form object in order to build forms from scratch at runtime.	See ui.Window.createForm on page 1770
`TOPMENU` definition in forms now allows attributes in parenthesis.	See TOPMENU section on page 903
The form layout syntax now allows you to specify the real width of form items by using a dash '-' in the layout tag.	See Widget size within hbox tags on page 1016
Important remark: Before build 530 the `MENU` has attached the window when returning from the `BEFORE MENU` actions. Since build 530 the `WINDOW` must exist before the `MENU` statement. So now the `Menu` AUI tree node is available in the `BEFORE MENU` block, but a window opened or made current in the `BEFORE MENU` block will NOT be used.	
Layout `GRID` now accepts HBox tags to group items horizontally.	See Hbox tags on page 875
Form `VERSION` attribute to distinguish form revisions.	See VERSION attribute on page 994
Form layout `SPACING` attribute to define space between widgets.	See SPACING attribute on page 984
Form `DEFAULT SAMPLE` instruction to define a default sample attribute for all form fields.	See INSTRUCTIONS section on page 950
New form item attributes: `SAMPLE`, `JUSTIFY`, `SIZEPOLICY` ...	See SAMPLE attribute on page 981, JUSTIFY attribute on page 972, SIZEPOLICY attribute on page 982
To hide form elements by default, that can be shown by the end user by option, use `HIDDEN=USER` as 'hidden to the user by default'.	See HIDDEN attribute on page 965
Individual table columns now have new attribute `UNMOVABLE` to avoid moving.	See UNMOVABLE attribute on page 990
`WANTCOLUMNSANCHORED` replaced by `UNMOVABLECOLUMN` and `WANTCOLUMNSVISIBLE` replaced by `UNHIDABLECOLUMNS`.	See UNMOVABLECOLUMNS attribute on page 990, UNHIDABLECOLUMNS attribute on page 990
Tables now accept a `WIDTH` and `HEIGHT` attribute to specify a size.	See WIDTH attribute on page 999, HEIGHT attribute on page 965
New `RADIOGROUP` attribute to define the orientation of the radio buttons: `ORIENTATION`.	See ORIENTATION attribute on page 977
The `MENU COMMAND` clause now generates action names in lowercase. This means, when you define `COMMAND "Open"`, it will bind to all actions views defined with the name 'open'.	See COMMAND [KEY()] "option" block on page 1054

Overview	Reference
New `ui.Interface.loadTopMenu()` method to load a global `TOPMENU`.	See ui.Interface.loadTopMenu on page 1765
The `ON CHANGE` block is now invoked when the user clicks on a `CHECKBOX`, `RADIOGROUP`, or changes the item in a `COMBOBOX`.	See ON CHANGE block on page 1069
New `DIALOG` keyword to reference the current dialog as a `ui.Dialog` object. This can be used for example to enable/disable fields durint the dialog execution.	See The Dialog class on page 1784

The `ui.Form` built-in class has new methods to handle form elements. The hidden attribute is now also managed at the model level, this allows you to hide form fields by name, instead of using the decoration node.
`CALL myform.setElementHidden("formonly.field1",2)`
`CALL myform.setFieldHidden("field1",2)`
`-- prefix is optional`
	See The Form class on page 1774

New methods are provided in `ui.Interface` to control the MDI children.	See Window containers (WCI) on page 1458
In `INPUT ARRAY`, `CANCEL INSERT` now supported in `AFTER INSERT`, to remove the new added line when needed.	See CANCEL INSERT instruction on page 1120
`TOOLBAR` and `TOPMENU` elements now have the hidden attribute so you can create them and hide the options the user is not supposed to see. **Important:** Hiding a toolbar or topmenu option does not prevent the use of the accelerator of the action. Use `ui.Dialog.setActionActive()` to disable an action.	See ui.Form.setElementHidden on page 1779
New option `NEXT FIELD CURRENT` to gives control back to the dialog instruction without moving to another field.	See Giving the focus to a form element on page 1272

Table 76: SQL databases

Overview	Reference
Support for PostgreSQL 7.4 with parameterized queries.	See SQL adaptation guide for PostgreSQL 9.x on page 683
A MySQL 3.23 driver is now provided for Windows™ platforms (was previously only provided on Linux™).	See SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x on page 625

The `fglcomp` compiler now converts static SQL updates like:
`UPDATE tab SET (c1,c2)=(v1,c2) ...`
to a standard syntax:
`UPDATE tab SET c1=v1, c2=v2 ...`
	See UPDATE on page 490

On Windows™ platforms only, the ix drivers automatically set standard Informix® environment variables with ifx_putenv(). Values are taken	See

Overview	Reference

from the console environment with getenv(). Additional variables can be specified with:
`dbi.stdifx.environment.count = n`
`dbi.stdifx.environment.xx = "variable"`	

What's new in Genero Business Development Language, v 1.20

This topic lists features added for the 1.20 release of the Genero Business Development Language.

Table 77: Core language

Overview	Reference
Integrated debugger with gdb syntax to interface with graphical tools like ddd.	See The debugger on page 1531.
The program profiler can be used to generate statistics of program execution, to find the bottlenecks in the source code.	See The profiler on page 1556.
Internationalize your application in different languages with localized strings, by using the %"string" notation.	See Localized strings on page 327.
The `TERMINATE REPORT` and `EXIT REPORT` can be used in reports to respectively stop a report from outside of the `REPORT` routine, or stop the report from inside the `REPORT` routine.	See TERMINATE REPORT on page 1468, EXIT REPORT on page 1480.
The `fgl_getversion()` function returns the version number of the runtime system.	See fgl_getversion() on page 1657.
Static arrays can be passed as parameters: all elements are expanded.	See Static arrays on page 298.
New methods for `StringBuffer` class: `base.StringBuffer.replaceAt()` and `base.StringBuffer.insertAt()`.	See The StringBuffer class on page 1738.
Operators equal (= or ==) and not equal (<> or !=) now can be used with records: All record members will be compared. If two members are `NULL` the result of this member comparison results in `TRUE`.	See DEFINE ... RECORD on page 294.
New `-W` option for fglform to show warnings.	See fglform on page 1495.
`LSTR()` operator, to get a localized string by name. Useful when the localized string identifier is known at runtime only.	See LSTR() on page 253.
`SFMT()` operator, to format strings with parameter placeholders. Useful to localize application messages with parameters.	See SFMT() on page 253.
The `base.StringTokenizer` class can be used to parse strings for tokens.	See The StringTokenizer class on page 1749.
`CONSTANT` language elements can now be defined as `GLOBALS`.	See Constants on page 291.
The `base.Application` class provides an interface to the program properties.	See The Application class on page 1703.
Review of the definition of `base.Channel` class, now based on objects.	See The Channel class on page 1707.

Table 78: User interface

Overview	Reference
Interactive instructions support the UNBUFFERED mode, to synchronise data model and view automatically: When you set a variable, the value is automatically displayed to the field, and when the user fires and action, the field value is automatically assigned to the corresponding program variable.	See The buffered and unbuffered modes on page 1262.
DISPLAY ARRAY can now work in paged mode, to avoid loading a large array of rows, with the ON FILL BUFFER clause.	See Paged mode of DISPLAY ARRAY on page 1374.
Centralize default attributes for actions in ACTION DEFAULTS.	See Configuring actions on page 1318.
Client side settings can now be saved by application name, with a specific API. By default it is the the name of the program.	See ui.Interface.setName on page 1766.
New attribute APPEND ROW = TRUE/FALSE attribute for the INPUT ARRAY instruction, to control the creation of the default append action.	See INPUT ARRAY row modifications on page 1377.
New attribute KEEP CURRENT ROW = TRUE/FALSE for the DISPLAY ARRAY and INPUT ARRAY instructions, to defines if the current row must remain highlighted when leaving the dialog. The default is FALSE.	See Handling the current row on page 1352
You can now define a TOOLBAR in form specification files.	See TOOLBAR section on page 905.
You can now define a TOPMENU in form specification files.	See TOPMENU section on page 903.
The fgl_gethelp() function returns the help text for the given help number.	See fgl_gethelp() on page 1655.
The fgl_set_arr_curr() function changes the current row in DISPLAY ARRAY or INPUT ARRAY.	See Handling the current row on page 1352.
Users can now send an interruption event to the program,to stop long running SQL queries, processing loops and reports.	See User interruption handling on page 1252.
The statusBarType window style attribute to define the statusbar layout.	See Window style attributes on page 839.
The new FIELD ORDER FORM option can be used to follow the new TABINDEX attribute, to define the field tabbing order. FIELD ORDER FORM can also be used at the dialog level as dialog attribute.	See Defining the tabbing order on page 1271.
For COMBOBOX form items, a default ITEMS list is created by fglform when an INCLUDE attribute is used.	See COMBOBOX item type on page 881.
The ON IDLE clause can be used to execute a block of instructions after a timeout.	See Get program control if user inactivity on page 1254.
New logical order of execution for INPUT ARRAY triggers: **1.** BEFORE INPUT **2.** BEFORE ROW **3.** BEFORE INSERT **4.** BEFORE FIELD	See Editable record list (INPUT ARRAY) on page 1098.
New ui.ComboBox class to configure COMBOBOX fields at runtime.	See The ComboBox class on page 1820.

Overview	Reference
`DISPLAY ARRAY` and `INPUT ARRAY` instructions now automatically use two predefined actions `nextrow` and `prevrow`, which allow binding action views for navigation.	See Predefined actions on page 1338.
`ON CHANGE` field trigger can be used to detect field modification. Useful for fields such as `CHECKBOX` and `COMBOBOX`.	See Reacting to field value changes on page 1267.
Program icon definition with `ui.Interface.setImage()`.	See ui.Interface.setImage on page 1766.
`LABEL` fields can now have a `FORMAT` attribute.	See LABEL item type on page 890.
Front-end function calls allow to execut code on the front-end side with the `ui.Interface.frontCall()` method.	See Front calls on page 395.
New ui.Form built-in class to handle forms.	See The Form class on page 1774.
New `ON ROW CHANGE` clause in `INPUT ARRAY`, executed when if at least one value in the row has been modified, and the user moves to another row or validates the dialog. The `ON ROW CHANGE` block is executed before the `AFTER ROW` block.	See ON ROW CHANGE block on page 1110.
`MENU` instruction now supports `ON ACTION` clause, to write abstract menus as simple action handlers.	See Ring menus (MENU) on page 1048.
New 'help' predefined action, to start help viewer for `HELP` clauses in dialog instructions.	See Predefined actions on page 1338.

Table 79: SQL databases

Overview	Reference
SQL Server driver now supports the TINYINT data type.	See NUMERIC data types on page 602.
The fglcomp compiler supports now ANSI outer join syntax in SQL statements (`LEFT OUTER JOIN`), to replace the Informix specific `OUTER()` syntax.	See SELECT on page 493.
`FOREACH` that raises an error no longer loops infinitely.	See FOREACH (result set cursor) on page 512.
New `SQLSTATE` and `SQLERRMESSAGE` registers, to give SQL execution information.	See SQL error identification on page 402.

What's new in Genero Business Development Language, v 1.10

This topic lists features added for the 1.10 release of the Genero Business Development Language.

Table 80: Core language

Overview	Reference
The language supports now built-in classes, a new object-oriented way to program in BDL.	See OOP support on page 349.
`CONSTANT` keyword to define constants in your programs.	See Constants on page 291.

Overview	Reference
The language now supports dynamic arrays with automatic memory allocation.	See Dynamic arrays on page 300.
A set of XML Utilities are provided in the runtime library as built-in classes.	See The om package on page 1833.
The `STRING` data type can be used to manipulate character strings without a length limit as with `CHAR`/`VARCHAR`.	See STRING on page 206.

Table 81: User interface

Overview	Reference
The Dynamic User Interface is the major new concept in Genero. It is the basement for the new graphical user interface.	See User interface basics on page 747.
Compared to classic IBM Informix 4gl, interactive instructions such as `INPUT`, `DISPLAY ARRAY`, have been extended with new control blocks and control instructions.	See Dialog instructions on page 1034.
Form specification files (.per) support now extended layout definition with the `LAYOUT` section.	See Form definitions on page 769.
Defining Window Containers (a.k.a. MDI) is a simple way to group programs.	See Window containers (WCI) on page 1458.

Table 82: SQL databases

Overview	Reference
The new `SCHEMA` instruction allows you to specific a database schema, without having an implicit connection, when the program executes.	See Database schema on page 355.

Frequently asked questions

The FAQ lists those questions frequently asked when migrating an existing 4GL application to Genero.

- FAQ001: Why do I have a different display with Genero than with BDS V3? on page 85
- FAQ002: Why does an empty window always appear? on page 86
- FAQ003: Why do some COMMAND KEY buttons no longer appear? on page 86
- FAQ004: Why aren't the elements of my forms aligned properly? on page 87
- FAQ005: Why doesn't the ESC key validate my input? on page 88
- FAQ006: Why doesn't the Ctrl-C key cancel my input? on page 88
- FAQ007: Why do the gui.* FGLPROFILE entries have no effect? on page 89
- FAQ008: Why do I get invalid characters in my form? on page 89
- FAQ009: Why do large static arrays raise a stack overflow? on page 89
- FAQ010: Why do I get error -6366 "Could not load database driver drivername"? on page 90

FAQ001: Why do I have a different display with Genero than with BDS V3?

Explanation

Genero Business Development Language (BDL) introduces major graphical user interface enhancements that sometimes require code modification. With BDS V3, application windows created with the `OPEN`

WINDOW instruction were displayed as static boxes in the main graphical window. In the GUI mode of Genero, application windows are displayed as independent, re-sizeable graphical windows.

Solution:
Review the program logic to reduce the number of windows created by the programs. Replace MENU created in specific windows by TOPMENU elements in your forms.

FAQ002: Why does an empty window always appear?

Description

An additional empty window appears when I explicitly create a window with the OPEN WINDOW instruction.

```
MAIN
  OPEN WINDOW w1 AT 1,1 WITH FORM "form1"
  MENU "Example"
    COMMAND "Exit"
      EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

Explanation

In the new standard GUI mode, all windows are displayed as real front-end windows, including the default SCREEN window. When an application starts, the runtime system creates this default SCREEN window, as in version 3. This is required because some applications use the SCREEN window to display forms (they do not use the OPEN WINDOW instruction to create new windows). To facilitate BDS V3 to Genero migration, the runtime system must keep the default SCREEN window creation; otherwise, existing applications would fail if their code was not modified.

Solution

You can either execute a CLOSE WINDOW SCREEN at the beginning of the program, to close the default window created by the runtime system, or use the OPEN FORM + DISPLAY FORM instructions, to display the main form in the default SCREEN window.

Example

```
MAIN
  OPEN FORM f FORM "form1"
  DISPLAY FORM f
  MENU "Example"
    COMMAND "Exit"
      EXIT MENU
  END MENU
END MAIN
```

FAQ003: Why do some COMMAND KEY buttons no longer appear?

Description

When creating a MENU with COMMAND KEY(*keyname*) "*option*" clause, the button for *keyname* is no longer displayed:

```
MAIN
  MENU "Example"
```

```
    COMMAND "First"
       EXIT PROGRAM
    COMMAND KEY (F5) "Second"
       EXIT PROGRAM
    COMMAND KEY (F6) -- Third is a hidden option
       EXIT PROGRAM
  END MENU
END MAIN
```

Explanation

In BDS Version 3, when using the `MENU` instruction, several buttons are displayed for each clause of the type `COMMANDKEY(`*`keyname`*`) "`*`option`*`"`: one for the menu option, and others for each associated key.

When using Genero, for a named `MENU` option defined with `COMMAND KEY`, the buttons of associated keys are no longer displayed (F5 in our example), because there is already a button created for the named menu option. The so called "hidden menu options" created by a `COMMAND KEY(`*`keyname`*`)` clause (F6 in our example) are not displayed as long as you do not associate a label, for example with the `FGL_SETKEYLABEL()` function.

FAQ004: Why aren't the elements of my forms aligned properly?

Description

In my forms, I used to align labels and fields by character, for typical terminal display. But now, when using the new `LAYOUT` section, some elements are not aligned as expected. In this example, the beginning of the field `f001` is expected in the column near the end of the digit-based text of the first line, but the field is actually displayed just after the label "`Name:`":

```
DATABASE FORMONLY

LAYOUT
  GRID {
  01234567890123456789
   Name:             [f001       ]
  }
  END
END

ATTRIBUTES
  f001 = formonly.field1 TYPE CHAR;
END
```

Explanation

By default, Genero displays form elements with proportional fonts, using layout managers to align these elements inside the window. In some cases, this requires a review of the content of form screens when using the new layout management, because the layout is based on new alignment rules which are more abstract and automatic than the character-based grids in Version 3.

In most cases, the fglform compiler is able to analyze the layout section of .per form specification file in order to produce an acceptable presentation, but sometimes you will have to touch the form files to give hints for the alignment of elements.

Solution

In this example, the field `f001` is aligned according to the label appearing on the same line. By adding one space before the field position, the form compiler will understand that the field must be aligned to the text in the first line:

```
DATABASE FORMONLY

LAYOUT
 GRID {
 01234567890123456789
  Name:              [f001       ]
 }
 END
END

ATTRIBUTES
  f001 = formonly.field1 TYPE CHAR;
END
```

In the next example, the fields are automatically aligned to the text in the first line:

```
DATABASE FORMONLY

LAYOUT
 GRID {
                First         Last
   Name:        [f001      ]  [f002       ]
 }
 END
END

ATTRIBUTES
  f001 = formonly.field1 TYPE CHAR;
  f002 = formonly.field2 TYPE CHAR;
END
```

FAQ005: Why doesn't the ESC key validate my input?

Description

The traditional ESC (escape) key does not validate an `INPUT`, it cancels the dialog instead.

Explanation

To follow platform standards (like Microsoft™ Windows™ for example), the ESC key as the standard key to cancel the current interactive statement.

Solution

You can change the accelerator keys for the 'accept' action with action defaults. However, is not recommended to change the defaults, because ESC is the standard key to be used to cancel a dialog in GUI applications.

FAQ006: Why doesn't the Ctrl-C key cancel my input?

Description

The traditional Ctrl-C key does not cancel an `INPUT` statement.

Explanation

To follow platform standards (like Microsoft™ Windows™ for example), the Ctrl-C key is used as the standard key to copy the current selected text to the clipboard, for cut and paste.

Solution

You can change the accelerator keys for the 'cancel' action with action defaults. However, is not recommended to change the defaults, because ESC is the standard key to be used to cancel a dialog in GUI applications.

FAQ007: Why do the gui.* FGLPROFILE entries have no effect?

Description

The `gui.*` and some other FGLPROFILE entries related to graphics no longer have effect.

Explanation

These entries are related to the old user interface. They are no longer supported. In BDS version 3, the `gui.*` entries were interpreted by the front end. As the user interface has completely been redesigned in Genero, some `gui.*` entries have been removed.

Solution:

Review all FGLPROFILE entries used in your current application and verify if there is a replacement.

FAQ008: Why do I get invalid characters in my form?

Description

The application starts, connects to the database and seams to work properly, but strange symbols (rectangles, question marks) are displayed in the forms for non-ASCII characters. The ASCII characters display properly.

Explanation

The is certainly a character set configuration mistake.

Solution

You have probably defined a wrong runtime system locale or the database client locale.

FAQ009: Why do large static arrays raise a stack overflow?

Description

When using very large static arrays (DEFINE a1 ARRAY[10000] OF ...), I get a stack overflow on Windows™ platforms.

Explanation

The runtime system uses the default stack size defined by the C compiler. Because function static arrays are allocated on the C stack, using very large static arrays in functions can result in a stack overflow error.

Solution

Review the program and use dynamic array instead of static arrays.

FAQ010: Why do I get error -6366 "Could not load database driver *drivername*"?

Description

Error **-6366** occurs when the runtime system fails to load the specified database driver.

Explanation

The database driver shared object (.so or . DLL) or a dependent library could not be found.

Solution

Make sure that the specified driver name does not have a spelling mistake. If the driver name is correct, there is probably an environment problem. Make sure that the database client software is installed on the system (Genero does not communicate directly with the database server, you need the client library). Check the UNIX™ LD_LIBRARY_PATH environment variable or the PATH variable on Windows™. These must point to the database client libraries. Another common error is the installation of a database client software of a different object type as the Genero runtime system. For example, if you install a 32 bit Genero version, you must install a 32 bit version of the database client software, the 64 bit version will not work.

Upgrade Guides for Genero BDL

Each upgrade guide is an incremental upgrade guide that covers only topics related to a specific version of Genero. It is important that you read all of the upgrade guides that sit between your existing version and the desired version.

- General upgrade guide on page 90
- 3.00 upgrade guide on page 91
- 2.51 upgrade guide on page 101
- 2.50 upgrade guide on page 106
- 2.40 upgrade guide on page 109
- 2.32 upgrade guide on page 112
- 2.30 upgrade guide on page 113
- 2.21 upgrade guide on page 116
- 2.20 upgrade guide on page 117
- 2.11 upgrade guide on page 122
- 2.10 upgrade guide on page 123
- 2.02 upgrade guide on page 123
- 2.01 upgrade guide on page 123
- 2.00 upgrade guide on page 123
- 1.33 upgrade guide on page 129
- 1.32 upgrade guide on page 129
- 1.31 upgrade guide on page 130
- 1.30 upgrade guide on page 130

General upgrade guide

These topics describe general considerations when upgrading to a new version of Genero BDL.

1. Runtime system and front-end compatibility on page 91
2. P-Code compatibility accross versions on page 91
3. Genero Web Services migration notes on page 91

Runtime system and front-end compatibility
For better compatibility and GUI related bug fixes, use front-end and runtime system with the equivalent version number.

Graphical User Interface new features and bug fixes usually require modifications inside the runtime system (`fglrun`) and front-ends (GDC, GWC, GMA, GMI).

When upgrading the runtime system to the latest version, we strongly recommend that you upgrade to the latest front-end version as well. For example, when upgrading to a runtime system 2.50.xx, upgrade front-ends to the latest available 2.50.xx version.

P-Code compatibility accross versions
P-Code incompatibility (within .42m files) may be introduced from version to version.

Recompilation is only needed when the p-code becomes incompatible. When executing a program with and older p-code version as expected, `fglrun` will raise the error -6201.

Recompile your .4gl sources when upgrading to a new features release. Recompilation is not required when upgrading to a bug-fix release.

Feature and bug-fix releases are distinguished by the product version number. The product version number can be found by executing the `fglrun` command with the `-V` option.

The product version number has the following format: `M.FF.BB`, where `M` stands for the major release number, `FF` is a feature number, and `BB` is the bug-fix number. For example: `2.31.14`.

- A new feature release is identified by the `M.FF` part of the product version number.
- A bug-fix release is idenfitied by the `BB` part of the product version number.

For example, you must recompile your sources when upgrading from 2.50 to 3.00, from 2.40 to 2.50, or from 2.51. to 2.52.

Genero Web Services migration notes
Upgrade notes for Genero Web Services are available in the chapter dedicated to web services programming.

For more details see Migration notes for Genero Web Services.

3.00 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 3.00.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Check prior upgrade guides if you migrate from an earlier version.

1. Form definitions for mobile applications on page 92
2. Desupported database drivers on page 92
3. Web Services changes on page 93
4. Oracle DB NUMBER type on page 95
5. Oracle DB scroll cursor emulation removal on page 95
6. MySQL VARCHAR size limit on page 95
7. MySQL DATETIME fractional seconds on page 96
8. PostgreSQL DATETIME type mapping change on page 96
9. MariaDB support on page 97
10. FreeTDS driver supports SQL Server 2008, 2012, 2014 on page 97
11. FGL_GETVERSION() built-in function on page 97
12. Built-in front-end icons desupport on page 97
13. Modifications in front calls on page 98
14. SERIAL emulation with SQL Server on page 98
15. Improved compilation time on page 98
16. Preprocessor changes on page 99

17. Current system time in UTC on page 100
18. Structured ARRAYs in list dialogs on page 100

Form definitions for mobile applications
Genero version 3 supports grid-based layout with all front-ends, and introduces STACK layout.

Support for grid-based and stack-based layout

Before Genero version 3.00 (i.e., with Genero Mobile version 1.1), the GMI front-end could only support a stack-based layout and it was required to create different forms for iOS apps and other front-ends supporting grid-based layout. In fact, to get a stack-based layout, grid-based `.per` forms were automatically transformed on the fly when displayed on the GMI front-end.

Starting with Genero 3.00, all mobile front-end supports now grid-based layout and stack-based layout, and a new STACK layout container was introduced to define stack-based layout forms explicitely. Therefore, you can now use the same form definition for all mobile front-ends, by implementing the layout type of your choice. It is even possible to mix grid-based or stack-based forms in the same app.

Loading different forms according to front-end type

If you want to use a grid-based or stack-based form according to the front-end, you can load the form with `OPEN FORM` (or `OPEN WINDOW`) based on the front-end name returned by the `feInfo.feName` front call:

```
-- util module
DEFINE fe_name STRING

FUNCTION get_fe_name()
   IF fe_name IS NULL THEN
      CALL ui.Interface.frontCall("standard", "feInfo", "feName", [fe_name])
   END IF
   RETURN fe_name
END FUNCTION

FUNCTION is_gmi()
   RETURN (get_fe_name() == "GMI")
END FUNCTION

-- main module
MAIN
   ...
   OPEN FORM f1 FROM IIF( is_gmi(), "myform_stack", "myform_grid" )

   ...
END MAIN
```

Desupported database drivers
Database drivers for old database client versions are removed according to the vendor de-support plans.

Database drivers desupported in version 3.00:

- SAP Sybase ASE 15.x (`dbmase_15`): New SAP Sybase ASE 16.x version is now supported.
- Oracle Database 10.1 and 10.2 (`dbmora_10`)

  Note that no more driver is available for Linux PowerPC 32/64 platforms, because Oracle has desupported this platform.
- IBM DB2 UDB 9.x (`dbmdb2_9`)

Web Services changes

There are changes in support of web services in Genero 3.00.

Default SSL protocol

The default for the FGLPROFILE entry `security.global.protocol` is now `SSLv23`, enabling all supported SSL protocols, including `TLSv1.2` as required by the Federal Law of USA. In prior versions, the default was TLSv1 (v1.0). It is up to the web server administrator to restrict the SSL protocol to TLSv1.2.

For more details, see HTTPS and password encryption on page 2510

Server socket read/write timeout

Before version 3.00, when a WS client did not send all the HTTP body (for instance, after connection has been accepted), by default the WS server was waiting indefinitely, and this could end up in a deny of service.

The `com.WebServiceEngine` class supports now a new option called `server_readwritetimeout`, to define the server socket read/write timeout: If a timeout occurs, the WS server program will raise the BDL exception -15553. By default this timeout is defined as 5 seconds.

For more details, see WebServiceEngine options on page 2032.

HTTPPart header default settings with com.HTTPPart.CreateAttachment()

The `com.HTTPPart.CreateAttachment()` method now by default headers fields according to the filename and file extension.

For more details, see com.HTTPPart.CreateAttachment on page 2080.

File path returned by com.HTTPPart.getAttachment()

Before version 3.00, the `com.HTTPPart.getAttachment()` method returned the path to a temporary file. Starting with Version 3.00, this method will now return the absolute path location of the received part filename, according to the "Content-Disposition" header.

For more details, see com.HTTPPart.getAttachment on page 2080.

XForms characters in com.HTTPServiceRequest.readFormEncodedRequest()

Starting with version 3.00, if the result string of the HTTP request contains & or = XForms special characters, these are escaped by doubling it.

For more details, see com.HTTPServiceRequest.readFormEncodedRequest on page 2046.

> **Note:** This behavior change is related to the bug fix FGL-401.

Specific exception -15575 when GAS disconnects web service server

The GWS methods listed below will raise an exception with a specific error code -15575, when the GAS disconnects properly the web service server. Before version 3.00, the generic error -15565 was raised. A specific error code allows you to distinguish properly a normal disconnection from other errors, in a `TRY/CATCH` block. See code examples in method reference pages:

- com.WebServiceEngine.GetHTTPServiceRequest on page 2027
- com.WebServiceEngine.HandleRequest on page 2028

SOAP fault handling in client stub

Web Services client stub generation has been changed to support fault response with HTTP error code of 200.

The generated code supports SOAP fault with HTTP error code of 200 and 500. To enable this new feature in your client stub code, re-generate the stubs with the `fglwsdl` tool.

For more details, see Client side SOAP fault handling.

Optional multipart handling in client stub

In the generated client stub code, all functions handling the SOAP request with multipart get an additional input parameter and/or return parameter as a `DYNAMIC ARRAY OF com.HTTPPart`, to pass and return optional parts.

When generating client stubs managing multipart, you get an extra input and/or output variable called "`AnyInputParts`" and "`AnyOutputParts`" that is a `DYNAMIC ARRAY` of `com.HTTPPart` objects. Those variables may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program by handling those dynamic arrays in any Genero functions calling such stubs.

Request example prior to 3.00:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

Request example 3.00 and greater, with extra input variable `AnyInputParts`:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

> **Note:** This change has also been backported in 2.50.25.

For more details, see Multipart in the client stub on page 2460.

Removal of FGLWSNOINFO environment variable

Before version 3.00, the GWS library was displaying by default a message about certificates used by the program:

```
—
WS-INFO (Certificate authority) | Loading from Windows keystore
—
```

To avoid this message, it was possible to set the FGLWSNOINFO environment variable to TRUE.

Starting with version 3.00, this message is no longer displayed by the GWS library, and the FGLWSNOINFO is no longer required.

Desupported Web Services APIs

The methods listed in the following table are de-supported in Genero 3.00.

Table 83: Table of de-supported methods (with their alternative)

Method de-supported as of 3.00	Alternative method to use
`com.Util.CreateDigestString`	`security.Digest.CreateDigestString` on page 2294
`com.Util.CreateRandomString`	`security.RandomGenerator.CreateRandomString` on page 2279
`com.Util.CreateUUIDString`	`security.RandomGenerator.CreateUUIDString` on page 2280

Oracle DB NUMBER type
The NUMBER/FLOAT Oracle data type can now be extracted by fgldbsch to create .sch files.

Before Genero 3.00, columns using the native Oracle NUMBER/NUMBER(p>32) type (with up to 38 significant digits), or the FLOAT(b) type (when (b/3)>32), were denied by the fgldbsch schema extractor. This restriction was applied to avoid potential overflow errors, if the Oracle NUMBER/FLOAT column contains values that do not fit into a BDL DECIMAL(32,s) type.

Starting with Genero 3.00, fgldbsch can map NUMBER/FLOAT native Oracle types to BDL DECIMAL(32) or DECIMAL(32,s) types, according to the `-cv` option:

- NUMBER (floating point number) is extracted as DECIMAL(32)
- NUMBER(p>32) (scale defaults to 0) is extracted as DECIMAL(32,0)
- NUMBER(p>32,s) or NUMBER(*,s) is extracted as DECIMAL(32,s)
- FLOAT(b) is extracted as DECIMAL(b/3) or FLOAT

For more details about Oracle type conversion rules and `-cv` type positions, run fgldbsch with the `-cx ora` option.

> **Note:** This new behavior has been introduced to simplify integration with existing Oracle databases, to extract .sch schema from databases using column types that have no exact equivalent BDL type. When designing new database tables, you should only use DECIMAL(p,s), with p<=32 to achieve maximum portability. When fetching numeric values with more than 32 significant digits into BDL decimals, values will be rounded for DECIMAL(32), or raise an overflow error -1226 for DECIMAL(32,s).

Oracle DB scroll cursor emulation removal
The scroll cursor emulation has been removed in the Oracle DB driver.

Before Genero 3.00, it was possible to enable scrollable cursor emulation (with temporary files) by defining the following FGLPROFILE entry:

```
dbi.database.mydbname.ora.cursor.scroll.emul = true
```

This feature was supported to workaround an Oracle DB bug in versions 8 and 9i. The Oracle bug does no longer exist in recent Oracle DB versions and the default native scrollable cursor feature can be safely used.

If this FGLPROFILE entry is set, the runtime system will print a warning to stderr.

MySQL VARCHAR size limit
MySQL 5 VARCHAR columns can be used to store VARCHAR(N>255) values.

Before Genero 3.00, the MySQL driver converted a VARCHAR(N>255) type to a MySQL TEXT type, because MySQL versions before 5.0.3 only allowed up to 255 characters for a VARCHAR column. MySQL TEXT type is a large object type with specific semantics and constraints, but it was the only available type to store character data above the 255 character limit. As a result, data type information was lost when extracting the database schema with fgldbsch from a MySQL database: When creating a table in a Genero BDL program, the original VARCHAR(N>255) type was converted to TEXT (with a fixed size of 65535

characters), and then converted by fgldbsch back to a VARCHAR2(65535) type in the .sch file. The original size of the VARCHAR type was lost.

Starting with Genero 3.00, when creating a table in a BDL program with CREATE TABLE, the MySQL driver leaves any VARCHAR(N) as-is, even if the size is greater as 255.

> **Note:** The MySQL driver does not distinguish MySQL server 5.0.x (5.0.2 / 5.0.3) versions. It assumes that we are connected to a server version 5.0.3 or above, supporting large VARCHAR types.

If your application is using VARCHAR(N) types with N>255 and your MySQL server version is 5.0.3 or above, you should review your database creation scripts to use VARCHAR(N) instead of TEXT.

> **Note:** The CHAR(N>255) types are still mapped to a MySQL TEXT type, because MySQL CHAR type has a limit of 255 characters. When designing a database, consider using CHAR only for short character string data storage (less than 50 characters), and use VARCHAR for larger character string data storage (name, address, comments).

MySQL DATETIME fractional seconds

MySQL 5.6.4 TIME and DATETIME types support fractions of seconds that can be used to store DATETIME HOUR TO FRACTION(N) or DATETIME YEAR TO FRACTION(N).

Before Genero 3.00, the MySQL driver converted DATETIME types as follows:

- DATETIME HOUR TO SECOND was converted to MySQL TIME.
- Other DATETIME types were converted to MySQL DATETIME.

Starting with Genero 3.00, when creating a table in a BDL program with the CREATE TABLE statement, if the MySQL server version is greater or equal to 5.6.4, the types are converted differently, as follows:

The SQL Translator of the MySQL driver makes the following conversions automatically for the DATETIME types:

- DATETIME HOUR TO MINUTE is converted to MySQL TIME (seconds set to 00).
- DATETIME HOUR TO SECOND is converted to MySQL TIME.
- DATETIME HOUR TO FRACTION(N) is converted to MySQL TIME(N).
- DATETIME YEAR TO MINUTE is converted to MySQL DATETIME (seconds set to 00).
- DATETIME YEAR TO SECOND is converted to MySQL DATETIME.
- DATETIME YEAR TO FRACTION(N) is converted to MySQL DATETIME(N).

This change has no impact if your application is using DATETIME HOUR TO SECOND or DATETIME YEAR TO SECOND. However, it is now possible to store DATETIME HOUR TO FRACTION(N) and DATETIME YEAR TO FRACTION(N) data. The DATETIME YEAR TO FRACTION(N) is typically used to implement data modification timestamps to track user changes.

PostgreSQL DATETIME type mapping change

Conversion of DATETIME type with fractional seconds to PostgreSQL TIME(N)/TIMESTAMP(N) was invalid and has been reviewed.

Before Genero 3.00, the PostgreSQL driver converted DATETIME types as follows:

- DATETIME HOUR TO MINUTE was converted to TIMESTAMP(3) WITHOUT TIME ZONE
- DATETIME HOUR TO SECOND was converted to TIME(0) WITHOUT TIME ZONE
- DATETIME HOUR TO FRACTION(N) was converted to TIME(N+1) WITHOUT TIME ZONE
- DATETIME YEAR TO MINUTE was converted to TIMESTAMP(3) WITHOUT TIME ZONE
- DATETIME YEAR TO SECOND was converted to TIMESTAMP(3) WITHOUT TIME ZONE
- DATETIME YEAR TO FRACTION(N) was converted to TIMESTAMP(N+1) WITHOUT TIME ZONE

Starting with Genero 3.00, when creating a table in a BDL program with CREATE TABLE, the types are converted in a different way.

The SQL Translator of the PostgreSQL driver makes the following conversions automatically for the DATETIME types:

- DATETIME HOUR TO MINUTE is converted to PostgreSQL TIME(0) WITHOUT TIME ZONE (seconds set to 00).
- DATETIME HOUR TO SECOND is converted to PostgreSQL TIME(0) WITHOUT TIME ZONE.
- DATETIME HOUR TO FRACTION(N) is converted to PostgreSQL TIME(N) WITHOUT TIME ZONE.
- DATETIME YEAR TO MINUTE is converted to PostgreSQL TIMSTAMP(0) WITHOUT TIME ZONE (seconds set to 00).
- DATETIME YEAR TO SECOND is converted to PostgreSQL TIMESTAMP(0) WITHOUT TIME ZONE.
- DATETIME YEAR TO FRACTION(N) is converted to PostgreSQL TIMESTAMP(N) WITHOUT TIME ZONE.

> **Note:** This behavior change is related to the bug fix FGL-3893.

This bug fix introduces a incompatibility and can have an impact on applications using DATETIME HOUR TO MINUTE, DATETIME HOUR TO FRACTION(N) or DATETIME YEAR TO FRACTION(N). If you are using one of these types, consider reviewing your database schema, to modify the column types accord to the new SQL type conversion rules.

MariaDB support
The MariaDB database is now supported by Genero 3.00.

MariaDB is the open source brand of Oracle's MySQL and has been adopted by several major organizations.

The purpose of the MariaDB project is to be a drop-in replacement for MySQL.

MariaDB supported versions are 10.0 and higher.

To connect to MariaDB, use the MySQL database driver (dbmmys), and follow MySQL adaptation guide for configuration and SQL portability issues.

According to the libmysqlclient library compatibility, you might need to configure Genero to use a version-stamped driver. As of Genero version 3.00, the generic driver name "dbmmys" can be used to connect to MariaDB 10.0. See Database driver specification (driver) on page 462 for more details.

FreeTDS driver supports SQL Server 2008, 2012, 2014
The FreeTDS driver can now be used for SQL Server versions > 2005.

Before Genero version 3.00, the FreeTDS driver could only be used to connect to SQL Server 2005. Starting with Genero 3.00 the `dbmftm` driver can connect to SQL Server 2008, 2012 and 2014.

With SQL Server version >= 2008, date/time types used to store DATE and DATETIME values is different as with SQL Server version 2005. See DATE and DATETIME data types on page 604 for more details.

> **Important:** For SQL Server version 2008, 2012 and 2014, you must set `TDS_Version=7.3` in odbc.ini. Using TDS version 8.0 introduces problems (tested with FreeTDS 0.95.5 to 0.95.19)

FGL_GETVERSION() built-in function
The `FGL_GETVERSION()` function now returns the product version number (for ex: 3.00.00).

Prior to Genero 3.00, the `FGL_GETVERSION()` built-in function was returning the internal build number.

Starting with Genero 3.00, the function returns the product version number as a string, such as 3.00.00.

Built-in front-end icons desupport
Images resources included in front-ends are desupported with Genero 3.00.

Starting with Genero 3.00, the icon files distributed in front-end packages are no longer provided (former `GDC-installation-dir/pics` for example)

Common icons for buttons, toolbars, topmenus, and other items using icons can be centralized on the application side where the program executes. This feature should be used to get the same icons on different type of front-ends, or use specific icons, but from the same central icon directory. For more details, see Providing the image resource on page 784.

Note however that mobile front-ends will display default icons, for default action views, if no `IMAGE` attribute is specified for the action. See Rendering default action views on mobile on page 1279 for more details.

Presentation style attribute changes
Deprecated and renamed presentation style attributes.

Starting with version 3.00:

The following presentation style attributes are desupported:

- `CheckBox: customWidget`

The following presentation style attributes are deprecated:

- `Image: imageContainerType ( = "browser")`

  Note: Replace `IMAGE` fields using this style attribute by URL-based `WEBCOMPONENT` fields.

Modifications in front calls
Describes changes done in front calls.

Front call modifications in BDL version 3.00:

- Before version 3.00, the `connectivity` front call was accepting a hostname as parameter. Starting with version 3.00, this front call no longer use a hostname: It will only check the available network type. For more details, see the mobile.connectivity front call.

SERIAL emulation with SQL Server
The SERIAL and BIGSERIAL types can be emulated with triggers and sequences when using SQL Server 2012 and higher.

By default when using SQL Server, the SERIAL and BIGSERIAL types are emulated with IDENTITY columns. This native sequence generator is the fastest and prefered solution. However, it requires to remove the serial column in all INSERT statements, which can lead to a large change in your legacy code.

Until version 3.00, you could workaround this limitation by using the "regtable" serial emulation. But this solution is using a dedicated SERIALREG table that must be updated for each INSERT statement. This can result in bad performances, when concurrent programs create rows in the same tables (locking issues in SERIALREG).

Starting with Genero 3.00, it is now possible to use a serial emulation based on triggers and sequences. Sequences where introduced in SQL Server version 2012, so you need at least a 2012 server in order to use this emulation:

```
dbi.database.mydb.ifxemul.datatype.serial.emulation = "trigseq"
```

Improved compilation time
The `fglcomp` and `fglform` compilers have been reviewed to achieve faster compilation.

A Genero project can be very large, with thousands of `.4gl` source files to compile. Compilation time can be an issue when the whole set of sources needs to be compiled every day, or several times a day.

In Genero 3.00, the `fglcomp` compiler has been improved to deliver better performances. Depending on the content of the source file, the compiler can be over twice as fast.

Loading `.sch` database schema files has also been improved. Using huge schema files with several thousands of lines is no longer an issue. This is especially useful when compiling forms that define fields based on database columns in a schema file.

Preprocessor changes
Several bugs have been fixed in the preprocessor, that can now result in a compilation error.

String token expansion

Before version 3.00, the following preprocessor syntax could be used to expand a string macro parameter:

```
&define T(x) DISPLAY "head_"#x"_tail"
-- macro usage:
T(body)
```

This was producing following result (after preprocessing):

```
"head_""body""_tail"
```

And was accepted by the compiler, because it was interpreted as a single string literal.

The new preprocessor now produces (as expected):

```
"head_" "body" "_tail"
```

However, this will now result in a compiler error, because this is not a valid string literal.

To solve such issue and get the same result string as before version 3.00, use the `||` concatenation operator in the preprocessor macro and add (escaped) double quotes before and after the *`#ident`* placeholder:

```
&define T(x) DISPLAY "head_""" || #x || """_tail"
```

or, by using single quotes as border strings delimiters:

```
&define T(x) DISPLAY 'head_"' || #x || '"_tail'
```

Identifier concatenation

Before version 3.00, the following type of macro:

```
&define FOO() foo
-- macro usage:
FOO()bar
```

was producing a single identifier token (accepted by the compiler):

```
foobar
```

But it will now produce two distinct identifier tokens (as expected):

```
foo bar
```

And this will result in a compilation error.

Backslash in macro parameters

Before version 3.00.00 is was possible to use the backslash to escape a comma in preprocessor macro parameters. This syntax is no longer allowed by the preprocessor, it is not a valid usage. To solve such issue, replace parameters by real string literals in the macro:

```
-- bad coding
&define FOO(p1) DISPLAY #p1
FOO(hello world)       -- expands to: DISPLAY "hello world"
FOO(hello \, world)    -- error

-- good coding
&define FOO(p1) DISPLAY p1
FOO("hello world")     -- expands to: DISPLAY "hello world"
FOO("hello , world")   -- expands to: DISPLAY "hello , world"
```

The ## paste operator

Before version 3.00.00, the ## paste operator could be used to construct code with two elements that did not result in a valid token, for example:

```
&define FOO(name) rec_ ## [ x ]
FOO(x)
```

was producing:

```
rec_[ x ]
```

This kind of preprocessor macro is no longer allowed in version 3.00.00 and will result in a compiler error:

```
x.4gl:2:1:2:1:error:(-8042) The operator '##' formed 'rec_[', an invalid
 preprocessing token.
```

The ## paste operator must be used to join two identifiers, to create a new identifier:

```
&define REC_PREFIX(name) rec_ ## name
LET REC_PREFIX(customer) = NULL
```

will produce:

```
LET rec_customer = NULL
```

Current system time in UTC

Use the util.Datetime.getCurrentAsUTC() method to get the current system date/time in UTC.

Starting with Genero version 3.00, you can use the util.Datetime.getCurrentAsUTC() method to get the current system time in UTC (Coordinated Universal Time).

This method has been added to solve the issue when using `util.Datetime.toUTC(CURRENT)` during the daylight saving time transition period in the fall, as described in util.Datetime.toUTC on page 1953.

Structured ARRAYs in list dialogs

ARRAYs with sub-records can be used in list dialogs, to simplify array definition based on database tables, requiring additional information at runtime.

Starting with Genero version 3.00, ARRAY variables defined with a sub-records can be bound to `DISPLAY ARRAY` and `INPUT ARRAY` screen records.

This is especially useful when you need to define arrays from database tables, and handle additional row information at runtime, for example, to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute.

An array is usually defined with a flat list of members:

```
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD LIKE items.*
...
```

With version 3.00, arrays structured with sub-records can now be used within a `DISPLAY ARRAY` or `INPUT ARRAY` dialog. The array members and the form fields used by the screen array are bound by position:

```
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
                    item_data RECORD LIKE items.*,
                    it_image STRING,
                    it_count INTEGER
                END RECORD
...
DISPLAY ARRAY a_items TO sr.*
   ...
```

For more details about program variable to form field binding in dialogs, see Variable binding in DISPLAY ARRAY on page 1078, Variable binding in INPUT ARRAY on page 1101, Binding variables to form fields on page 1264, Example 4: DISPLAY ARRAY with structured array.

2.51 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.51.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Check prior upgrade guides if you migrate from an earlier version.

1. Desupported database drivers on page 101
2. New database driver name specification on page 102
3. The FIELD form item type and .val schema file on page 102
4. TRY/CATCH and ERROR LOG on page 103
5. DATETIME types with SQLite on page 103
6. Desupport of C-Extension API functions on page 103

Desupported database drivers

Database drivers for old database client versions are removed according to the vendor de-support plans.

Database drivers desupported in version 2.51:

- Genero DB is no longer supported (`dbmads*`).
- Oracle MySQL 5.0, 5.1 (`dbmmys50x`, `dbmmys51x`).
- PostgreSQL 8.3, 8.4 (`dbmora83x`, `dbmpgs84x`)
- Oracle MySQL 4.1 (`dbmmys41x`)
- Oracle Database 9.2 (`dbmora92x`)
- IBM DB2 UDB version 7.x and 8.x (`dbmdb27x`, `dbmdb28x`)

Note also that database driver naming convention has changed in 2.51, for more details see New database driver name specification on page 102.

New database driver name specification
Allows database driver specification without target database version information.

Starting with version 2.51, the database drivers are following a new file name convention, to let you specify a generic name according to the target database type, without any database version information.

> **Important:** Most database driver names have changed. You need to re-configure the `"driver"` entry in your FGLPROFILE settings (or database connection string parameters), to match the new driver names. If you are using the default driver (`dbmdefault`), there is no configuration change needed. To simplify upgrading, the runtime system identifies old driver names and converts them to new names. However, you should consider using the generic driver name corresponding to the type of database your applications connect to. The error -6366 occurs if the runtime system is not able to load the specified database driver, or cannot identify an old driver name.

Before version 2.51, it was required to specify the exact database type and version, that had to match both the database client and the server version. For example, when using Oracle 11.2 (server and client):

```
dbi.database.stores.driver = "dbmoraB2x"
```

Starting with 2.51, you can now for example specify a generic driver name without version, which can connect to any database server version supported by the DB vendor client/server protocol. The generic name defines a database driver for the latest database client version that is available on the platform:

```
dbi.database.stores.driver = "dbmora"
```

Each generic database driver name has also a human-readable alias, such as "informix" or "oracle".

```
dbi.database.stores.driver = "oracle"
```

To simplify driver specification, install the latest database client software that corresponds to the generic driver name, especially if it does not require a database server upgrade.

For some type of database client software, additional database drivers are still provided for older database client versions (if available on the platform). In such case, the driver file name gets a version identifier.

For example:

- `dbmora_11` (Oracle 11g client)
- `dbmmys_5_1` (Oracle MySQL 5.1.x)
- `dbmsnc_10` (SQL Native Client 10 (SQLNCLI10.DLL))
- `dbmsnc_9` (SQL Native Client 9 (SQLNCLI.DLL))

Such database drivers with version info are provided to follow db client library dependency rules, as defined by the database vendors. For example, on a Linux platform, Oracle MySQL version 5.1.x provides the db client library named `libmysqlclient.so.16`. In this file name, "16" is the version number that defines the shared library compatibility. The database driver that was compiled and linked in a compatible db client environment is `dbmmys_5_1`: This database driver is linked to `libmysqlclient.so.16`. Starting with Oracle MySQL version 5.5.x, the db client library version number has incremented to 18 (i.e. `libmysqlclient.so.18`). The driver to be used with that library version is `dbmmys_5_5`, which was compiled and linked with a 5.5.x environment.

The FIELD form item type and .val schema file
Form files using the FIELD item type and/or .val attribute definitions must be reviewed.

Starting with version 2.51, the `FIELD` item type defining abstract fields in forms, based on .val schema file attributes is deprecated.

Further, any non-I4GL attribute defined in the .val schema file must be avoided: Reading attributes in the .val is now only supported for compatibility with I4GL projects.

With Genero, it is recommended to define all form item attributes in the form definition file.

TRY/CATCH and ERROR LOG
Errors are no longer logged when raised in a TRY/CATCH block.

Before version 2.51, exceptions occuring in a TRY/CATCH block were logged if the error log is initiated with the startlog() function. With version 2.51, if an exception is raised in a TRY/CATCH block, it will no longer be logged in the error log file. In other words, the TRY/CATCH block will behave like WHENEVER ERROR CONTINUE, regarding error logging.

> **Note:** This behavior change is related to the bug fix FGL-3091.

Example:

```
CALL startlog("errors.txt")
...
TRY
    INSERT INTO customer ...
CATCH
    -- Handle errors and write to error log with errorlog() if needed.
    IF SQLCA.SQLCODE == -8634 THEN
       ...
    END IF
END TRY
```

> **Important:** In order to get this new behavior, the pcode is no longer compatible with older versions (<=2.50): All programs must be recompiled.

DATETIME types with SQLite
Better support for Informix DATETIME types emulation within SQLite.

Before version 2.51, DATETIME SQL types where converted to SQLite types as follows:

- DATETIME HOUR TO SECOND type was translated to TIME (hh:mm:ss).
- DATETIME YEAR TO FRACTION and all other combinations (except HOUR TO SECOND) were translated to TIMESTAMP (YYYY-MM-DD hh:mm:ss.fff).

Since most DATETIME types were converted to TIMESTAMP, it was not possible to distinguish common date/time types such as DATETIME HOUR TO MINUTE or DATETIME YEAR TO MINUTE, especially when extracting the database schema with fgldbsch: Type information was lost and this prevented schema-base variable definitions with DEFINE LIKE.

Starting with version 2.51, common DATETIME SQL types are now mapped to different types in SQLite, for a better support of these types. In fact, SQLite allows to define table columns with custom types (you can use any type name), however the number of tokens in the syntax is limited so it's not possible to use for ex the tokens DATETIME YEAR TO SECOND directly. The Genero database driver uses this SQLite SQL language feature to map Informix-style DATETIME types to specific custom types. For example, a DATETIME HOUR TO MINUTE becomes a SMALLTIME, a DATETIME YEAR TO FRACTION(2) becomes a DATETIME(2), etc. Further, the data values inserted in the database do now match exactly the precision of the original DATETIME type. For more details about date/time mapping and emulation, see DATE and DATETIME data types on page 714.

Desupport of C-Extension API functions
BIGINT and BOOLEAN stack functions and C API functions for C-Extensions are no longer supported.

Since version 2.51:

The C-Extension stack functions to handle BIGINT and BOOLEAN types have been removed:

Table 84: Desupported FGL stack functions

popboolean()
popbigint()

pushboolean()
pushbigint()

The C API functions such as `decadd()`, `risnull()`, `rsetnull()`, have been removed. These functions are part of the IBM® Informix® ESQL/C product and cannot be part of the Genero BDL product. The Genero runtime system provides only the C functions to push and pop data on the Genero BDL stack.

Below is the list of C API functions that have been removed, check your C extension code for the usage of these functions. If such functions are required, link your C-Extensions with the IBM® Informix® ESQL/C libraries.

Table 85: Desupported C API functions

bycmpr()
byleng()
bycopy()
byfill()
risnull()
rsetnull()
rgetmsg()
rgetlmsg()
rtypalign()
rtypmsize()
rtypname()
rtypwidth()
rdatestr()
rdayofweek()
rdefmtdate()
ifx_defmtdate()
rfmtdate()
rjulmdy()
rleapyear()
rmdyjul()
rstrdate()
ifx_strdate()
rtoday()
ldchar()
rdownshift()
rfmtdouble()
rfmtint4()
rstod()

rstoi()
rstol()
rupshift()
stcat()
stchar()
stcmpr()
stcopy()
stleng()
decadd()
deccmp()
deccopy()
deccvasc()
deccvdbl()
deccvflt()
deccvint()
deccvlong()
decdiv()
dececvt()
decfcvt()
decmul()
decround()
decsub()
dectoasc()
dectodbl()
dectoflt()
dectoint()
dectolong()
dectrunc()
rfmtdec()
dtaddinv dtaddinv()
dtcurrent()
dtcvasc()
ifx_dtcvasc()
dtcvfmtasc()
ifx_dtcvfmtasc()
dtextend()
dtsub()

dtsubinv()
dttoasc()
dttofmtasc()
ifx_dttofmtasc()
incvasc()
incvfmtasc()
intoasc()
intofmtasc()
invdivdbl()
invdivinv()
invextend()
invmuldbl()

2.50 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.50.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Check prior upgrade guides if you migrate from an earlier version.

1. Desupported database drivers on page 106
2. TEXT/BYTE support with FTM/ESM database drivers on page 106
3. Presentation style attribute changes on page 107
4. Floating point to string conversion on page 107
5. Web Services changes on page 108
6. Implicit creation of certificates for HTTPS on page 108
7. PostgreSQL schema extraction needs namespace on page 108
8. Client stubs managing multipart changes on page 109

Desupported database drivers

Database drivers for old database client versions are removed according to the vendor de-support plans.

Database drivers desupported in versions 2.50:

- SQL Server MDAC drivers (Code MSV, name: `dbmmsv*`):

  On a Microsoft™ Windows™ platform, use the SQL Native Client driver instead (Code SNC).

  With the SNC drivers, set the **dbi.database.*dbname*.snc.widechar** FGLPROFILE entry to **false** when using CHAR/VARCHAR/TEXT in the SQL Server database.
- Oracle MySQL 5.4 (`dbmmys54x`)
- Oracle Database 8.1 (`dbmora81x`)
- Oracle Database 9.0 (`dbmora90x`)

TEXT/BYTE support with FTM/ESM database drivers

FTM and ESM database drivers TEXT/BYTE type mapping has changed.

Since version 2.50, the TEXT and BYTE data types are now converted respectively to VARCHAR(MAX) and VARBINARY(MAX) data types, the recommended LOB types introduced in SQL Server 2005. Before version 2.50, the TEXT and BYTE data types were converted to TEXT and IMAGE data types, respectively, in SQL Server.

Note: This behavior change is related to the bug fix FGL-2534.

It is still possible to use SQL Server TEXT and IMAGE types, but if you create or alter tables in an FGL program, the VARCHAR(MAX) and VARBINARY(MAX) types will be used instead.

Presentation style attribute changes
Deprecated and renamed presentation style attributes.

Starting with version 2.50:

The following presentation style attributes are deprecated (still implemented, but not to be used):

- `Window: backgroundImage`
- `TextEdit: textSyntaxHighlight`

The next presentation style attributes have been replaced by a new style attribute, or have been renamed:

- `CheckBox: nativeLook => customWidget` (with same possible values)

Important: In 3.00, the `customWidget` attribute is desupported.

Floating point to string conversion
The default formatting of a `DECIMAL(P)`, `SMALLFLOAT` and `FLOAT` adapts to the significant digits of the value.

Floating point decimal types (like `DECIMAL(5)`) can store a large range of values, with a variable number of digits after the decimal point: For example, a `DECIMAL(5)` can store 12345 as well as 0.12345. See DECIMAL(p,s) on page 200 for more details about floating point decimal types.

With Genero 2.50, the conversion to string from a `DECIMAL(P)`, `FLOAT` and `SMALLFLOAT` has been revised, to keep all significant digits and avoid data loss.

Note: This behavior change is related to the bug fix FGL-3915.

Before Genero 2.50, floating point decimals converted to strings were formatted with 2 decimal digits by default, which could lead to data loss. See following example using a `DECIMAL(12)`:

```
MAIN
   DEFINE str STRING, dec12, dec12_bis DECIMAL(12)
   LET dec12 = 10.12999
   LET str = dec12
   DISPLAY str
   LET dec12_bis = str
   DISPLAY (dec12 == dec12_bis)
END MAIN
```

Prior to Genero 2.50, the above code would display:

```
10.13
     0
```

Starting with Genero 2.50, all significant digits are kept, which allows proper decimal data serialization:

```
10.12999
     1
```

Prior to Genero 2.50, floating point decimal values conversion of huge values could also lose digits in the whole part of the number; the width of the result was never longer than p + 2. Starting with Genero 2.50, all significant digits of a floating point decimal are kept in the result string:

```
   Values            Vers<2.50       Vers>=2.50
-----------------------------------------------
```

```
    1.23456e123       1.23456e123     1.23456e123
    1.23456e40        1.235e40        1.23456e40
    123.456           123.46          123.456
    123456.0          123456.0        123456.0
    0.123456          0.12            0.123456
    0.0123456         0.01            0.0123456
    0.00123456        0.00            0.00123456
    1.23456e-08       0.00            1.23456e-08
```

If you expect that any `DECIMAL(P)` to string conversion rounds to 2 digits, define the following FGLPROFILE entry:

```
fglrun.decToCharScale2 = true
```

> **Note:** Do not use this configuration parameter unless you have migration issues. This configuration parameter applies only to `DECIMAL(P)` types, `FLOAT` and `SMALLFLOAT` conversions to string is not impacted.

Web Services changes
Several methods of built-in and extension classes are de-supported.

The methods listed in the following table are deprecated in version 2.50.

Table 86: Table of deprecated methods (with their alternative)

Method deprecated as of 2.50	Alternative method to use
`com.Util.CreateDigestString`	`security.Digest.CreateDigestString` on page 2294
`com.Util.CreateRandomString`	`security.RandomGenerator.CreateRandomString` on page 2279
`com.Util.CreateUUIDString`	`security.RandomGenerator.CreateUUIDString` on page 2280

Implicit creation of certificates for HTTPS
Certificates for HTTPS are now created implicitly, when nothing is specified in FGLPROFILE.

Before version 2.50, certificates for HTTPS had to be specified explicitly in FGLPROFILE.

Starting with 2.50, in no HTTPS certificate is defined in FGLPROFILE, when a web services program starts, the creation is implicit.

PostgreSQL schema extraction needs namespace
To extract a database schema from PostgreSQL, the fgldbsch tool now requires db namespace specification.

In version 2.50, the fgldbsch database schema extractor can only extract the schema from a PostgreSQL database if you specify the `-ow` option.

> **Note:** This behavior change is related to the bug fix FGL-2647.

PostgreSQL distinguishes table owners from table schemas (i.e. table namespaces). The real table namespace is defined by the `pg_class.relnamespace` column: it contains the `oid` of a namespace defined in `pg_namespace`. For PostgreSQL, the fgldbsch `-ow` option will specify the namespace, instead of the owner of the table, because an db user can create several schemas/namespaces and use the same table name in those different namespaces. As result, filtering on user name can mix table definitions from different schemas/namespaces.

When extracting a database schema from a PostgreSQL database, you must specify the namespace of tables with the `-ow` option. If no `-ow` option is specified and the `-un` option is specified, fgldbsch will use

the login name of the `-un` option as namespace. If neither `-ow`, nor `-up` options are specified, fgldbsch will use the PostgreSQL "public" namespace/schema by default.

Since database tables are usually created in the "public" namespace, you typically specify this namespace with the `-ow` option:

```
fgldbsch -db test1 -dv dbmpgs -un pgsuser -up fourjs -v -ow public
```

Client stubs managing multipart changes
You must update client programs that call client stubs managing multipart.

> **Important:** This change has been backported from V 3.00

Starting with version 2.50.25, when generating client stubs managing multipart, you get an extra input and/or output variable called "`AnyInputParts`" and "`AnyOutputParts`" that is a `DYNAMIC ARRAY` of `com.HTTPPart` objects. Those variables may contain additional input and/or output HTTP parts not specified in the WSDL. You will have to adapt your client program by handling those dynamic arrays in any Genero functions calling such stubs.

Request example prior to 2.50.25:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

Request example 2.50.25 and greater, with extra input variable `AnyInputParts`:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus
END FUNCTION
```

2.40 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.40.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Desupported database drivers on page 110
2. Program size option removal (fglrun -s) on page 110
3. Informix SERIAL emulation with SQL Server on page 110
4. SIZEPOLICY attribute removal for containers on page 110
5. The LVARCHAR type in IBM Informix databases on page 110
6. Right-trim collation for character types in SQLite on page 111
7. Message files support now 4-bytes integer message numbers on page 111
8. MySQL client library version change in MySQL 5.5.11 on page 111
9. New compiler warning to avoid action shadowing on page 111
10. Runtime error raised when report dimensions are invalid on page 112
11. Linker checks all referenced functions on page 112

Desupported database drivers
Database drivers for old database client versions are removed according to the vendor de-support plans.

Database drivers desupported in versions 2.40:

- PostgreSQL versions 8.0, 8.1 and 8.2 (dbmpgs80x, dbmpgs81x, dbmpgs82x).
- Sybase Adaptive Server Anywhere (ASA) version 8 driver (dbmasa8x)

  The dbmasa8x driver was provided for the Sybase ASA desktop database engine (renamed Sybase SQL Anywhere when writing these lines). Since version 2.30, Genero Business Development Language supports a new driver to connect to Sybase Adaptive Server Enterprise (ASE): dbmase*.

Program size option removal (fglrun -s)
The -s option of fglrun is no longer available.

Before version 2.30 the -s option of fglrun could be used to compute the size of program elements such as global and module variables, p-code and structured data types. Starting with version 2.30, this option reported a size of zero. With version 2.40 the -s option is now fully desupported.

The -s option was mainly implemented for internal use. Regarding the amount of memory used by a program, you should consider the memory allocated dynamically at runtime: If you fill large dynamic arrays, or leave a lot of SQL cursors open without freeing them, the memory footprint of a program can be much larger as the actual size of static elements that could be reported by the -s option.

Informix® SERIAL emulation with SQL Server
SERIAL type emulation has been enhanced for SQL Server.

Using SCOPE_IDENTITY() to get the last sequence

Before version 2.40, the SQL Server drivers (SNC, MSV, ESM, FTM) were using the @@IDENTITY expression to retrieve the last generated identity column, if the native serial emulation is configured. But @@IDENTITY is not recommended, because it can return an identity value generated for another table in a trigger of the main table.

Starting with 2.40, the SQL Server drivers use the SCOPE_IDENTITY() function, which returns the last number generated in the current scope (ignoring identity numbers generated in triggers).

Regtable serial emulation trigger code change

When using the "regtable" serial emulation, the code of the triggers has changed in version 2.40, using now the SET NOCOUNT ON instruction. Existing serial triggers created by prior versions must be reviewed, to have the same trigger body in all tables, otherwise an SQL error is raised when executing INSERT statements.

SIZEPOLICY attribute removal for containers
The SIZEPOLICY attribute is no longer availabel for layout containers like TABLE / GRID.

Before version 2.40 it was possible to specify a SIZEPOLICY attribute for several sort of form elements, including containers such as TABLE, GRID. The SIZEPOLICY attribute make no sense in containers and is only meaningful for leaf nodes (i.e. widgets such as EDIT, COMBOBOX). The form compiler will now report a syntax error if the SIZEPOLICY attribute is used in the definition of elements that are not widgets.

The LVARCHAR type in IBM® Informix® databases
Native LVARCHAR type of Informix is now mapped by default to a large VARCHAR in schema file.

Starting with version 2.40, the fgldbsch database schema extractor converts now by default IBM® Informix® LVARCHAR(N) types to VARCHAR2(N) with type code 201. Before 2.40, you had to pass -cv AAAB... option to avoid a conversion error when generating the schema file.

The static SQL syntax has been enhanced, to support the LVARCHAR type name in DDL statements such as CREATE TABLE. The non-Informix ODI drivers have been adapted to convert LVARCHAR type names to VARCHAR.

Two-Pass reports can now use VARCHAR types with a size greater as 255 bytes (the temporary table will be created with an LVARCHAR column). Note however that and index is created as well, and IBM® Informix® IDS (version 11 when writing these lines) has a size limitation for indexes. You may get an SQL error -517 if the VARCHAR variable used to group / order rows in the report routine exceeds ~350 bytes (see IDS SQL error -517 for details).

Right-trim collation for character types in SQLite
CHAR and VARCHAR columns in SQLite need to be defined with a TRIM collation to ignore trailing spaces in comparisons.

Since version 2.40, the SQLite database driver adds the COLLATE RTRIM keywords after the CHAR(N) and VARCHAR(N) types in CREATE TABLE statements, when Informix® emulation is enabled (the default). This collation clause forces SQLite to use right-trim comparison rules instead of the default binary mode. The binary mode requires to have the same number of trailing spaces in both character values to be equal. By using COLLATE RTRIM clause, the trailing blanks are trimmed and thus ignored. You should also use [VAR]CHAR(N) COLLATE RTRIM in database creation scripts.

Message files support now 4-bytes integer message numbers
2-byte .msg message number limitation was removed.

Before version 2.40, message files entries could only be defined with numbers in the range -32767 to 32767 (i.e. SMALLINT). This limitation is not longer true in 2.40: Message numbers can now be in the range -2147483648 to 2147483647 (i.e. INTEGER).

> **Note:** This behavior change is related to the bug fix FGL-1670.

MySQL client library version change in MySQL 5.5.11
Shared library version number of the MySQL client library must match the library used to link the ODI driver.

Starting with MySQL 5.5.11, the client library version number was changed from 16 to 18. In fact the `libmysqlclient.so.16` file was renamed to `libmysqlclient.so.18`. From a cross-5.5.x compatibility point of view, this was maybe not the best thing to do: Since the major shared library version has changed, client applications using the C API (such as Genero ODI MySQL drivers) need to be recompiled and re-linked in order to use le latest library.

In Genero version 2.40, the dbmmys55x ODI driver is linked with libmysqlclient.so.18 on the platforms where MySQL 5.5.11+ is available. That is: Linux™, Solaris and Mac OS X® platforms, when writing these lines. On other UNIX™ platforms such as HP, the client library is still `libmysqlclient.so.16`. This may change in the future Genero versions, following the availability of MySQL 5.5.11+ versions.

Therefore, you must pay attention to the MySQL 5.5 version you have installed: You need to upgrade your MySQL 5.5 client software to match le client library that was used to build the `dbmmys55x.so` shared library. On Linux™, you can run the ldd command to check what `libmysqlclient.so` version is required. If it's not possible to upgrade your MySQL client software, please contact the support channel.

New compiler warning to avoid action shadowing
Prevent the same action name at different levels of ON ACTION handlers in a dialog.

The fglcomp compiler of version 2.40 will now print warning -8409, if a dialog block defines ON ACTION handlers at different levels with the same action name.

It is not good practice to use the same action names at different levels of a dialog. For example, you can define several ON ACTION INFIELD handlers using the action name "zoom", but you should not define and ON ACTION zoom at the sub-dialog or dialog level.

If the warning occurs during compilation, modify your code in order to use specific action names at each level, and do not forget to rename the actions of the corresponding action views in the forms.

Runtime error raised when report dimensions are invalid
Report page length checking error -4375 might occur at compile time or runtime.

Starting with version 2.40, a START REPORT instruction raises the runtime error -4375, when the top/bottom margin sizes do not fit the page length.

In version 2.40, the error is not returned at compile time, because report dimensions can be specified with variables in START REPORT.

But since version 2.50.00 (build 2155) fixing bug FGL-3711, the compiler will also raise error -4375 when using constants.

Further change is done in version 2.51.07 (build 2506), by fixing the bug FGL-651, to relax the page length test and allow FIRST PAGE HEADER blocks with the same number of rows as the page length.

> **Note:** This behavior change is related to the bug fix FGL-3035, FGL-3711 and FGL-651.

Linker checks all referenced functions
The linker checks definition of all functions referenced in all modules provided in the link command.

Starting with version 2.40, any reference to a function has to be resolved by the linker: When linking a 42r program, if an unused module references an undefined function, the linker (`fglrun -l` or `fgllink`) will stop with the error -1338. Before version 2.40, the undefined function was ignored.

> **Note:** Complete function reference is only checked by the linker when creation a 42r program file. When creating a 42x library, there can be references to undefined functions.

In the next example, the `main.4gl` module does not call any function, but the module used in the link line (`module.4gl`) defines an unused function (`f1`) calling an undefined function (`f2`):

`main.4gl`:

```
MAIN
    DISPLAY "In main..."
END MAIN
```

`module.4gl`:

```
FUNCTION f1() -- Unused in program
    DISPLAY "In f1..."
    CALL f2() -- Undefined
END FUNCTION
```

Compiling and linking:

```
$ fglcomp main.4gl
$ fglcomp module.4gl
$ fgllink -o prog.42r main.42m module.42m
ERROR(-1338):The function 'f2' has not been defined in any module in the
 program.
```

2.32 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.32.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Front-end protocol compression disabled on page 113
2. SQLite driver does no longer need libiconv on Windows on page 113

3. Need for Informix CSDK to compile C extensions on page 113
4. FESQLC tool removal on page 113

Front-end protocol compression disabled
GUI communication does not require protocol compression on LAN networks.

Until version 2.32.00, front-end protocol compression was enabled by default, to speed GUI communication on slow networks. However, on regular networks, compression is useless and can be disabled to save processing resources. With version 2.32.00, the compression is now disabled by default. If needed, compression can be enabled with this FGLPROFILE entry:

```
gui.protocol.format = "zlib"
```

Note also that compression needs the zlib library to be present on the computer where fglrun executes. Starting with 2.32.00, the product package does no longer include the fallback zlib library ($FGLDIR/lib/libzfgl.so or %FGLDIR%\bin\libzfgl.dll). If no standard zlib is installed on your system, compression will not be possible.

SQLite driver does no longer need libiconv on Windows™
UTF-8 string data storage in SQLite requires conversion when the application is not UTF-8.

Starting with version 2.32, the SQLite driver (dbmsqt3xx) does no longer need the LIBICONV.DLL library on Windows™ platforms to do charset conversion, when the application locale is not UTF-8.

Need for Informix® CSDK to compile C extensions
Compiling C Extensions requires now the Informix CSDK.

> **Note:** This upgrade note is related to C Extensions or ESQL/C Extensions, and can be ignored if your application does not use such extensions.

To compile C or ESQL/C extensions manipulating data types like DECIMAL, you need IBM® Informix® data type structure definitions such as dec_t, dtime_t, intrvl_t, as well as macros like DECLEN() or TU_ENCODE(). Before version 2.32, these C structure and macros where provided in the files of the FGLDIR/include/f2c directory.

Genero BDL version 2.32 does no longer provide the IBM® Informix® ESQL/C structure definitions in FGLDIR/include/f2c files, because we have identified that some of the definitions are platform specific. However, to compile your C extensions, you need these definitions if your extensions use complex data types such as DECIMAL, DATETIME/INTERVAL, BYTE/TEXT. The definitions are not required if you use standard C types such as int or char[].

Starting with version 2.32, you need to install an IBM® Informix® CSDK on your development machine in order to get the structure and macro definitions to compile your C extensions. Understand that the IBM® Informix® CSDK is only required on the development platform. It is not required to install the CSDK on the production machines, except of course if you want to connect to an IBM® Informix® database server.

FESQLC tool removal
The ESQL/C compiler (fesql) has been removed from the Genero BDL product.

Starting with version 2.32, the `fesqlc` compiler and linker is no longer part of the Genero BDL package.

Contact you support channel for more details.

2.30 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.30.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. GUI server auto start on page 114
2. Form compiler is more strict on page 114

3. ORACLE and INTERVAL columns on page 114
4. DIALOG.setCurrentRow() changes row selection flags on page 115
5. Schema extractor needs table owner on page 115
6. Windows installation for all users only on page 115
7. MenuAction close no longer created by default on page 115
8. Emulated scrollable cursor temp files in DBTEMP on page 115
9. Modifying tree view data during dialog execution on page 115
10. FPI tool removal on page 115

GUI server auto start
FGLSERVER defaults the server defined by wsmap settings, when starting GUI server

Before version 2.30, the runtime system was trying to connect to `localhost:0` when FGLSERVER was not set, even if `gui.server.autostart` FGLPROFILE entries are defined.

This behavior has been identified as a bug (FGL-1583) and has been fixed, changing the way fglrun proceeds with the GUI connection when autostart settings are defined; with 2.30, the `wsmap` workstation mappings are now taken into account, so that FGLSERVER defaults to `locahost:`*n*, where *n* is the GUI server number found from the wsmap settings.

> **Note:** This behavior change is related to the bug fix FGL-1583.

Form compiler is more strict
The .per grammar parser has been reviewed to deny invalid code.

In version 2.30, the internals of fglform have been reviewed to simplify the extension of the form syntax with new item types and attributes. This code review has removed some inconsistencies in the grammar parser; as a result, the form compiler is more strict regarding invalid syntaxes. Thus, you may experience compilation errors with forms that compiled with prior versions. Simply fix the invalid syntax in your forms and recompile.

ORACLE and INTERVAL columns
INTERVAL storage bug fix needs a review of existing databases in production.

Before 2.30.00 (build 1566), negative (and only negative) INTERVAL values were inserted incorrectly. This a critical bug.

For example, it was not possible to compare an INTERVAL value inserted by a program with an INTERVAL literal:

```
SELECT ... FROM table
  WHERE interval_col = INTERVAL '-55555-11' YEAR(9) TO MONTH
```

The problem concerns database columns with the following interval types:

```
INTERVAL YEAR(p) TO MONTH
INTERVAL DAY(p) TO FRACTION(n)
```

(Other INTERVAL types are stored in a CHAR(50))

A simple INTERVAL to CHAR to INTERVAL conversion will fix the values:

```
UPDATE table SET interval_col = TO_CHAR(interval_col)
```

> **Note:** This behavior change is related to the bug fix FGL-95.

DIALOG.setCurrentRow() changes row selection flags
Row selection flags are reset by a call to setCurrentRow().

Before version , the DIALOG.setCurrentRow() method was not modifying the row selection flags.

Starting with version 2.30, the method resets row selection flags to false and marks the new current row as selected.

Schema extractor needs table owner
The fgldbsch schema extractor requires a -ow option to distinguish different database users/shemas.

Starting with version 2.30, the fgldbsch schema extractor will always use a table owner / schema to select tables from databases where several schemas can hold tables with the same name.

> **Note:** This behavior change is related to the bug fix FGL-2072.

The table owner can be specified with the `-ow` option, and defaults to the user name passed with the `-un` option, or to the current database user if no `-up` option was given. The last case can occur when the database connection information is taken from the FGLPROFILE configuration file, or when the OS user authentication is used.

Windows™ installation for all users only
Installation on Windows platforms is for all users.

Starting with version 2.30, the Windows™ installer forces you to install the product for all users.

MenuAction close no longer created by default
The close action is no longer created by default in MENU dialog.

Before version 2.30, a `close` MenuAction was created by default for MENU dialogs. This action node is no longer created, except if you have a COMMAND KEY(INTERRUPT) in the MENU, or if you have your own user action handler ON ACTION close, of course. You must take this change into account if you are manipulating the AUI tree with om classes in MENUs.

Emulated scrollable cursor temp files in DBTEMP
Directory of scrollable cursor data storage can be defined with DBTEMP.

On UNIX™ platforms, starting with 2.30, the temporary files for emulated scrollable cursors will be created in the directory defined by the DBTEMP environment variable when defined, otherwise TMPDIR, TEMP or TMP will be used. Using DBTEMP for database files conforms to DBTEMP usage for temporary files of TEXT and BYTE data storage.

Modifying tree view data during dialog execution
Use ui.Dialog methods to insert/append/delete treeview nodes.

Before version 2.30, it was possible to use the `insertRow()` / `appendRow()` / `deleteRow()` / `deleteAllRows()` dialog class methods to modify the tree array during the dialog execution. But these methods were not prepared to handle tree data properly. You could use program array methods instead, but when modifying the program array directly, multi-range selection flags or cell attributes were not synchronized.

Starting with 2.30.02, you can now use the `insertNode()`, `appendNode()` and `deleteNode()` methods of the `ui.Dialog` class. You can still directly fill the program array before the dialog execution, but you should use dialog methods during the dialog execution.

FPI tool removal
The fgi tool to show version information is no longer available, use fglrun -V.

Up to version 2.30, the fpi tool was provided to print version information of the different components of Genero BDL.

Starting with 2.32.00, this tool is no longer distributed.

To print version information, you must use the -V option of fglrun.

2.21 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.21.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. PostgreSQL 8.4 and INTERVAL type on page 116
2. fglcomp --build-rdd compiles the module on page 116
3. Unique and primary key constraint violation on page 116
4. IMPORT with list of C-Extensions on page 117
5. Initializing dynamic arrays to null on page 117
6. Strict screen record definition for tables on page 117

PostgreSQL 8.4 and INTERVAL type

The `dbmpgs84x` database driver requires your database schema use the INTERVAL type, rather than a CHAR(50) type.

Version 2.21 introduced support for PostgreSQL 8.4 with the new database driver `dbmpgs84x`. This version of PostgreSQL implements a native INTERVAL data type that is similar to the Genero Business Development Language INTERVAL type.

When using the `dbmpgs84x` (and higher) driver, Informix-style INTERVAL types will be mapped / translated to native PostgreSQL INTERVALs. Prior drivers will keep using the CHAR(50) replacement. If your application is storing INTERVAL in a PostgreSQL database, you will have to modify you database schema to replace the existing CHAR(50) column with the native INTERVAL data type of PostgreSQL 8.4. If you cannot migrate the database, you can still use the older dbmpgs83x driver using CHAR(50) for INTERVALs, but that driver requires a PostgreSQL client version 8.3.

fglcomp --build-rdd compiles the module

`fglcomp --build-rdd` now creates both the `.42m` and `.rdd` files.

Before version 2.21, `fglcomp --build-rdd` only produced the `.rdd` data definition file.

This option is now a compilation option. Both `.42m` and `.rdd` files are created at the same time.

Unique and primary key constraint violation

Unique and primary key constraint violations mostly return error -268, however you might also need to check for error -269 in some instances.

When a unique or primary key constraint is violated, the IBM® Informix® driver returns the error -268 in SQLCA.SQLCODE if the database uses transaction logging, and error -239 if the database uses no logging.

Regarding non-Informix drivers, all 2.21 drivers now return -268 when a unique constraint or primary key constraint is violated. Before 2.21, the Oracle and SQL Server / Sybase drivers returned error -239, which is only returned by IBM® Informix® databases without transaction logging. Returning error -268 for all drivers is the best choice in a context of transactional databases.

Check your code for -239 error code usage and replace by -268. If you still need to test error -239 (for example because you have IBM® Informix® databases without transactions), we recommend that you write a function testing different error codes to check unique constraint violation:

```
FUNCTION isUniqueConstraintError()
  IF (SQLCA.SQLCODE==-239 OR SQLCA.SQLCODE==-268)
  OR (SQLCA.SQLCODE==-346 AND SQLCA.SQLERRD[2]==-100)
  THEN
    RETURN TRUE
  ELSE
    RETURN FALSE
```

```
  END IF
END FUNCTION
```

IMPORT with list of C-Extensions
The `IMPORT` instruction for C extensions denies a comma-separated syntax.

Before version 2.21.00, the `IMPORT` instruction for C extensions was documented as allowing a comma-separated list of libraries:

```
IMPORT lib1, lib2
```

This compiled, but at runtime only the first library was found. Using elements of the other libraries raised a runtime error.

With 2.21.00 and the new .42m module importation support, the compiler is now more strict and denies the comma-separated syntax. You must specify every library, Java™ class or .4gl module in separate lines:

```
IMPORT lib1
IMPORT JAVA myclass
IMPORT FGL mymodule
```

Initializing dynamic arrays to null
The INITIALIZE TO NULL instruction clears the dynamic array.

(This issue was actually registered as a bug/enhancement #15128)

Starting with version 2.21.00, the INITIALIZE TO NULL instruction clears the dynamic arrays (i.e. array.getLength() returns 0). Before this version, all elements of the dynamic array were kept, and set to null. Since the old behavior was documented, this behavior change required a migration note. The new behavior is expected by most programmers.

Strict screen record definition for tables
The fglform compiler of version 2.21.00 now makes a strict checking of the fields used in the screen record definition for table containers.

It generates error -6819 if the screen record do not use all columns used in the table. The order can be different, however.

> **Note:** This behavior change is related to the bug fix FGL-2701 and FGL-3174.

2.20 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.20.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Sort is now possible during INPUT ARRAY on page 118
2. Cell attributes and buffered mode on page 118
3. Field methods are more strict on page 118
4. Strict variable identification in SQL statements on page 118
5. SQL Warnings with non-Informix databases on page 119
6. SERIALREG table for 64-bit serial emulation on page 119
7. Extracting the database schema with fgldbsch on page 120
8. Database driver internal error changed from -768 to -6319 on page 120
9. Searching for image files on the application server on page 120
10. Strict action identification in dialog methods on page 120
11. Strict field identification in dialog methods on page 120
12. Form compiler checking invalid layout definition on page 121

13. Database schema compatibility on page 121
14. Predefined actions get automatically disabled according to context on page 122
15.BEFORE ROW no longer executed when array is empty on page 122
16. Controlling INPUT ARRAY temporary row creation on page 122

Sort is now possible during INPUT ARRAY
Starting with version 2.20, the built-in sort is now available during `INPUT ARRAY`. If you want to avoid sorts in a Table, use the `UNSORTABLECOLUMNS` attribute.

Cell attributes and buffered mode
Use the UNBUFFERED mode when setting cell attributes.

Before version 2.20, array cell attributes were synchronized quite often by the runtime system, and this was not very efficient. As a result, there was not much difference between using buffered or unbuffered mode; when changing cell attributes, the result was immediate even in buffered mode.

Starting with version 2.20, it is recommended that you use the UNBUFFERED mode when setting cell attributes; otherwise, the colors will not be synchronized on the front-end.

Field methods are more strict
Starting with Genero 2.20 (or when using multiple dialogs in 2.11.08 and higher), DIALOG class methods such as `setFieldActive()` need the correct field specification with the screen-record name prefix, if the field was explicitly bound with the `FROM` clause of `INPUT` or `INPUT ARRAY`.

In prior versions, the field was found by these methods even if the prefix was invalid. (Actually, the prefix was just ignored and only the fieldname was used.)

Strict variable identification in SQL statements
Program variable identification in static SQL statements is more strict in version 2.20 than older versions.

If you define a variable with the same name as a SQL object (i.e. table name, table alias), the fglcomp compiler will raise an error because it will consider the program variable first. For example, if the variable name matches the table or alias identifier, using *table.column* in the SQL statement will be resolved as *variable.member*, which does not exist.

The next code example will not compile because the program defines a variable using the same name as the table alias *c*:

```
MAIN
   DEFINE c INTEGER
   DATABASE stores
   SELECT COUNT(*) INTO cFROM customer c
          WHERE c.fnameIS NULL
END MAIN
```

The code also fails to compile with IBM® Informix® 4gl 7.32, but it did compile with version of Genero Business Development Language.

To work around this, you must either rename the program variable, or explicitly identify SQL objects with the @ prefix in the SQL statement:

```
MAIN
   DEFINE c INTEGER
   DATABASE stores
   SELECT COUNT(*) INTO cFROM customer c
          WHERE @c.fnameIS NULL
END MAIN
```

Recompile all your programs to find the conflicts.

SQL Warnings with non-Informix databases
SQL Warnings are now propagated for all database drivers, and can set the SQLCA.SQLAWARN, SQLSTATE and SQLERRMESSAGE registers.

Before version 2.20, is was impossible for a non-Informix driver to return SQL Warning information in SQLCA, SQLSTATE and SQLERRMESSAGE.

This new behavior will have no impact if you test SQL Errors with STATUS or SQLCA.SQLCODE, as these registers remain zero if an SQL Warning is raised. However, if you are using SQLSTATE to check for SQL Errors, you must now distinguish SQLSTATE of class 01: These are SQL Warnings, not SQL errors.

In this example, when connected to IBM® DB2®, the SQLSTATE register will get the value 01504 indicating that all rows of the table have been deleted. As a result, testing SQLSTATE against 00000 will evaluate to false, and run into the error handling block, which is unexpected:

```
MAIN
   DATABASE stores
   WHENEVER ERROR CONTINUE
   DELETE FROM customer
   IF SQLSTATE <> "00000" THEN
      -- handle error
   END IF
END MAIN
```

To check for successful SQL execution with or without warning, you can, for example, code:

```
MAIN
   DATABASE stores
   WHENEVER ERROR CONTINUE
   DELETE FROM customer
   IF NOT (SQLSTATE=="00000" OR SQLSTATE MATCHES "01*") THEN
      -- handle error
   END IF
END MAIN
```

SERIALREG table for 64-bit serial emulation
You must alter the `SERIALREG` table to do serial emulation on a `BIGINT` column.

The SERIALREG based serial emulation is defined by the following FGLPROFILE entry:

```
dbi.database.<dbname>.ifxemul.datatype.serial.emulation = "regtable"
```

Version 2.20 introduces the `BIGINT` data type, which is a 64-bit signed integer. You can use `BIGSERIAL` or `SERIAL8` columns with IBM® Informix®, and ODI drivers can emulate 64-bit serials in other database servers. However, if you are using serial emulation based on the `SERIALREG` table, you must redefine this table to change the `LASTSERIAL` column data type to a `BIGINT`. If the `BIGINT` data type is not supported by the database server, you can use a `DECIMAL(20,0)` instead:

```
CREATE TABLE serialreg (
  tablename   VARCHAR2(50) NOT NULL,
  lastserial  BIGINT       NOT NULL,
  PRIMARY KEY ( tablename )
)
```

> **Important:** If you need to migrate an installed database using `SERIALREG`-based triggers, you will have to keep the current registered serials and use `ALTER TABLE` instead of `CREATE TABLE`. This example shows the `ALTER TABLE` syntax for SQL Server. Check the database server manuals for the exact syntax of the `ALTER TABLE` statement.
>
> ```
> ALTER TABLE serialreg ALTER COLUMN lastserial BIGINT NOT NULL
> ```

Additionally, all existing `SERIALREG`-based triggers must be modified, in order to use `BIGINT` instead of `INTEGER` variables, otherwise you will get `BIGINT` to `INTEGER` overflow errors. For example, to modify existing triggers with SQL Server, you can use the `ALTER TRIGGER` statement, which can be easily generated from the database browser tool (there is a *modify* option in the contextual menu of triggers). After the existing trigger code was generated, you must edit the code to replace the `INTEGER` data type by `BIGINT` in the variable declarations, and execute the `ALTER TRIGGER` statement.

Extracting the database schema with fgldbsch
The fgldbsch database schema extraction tool has been updated to map native database types to newly-added types.

Version 2.20 implements new data types such as BIGINT and BOOLEAN. The fgldbsch database schema extraction tool has been reviewed to map native database types to these new types when possible. Pay attention to these changes, when extracting a schema from your database.

For example, before version 2.20, fgldbsch converted an Oracle NUMBER(20,0) to a DECIMAL(20,0) by default. Now, since 2.20 provides the BIGINT native FGL type, it can be used to store a NUMBER(20,0) from Oracle.

You can get the previous behavior by using a conversion directive with the `-cv` option of fgldbsch.

To see the new conversion rules, run the fgldbsch tool with the `-ct` option.

Database driver internal error changed from -768 to -6319
The internal error raised was changed to avoid conflicts with an IBM® Informix® SQL error code.

Prior to version 2.20, if an unexpected error occurred in a database driver, the driver could return error -768, which is a real IBM® Informix® SQL error that instructs the user to call the IBM® support center.

To avoid any mistake, 2.20 database drivers return now the error -6319 if an internal error occurs, which is a Genero Business Development Language specific error message that suggests you to set the FGLSQLDEBUG environment variable to get detailed debug messages.

Searching for image files on the application server
For security reasons, the image file transfer mechanism has been slightly modified in version 2.20.

(This modification has also been back-ported in 2.11.14):

If FGLIMAGEPATH is set, the current working directory is no longer searched as in previous versions. You must explicitly add "." to the list of directories. By default, if FGLIMAGEPATH is not defined, the runtime system still searches the current directory.

If FGLIMAGEPATH is defined, the image files used in IMAGE form fields or in the IMAGE attribute must be located below one of the directories listed in the environment variable. This constraint does not exist if FGLIMAGEPATH is not set and has been relaxed in 2.21.00 for image fields displayed by program.

Starting with 2.21.00, images displayed by program to IMAGE fields are considered as valid files to be transferred to the clients without risk and do not follow the FGLIMAGEPATH security restrictions. Images are however searched according to the path list defined in FGLIMAGEPATH.

Strict action identification in dialog methods
Actions referenced in methods of the dialog class must exist in the current dialog, or an error is raised.

Starting with version 2.20.00, dialog class methods like `ui.Dialog.setActionActive()` can now raise a runtime error -8089 if the action name is invalid. Before version 2.20, the method ignored the invalid action name, and it could take a while for the programmer to find the mistake.

Strict field identification in dialog methods
Fields referenced in methods of the dialog class must exist in the current dialog, or an error is raised.

Starting with version 2.20.05, dialog class methods like `ui.Dialog.setFieldTouched()` can now raise a runtime error -1373 if the field specified does not match a field in the current dialog. Before version

2.20.05, these methods previously ignored the invalid field specification, and it could take a while for the programmer to find the mistake.

Form compiler checking invalid layout definition
It is better to identify form layout mistakes when the form is compiled, rather than at runtime.

Starting with version 2.20.05, the fglform compiler performs more layout checking than before. Thus, existing (invalid) forms that compiled with prior versions of Genero may no longer compile with 2.20.05. This strict checking is done to detect layout mistakes during form design, instead of having the front-ends render invalid forms in a unknown manner at run time.

For example, the following form definitions are invalid and will raise a compilation error with fglform:

```
SCHEMA FORMONLY
LAYOUT
GRID
{
  [f01:  |f02    ]     -- HBox layout tags in lists are denied
  [f01:  |f02    ]
  [f01:  |f02    ]
  [f01:  |f02    ]
}
END
END
```

```
SCHEMA FORMONLY
LAYOUT
GRID
{
  [f01    ]  [f02    ]
  [f01    ]       [f02    ]   -- Misaligned field tags (vertical)
  [f01    ]  [f02    ]
  [f01    ]       [f02    ]
}
END
END
```

```
SCHEMA FORMONLY
LAYOUT
GRID
{
  [f01][f01][f01]  [f01]   -- Misaligned field tags (horizontal)
}
END
END
```

Database schema compatibility
Version 2.20.06 database schema extraction now generates a different type code for BOOLEAN, that introduces a compatibility issue with older versions of fglcomp and fglform.

> **Note:** This behavior change is related to the bug fix FGL-2048.

If database tables use data types that are equivalent to the BOOLEAN Informix® type, such as the BIT type in SQL Server, you must regenerate the .sch database schema file with the fgldbsch tool. If you keep using the schema generated by an older version such as 2.20.04, fglcomp or fglform will raise the error -6634.

This problem will only occur if your database tables use the BOOLEAN (or native equivalent type). See ODI Adaptation Guides for more details about database specific boolean types.

Predefined actions get automatically disabled according to context
Dialogs will automatically disable some predefined actions, if it makes no sense to trigger the action in the current context.

Starting with version 2.20, (or with version 2.10 when `FGL_USENDIALOG=1`), the dialogs will automatically disable some predefined actions if it makes no sense to trigger the action in the current context. For example, during an `INPUT ARRAY`, if there are no rows to remove, the predefined delete action will be disabled automatically. Similarly, the insert and append actions get disabled when the array is full (this can happen with static arrays or when using the `MAXCOUNT` attribute). The predefined actions will also be disabled if you overwrite them with your own `ON ACTION` handler.

BEFORE ROW no longer executed when array is empty
In order to trigger the BEFORE ROW block when entering an array, the array must not be empty.

Before version 2.20, the `BEFORE ROW` block was always executed when entering a `DISPLAY ARRAY` or `INPUT ARRAY` dialog, even if the number of real data rows was zero. Starting with 2.20, when using an empty dynamic array or when using a static array and specifying zero data rows with a `SET_COUNT(0)` call or with the `COUNT=0` attribute, the `BEFORE ROW` control block is no longer executed when the dialog starts.

The `BEFORE ROW` block will be executed when a new row is created in `INPUT ARRAY`. When entering an `INPUT ARRAY` with an empty array, a new temporary row is created by default, except if you use the `AUTO APPEND = FALSE` attribute.

Controlling INPUT ARRAY temporary row creation
The INPUT ARRAY dialog and sub-dialog provides the APPEND ROW and AUTO APPEND attributes to control row creation at the end of a list (known as *temporary row creation*).

`APPEND ROW` controls explicit temporary row creation, while `AUTO APPEND` controls automatic temporary row creation.

Starting with version 2.20, moving down after the last row (with the mouse or keyboard) or leaving the last column of the last row with a TAB key are considered events that trigger automatic temporary row creation.

Before version 2.20, these cases were considered as events for an explicit temporary row creation. In other words, if you want to deny temporary row creation in such case, it is now done with `AUTO APPEND = FALSE` while in older versions it was controlled by `APPEND ROW = FALSE`.

2.11 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.11.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check previous upgrade guides if you migrate from an earlier version.

1. Writing timestamp information in p-code modules on page 122

Writing timestamp information in p-code modules
A compilation timestamp is no longer automatically written to p-code files, when the source code is not modified.

Before release 2.10, the 42m p-code files were stamped with a compilation timestamp. This information changed after every compilation, even if the source code was not modified.

Since 2.10, the timestamp information is no longer written to p-code files by default, allowing 42m file comparison, checksum creation, or storage of 42m file in versioning tools. Version 2.11.05, provides the --timestamp fglcomp option to force a timestamp in p-code modules:

```
$ fglcomp --timestamp mymodule.4gl
$ fglrun -b mymodule.42m
2008-12-24 11:22:33 2.11.05-1169.84 /home/devel/stores/mymodule.4gl 15
```

2.10 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.10.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check previous upgrade guides if you migrate from an earlier version.

1. XML declaration added automatically on page 123

XML declaration added automatically

The XML declaration is added automatically when writing XML files.

An XML file must start with a "Prolog" or "XML Declaration" defining the XML version and character set used by the file:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<root ...>
...
</root>
```

Starting with Genero version 2.10.05, the XML declaration is now added automatically when writing XML files.

> **Note:** This behavior change is related to the bug fix FGL-285.

Before 2.10.05, you could workaround this by writing this header yourself as a processing instruction, but this solution was subject to mistakes: The non-ASCII characters written to the XML file must match the encoding specification in the XML Declaration.

To avoid invalid character set definitions, the Genero BDL built-in classes now add the XML Declaration with the `encoding` attribute defined according to the current locale used by the runtime system. The value written in the `encoding` attribute is defined by the charmap.alias file.

2.02 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.02.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Automatic HBox/VBox on page 123

Automatic HBox/VBox

Starting with version 2.02.01, the form compiler automatically adds HBox and VBox containers with splitter around stretchable form elements that are placed side-by-side.

When recompiling your forms with this new version of fglform, the generated .42f can get additional HBox/VBox nodes even if you did not touch the .per source file.

2.01 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.01.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

There is no upgrade note with this version.

2.00 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 2.00.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Runner creation is no longer needed on page 124
2. Desupported Informix client environments on page 124
3. Desupported database drivers on page 124
4. fglmkrtm tool removed on page 125
5. fglinstall tool removed on page 125
6. Linking the utility functions library on page 125
7. Dynamic C extensions on page 125
8. WANTCOLUMNSANCHORED is desupported on page 125
9. PIXELWIDTH / PIXELHEIGHT are desupported on page 125
10. Pre-fetch parameters with Oracle on page 126
11. Preprocessor directive syntax changed on page 126
12. Static SQL cache is removed on page 126
13. Connection database schema specification on page 127
14. Changes in the schema extraction tools on page 127
15. Global and module variables using the same name on page 128
16. Connection parameters in FGLPROFILE when using Informix on page 128
17. Inconsistent USING clauses on page 128
18. Usage of RUN IN FORM MODE on page 129
19. TTY and COLOR WHERE attribute on page 129

Runner creation is no longer needed

Starting with version 2.00, you no longer need to recompile/build a runner.

The runtime system architecture is now based on shared libraries (or DLLs on Windows™), and the database drivers are automatically loaded according to FGLPROFILE configuration parameters.

If you have C extensions, you must rebuild them as shared libraries.

> **Important:** Database vendor client libraries (libclntsh, libcli, libpq, libaodbc) must be provided as shared objects (or DLL on Windows™).

Desupported Informix® client environments

We strongly recommend you to upgrade the IBM® Informix® Client Software Development Kit (CSDK) to the most recent version supported by Genero BDL.

The database interface of Genero Business Development Language (BDL) version 2.00 was redesigned to allow dynamic loading of database drivers. The following IBM® Informix® drivers and environments have been desupported with this redesign:

- ix210: Informix® ESQL/C 2.10
- ix410: Informix® ESQL/C 4.10
- ix501: Informix® ESQL/C 5.01
- ix711: Informix® ESQL/C 7.11
- ix720: Informix® ESQL/C 7.20

If required, old IBM® Informix® drivers can be re-enabled in a next Genero BDL version. However, we strongly recommend you to upgrade the IBM® Informix® Client Software Development Kit (CSDK) to the most recent version supported by Genero BDL.

Desupported database drivers

Database drivers for old database client versions are removed according to the vendor de-support plans.

Database drivers desupported in versions 2.00:

- Adabas D 12 (dbmabd12)
- PostgreSQL 7 (dbmpgs7x)

- SQL Server 7 (dbmmsv7x)

fglmkrtm tool removed
The fglmkrtm tool has been removed, as database drivers are loaded dynamically.

Starting with version 2.00, database drivers are now always loaded dynamically. Thus the fglmkrtm tool has been removed from the distribution. This tool was provided in previous versions to create a fglrun runner with the correct database driver.

Refer to Database connections on page 457 for more details about database driver configuration.

fglinstall tool removed
The fglinstall tool has been removed from the distribution.

This tool was provided in previous versions to compile product message files, form files, and program modules provided in the distribution. The compiled versions of all these files are now included in the package.

Linking the utility functions library
All utility functions are in the libfgl4js.42x library, up until 2.21.

Prior to version 2.00, some utility functions (canvas draw* and database db_* functions) were linked automatically to the 42r program when using fglrun -l or fgllink. These functions are implemented in the fgldraw.4gl and fgldbutl.4gl modules, which were linked in the libfgl.42x library and loaded automatically at runtime by fglrun.

Starting with version 2.00, all utility functions are now in the libfgl4js.42x library. So, if you use the draw* or db_* utility functions, you must now add the libfgl4js.42x library explicitly when using fglrun -l or fgllink, or you can use the fgl2p tool to link .42r programs. The fgl2p tool links the program with the libfgl4js.42x library by default.

Starting with version 2.21, the libfgl.42x library is no longer provided.

Dynamic C extensions
Dynamic C extensions are automatically loaded according to `IMPORT` instructions.

Prior to version 2.00, you must use FGLPROFILE entries to specify Dynamic C extensions to be loaded at runtime.

Starting with version 2.00, Dynamic C extensions are automatically loaded according to `IMPORT` instructions. The FGLPROFILE entries are no longer used.

> **Important:** Global variables (userData) can no longer be shared between the runtime system and the C extensions. You must use functions to pass global variable values.

There is no longer a need to define the FGL_API_MAIN macro in the extension interface file.

All C data type definitions are now centralized in the fglExt.h header file, header files like Date.h, MyDecimal.h have been removed from the distribution.

WANTCOLUMNSANCHORED is desupported
Use `UNMOVABLECOLUMNS` to specify that table columns cannot be moved around by the user.

Before version 2.00, the `WANTCOLUMNSANCHORED` attribute was undocumented but still supported by the language, to simplify migration from 1.20.

Starting with version 2.00, the `WANTCOLUMNSANCHORED` attribute is de-supported; you must use `UNMOVABLECOLUMNS` to specify that table columns cannot be moved around by the user.

PIXELWIDTH / PIXELHEIGHT are desupported
Use the `WIDTH` and `HEIGHT` attributes to specify the size of an image.

Before version 2.00, the `PIXELWIDTH` and `PIXELHEIGHT` attributes were used to specify the real size of an `IMAGE` form item.

Starting with version 2.00, you must use the `WIDTH` and `HEIGHT` attributes to specify the size of an image:

In the .per form file:

```
IMAGE img1 = FORMONLY.image1,
      HEIGHT = 100 PIXELS,
      WIDTH = 100 PIXELS;
```

The `PIXELWIDTH` and `PIXELHEIGHT` attributes are still supported by the form compiler, but are deprecated and will be removed in a future version.

Pre-fetch parameters with Oracle
Pre-fetch parameters allow an application to automatically fetch rows from the Oracle database when opening a cursor.

Before version 2.00, the default pre-fetch parameters are 50 rows and 65535 bytes for the pre-fetch buffer. Some customers experienced a huge memory usage with those default values, when using a lot of cursors: It appears that the Oracle client is allocating a buffer of pre-fetch.memory (i.e. 64 Kbytes) for each cursor.

Starting with version 2.00, the default is 10 rows and 0 (zero) bytes for the pre-fetch buffer (memory), meaning that memory is not included in computing the number of rows to pre-fetch.

Preprocessor directive syntax changed
The preprocessor directives use an ampersand character (&) instead of a sharp (#) character.

Before version 2.00, the preprocessor directives start with a (#) sharp character, to be compliant with standard preprocessors (like **cpp**). This caused too many conflicts with standard language comments that use the same character:

```
#include "myheader.4gl"
# This is a comment
```

Starting with version 2.00, the preprocessor directives use an ampersand character (&):

```
&include "myheader.4gl"
FUNCTION debug( msg )
  DEFINE msg STRING
&ifdef DEBUG
  DISPLAY msg
&endif
END FUNCTION
```

The preprocessor is now integrated in the compiler, to achieve faster compilation.

> **Important:** To simplify the migration, the # sharp character is still supported when using the -p fglpp option of compiler. However, you should review your source code and use the & character instead; # sharp will be desupported in a future version.

Static SQL cache is removed
The Static SQL Cache has been removed.

Before version 2.00, the size of the static SQL cache is defined by a FGLPROFILE entry:

```
dbi.sql.static.optimization.cache.size = max
```

This entry was provided to optimize SQL execution without touching code using a lot of static SQL statements, especially when using non-Informix databases where the execution of static SQL statements is slower than with Informix®. This is useful for fast migrations, but there were a lot of side effects and unexpected errors.

Starting with version 2.00, the Static SQL Cache has been removed for the reasons described. Programs continue to run without changing the code, but if you want to optimize program execution, you must use dynamic SQL (PREPARE + EXECUTE).

Connection database schema specification
Oracle- and DB2-specific FGLPROFILE entries can be specified to define the database schema at runtime.

Before version 2.00, an FGLPROFILE entry can be specified to define the database schema at runtime:

```
dbi.database.dbname.schema = "schema-name"
```

This entry could be used to select the native database schema after connecting to the server, for Oracle and DB2 only.

Starting with version 2.00, this entry is now specific to the Oracle and DB2 database driver configuration parameters:

```
dbi.database.dbname.ora.schema = "schema-name"
dbi.database.dbname.db2.schema = "schema-name"
```

For other database servers, this configuration parameter is not defined.

> **Important:** It is no longer possible to specify the "schema" parameter in the connection string (dbname+schema='name').

Changes in the schema extraction tools
The fgldbsch schema extractor is recommended, and has been enhanced.

Unique tool

Version prior to 2.00 provide two schema extractors: fglschema and fgldbsch. The first can only extract schemas from Informix® databases, while the second one can extract schemas from all supported databases.

Starting with version 2.00, the fgldbsch tool has been extended to support the old fglschema options, and fglschema has been replaced by a simple script calling fgldbsch. When you call fglschema, you actually call fgldbsch. We recommend that you use fgldbsch with its specific command line options.

System tables

In 2.0x, fgldbsch does not extract system tables by default. You must specify the -st option to get the system tables description in the schema files.

Remote synonyms

The original fglschema tool was searching for *remote synonyms* with Informix® databases. The fgldbsch tool of version 2.00 does not search for remote synonyms.

Public and private synonyms

Since version 1.32.1b (build 620.313), fgldbsch does not extract *private synonyms* anymore. Only *public synonyms* are extracted. The **.sch** schema files do not contain table owners, and if two *private synonyms* have the same names, there is no way to distinguish them in the schema files. Therefore, to avoid any mistakes, *private synonyms* are not extracted anymore.

> **Note:** This behavior change is related to the bug fix FGL-1033.

Global and module variables using the same name
Since version 2.00, when you declare a module variable with the same name as a global variable, a compilation error must be thrown.

> **Note:** This behavior change is related to the bug fix FGL-114.

This is critical to avoid confusion with the variable usage:

```
GLOBALS
  DEFINE level INTEGER
END GLOBALS
```

```
GLOBALS "globals.4gl"
DEFINE level INTEGER
FUNCTION func1()
  LET level = 123  -- is this the global or the module variable?
END FUNCTION
```

Before version 2.00, the compiler did not detect this and the module variable was used, but one might want to use the global variable instead!

If you have module variables defined with the same name as global variables, the compiler now raises the following error:

```
-4319: The symbol 'variable-name' has been defined more than once.
```

You can easily fix this by renaming the module variable. There is no risk to do this modification, because in versions before 2.00, the module variable was used, not the global variable.

Remark: The compiler now also detects duplicate global variable declaration. Just remove the duplicated lines in your source.

Connection parameters in FGLPROFILE when using Informix®
The dbi.database.* connection parameters defined in FGLPROFILE are used by the Informix® driver

Before version 2.00, the dbi.database.* connection parameters defined in FGLPROFILE are ignored by the Informix® drivers.

Starting with version 2.00, the dbi.database.* connection parameters defined in FGLPROFILE are used by the Informix® driver, as well as other database vendor drivers. For example, if you connect to the database "stores", and you have the following entries defined, the driver tries to connect as "user1" with password "alpha":

```
dbi.database.stores.username = "user1"
dbi.database.stores.password = "alpha"
```

You typically get SQL errors -387 or -329 when the wrong database login or the wrong database name is used.

Inconsistent USING clauses
Having data types changing at each execute is no longer supported.

> **Important: This issue applies to non-Informix databases only.**

Before version 2.00, it was possible to execute a prepared statement with the variable list changing at each EXECUTE statement:

```
DEFINE var1 DECIMAL(6,2)
DEFINE var2 CHAR(10)
DEFINE var3 DATE
```

```
PREPARE st1 FROM "INSERT INTO tab1 VALUES ( ?. ?, ? )"
EXECUTE st1 USING var1, var2, var3
EXECUTE st1 USING var2, var3, var1   -- different order = different data
 types
```

The database interface of version 2.00 has been rewritten for better performance. Having data types changing at each execute is no longer supported.

Error -254 will be raised if different data types are used in subsequent EXECUTE statements (with the same statement name).

Usage of RUN IN FORM MODE
RUN ... IN LINE MODE is recommended to run interactive applications.

Before version 2.00, RUN...IN FORM MODE was recommended to run interactive applications.

Starting with version 2.00, RUN ... IN LINE MODE is recommended to run interactive applications. The RUN command should be used as follows (in both GUI and TUI mode):

1. When starting an interactive program, either use RUN ... IN LINE MODE or, if the default mode is LINE MODE, use the RUN instruction without any option.
2. When starting a batch program that does not display any message, you should use RUN ... IN FORM MODE.

For more details about the RUN options, see the RUN instruction.

TTY and COLOR WHERE attribute
All type of fields now allow TTY attributes and the conditional COLOR WHERE attribute.

Before version 2.00, only some field types like EDIT or TEXTEDIT could support TTY attributes (COLOR, REVERSE), and the conditional COLOR WHERE attribute.

Starting with version 2.00, all type of fields now allow TTY attributes and the conditional COLOR WHERE attribute. So when using any ATTRIBUTES(*tty-attribute*) in programs, all fields will now be affected.

For example, CHECKBOX and RADIOGROUP fields will now get a colored background, this was not the case in prior versions.

1.33 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 1.33.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Desupported database drivers on page 129

Desupported database drivers
Database drivers for old database client versions are removed according to the vendor de-support plans.

Database drivers desupported in versions 1.33:

- MySQL 3.23.x (dbmmys32x)

1.32 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 1.32.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

There is no upgrade note with this version.

1.31 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 1.31.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

There is no upgrade note with this version.

1.30 upgrade guide

These topics describe product changes you must be aware of when upgrading to version 1.30.

> **Important:** This is an incremental upgrade guide that covers only topics related to a specific version of Genero. Also check prior upgrade guides if you migrate from an earlier version.

1. Action and field activation on page 130
2. Using HBox tags in forms on page 130
3. Width of ButtonEdit/DateEdit/ComboBox on page 133
4. Form fields default sample on page 135
5. Size policy for ComboBoxes on page 136
6. Action defaults at form level on page 139
7. Compiled string files have now .42s extension on page 139

Action and field activation

Dialog methods can be used to control action and field activation.

Version 1.30 provides dialog methods to control action and field activation:

- ui.Dialog.setActionActive(action-name, TRUE/FALSE)
- ui.Dialog.setFieldActive(field-name, TRUE/FALSE)

Previous versions allowed to modify directly the 'active' attribute of the underlying DOM node in the AUI tree; This is now forbidden: It is mandatory to use the methods to enable/disable action or fields. The dialog will synchronize the 'active' attribute in the AUI tree according to the value passed to the methods and according to the context (some actions or fields can be automatically disabled).

Using HBox tags in forms

You can use HBox tags to stack form items horizontally, without being influenced by elements above or below.

Version 1.30 supports now HBox tags to stack form items horizontally without being influenced by elements above or below.

In an HBox there is a free mix of Form Fields, labels, and Spacer Items possible.

A typical usage of an HBox Tag is to have zipcode/city form fields side by side with predictable spacing in-between.

The "classic" layout would look like the following form definition:

```
<G "User Data(version 1.20)"                                        >
 Last Name     [l_name           ]First Name[f_name          ]
 Street        [street                                       ]
 City          [city          ]Zip Code[zip  ]
 Phone(private)[phone           ] At work [                 ]
 Code          [aa]-[ab]-[ac]
```

In Figure 7: HBox tag example screenshot on page 131 you will notice that the distance between "`l_name`" and "`First Name`" is smaller than between "`First Name`" and "`f_name`". How can this be? Two lines under, there is the "zip" field which affects this distance.

If we put HBox Tags around the fields we want to group horizontally together, we get the predictable spacing between "l_name","First Name" and "f_name".

```
<G "User Data in HBoxes stacked"                                    >
Last Name     [l_nameh         :"First Name":f_nameh         ]
Street        [streeth                                      :]
City          [cityh        :"Zip Code":ziph :              ]
Phone(private)[phoneh          :"At work":phonewh          : ]
Code          [ba:"-":bb:"-":bc:                             ]
```

Here "l_nameh", "First Name" and "f_nameh" are together in one HBox; the ":" colon acts as a separator between the 3 elements.

The width of an element is calculated from the space between "[" and ":" (width of cityh is 14), or from the space between ":" and ":" (width of "bb" is 2), or from the space between ":" and "]" (width of "f_nameh" is 16). The "zip" field in the version 1.20 example has a width of five and the "ziph" field has also a width of five.

In the second Groupbox in Figure 7: HBox tag example screenshot on page 131 you will notice that the HBox is smaller than the first one, even though it uses two characters more in the screen definition. The reason is that each HBox occupies only ONE cell in the parent grid, and the content in one HBox is independent of the content in another HBox. This relaxes the parent grid; it has to align only the edges of the HBoxes and the labels left of the HBoxes. The two extra characters in the Form file for the second Group come from the fact that the labels need quoting to distinguish them from field definitions. Of course, you could use a Label field if the two extra characters are unwanted (which is done in the third Groupbox).

The third Groupbox shows how the alignment in an HBox can be affected by putting empty elements (: :) inside the HBox Tag:

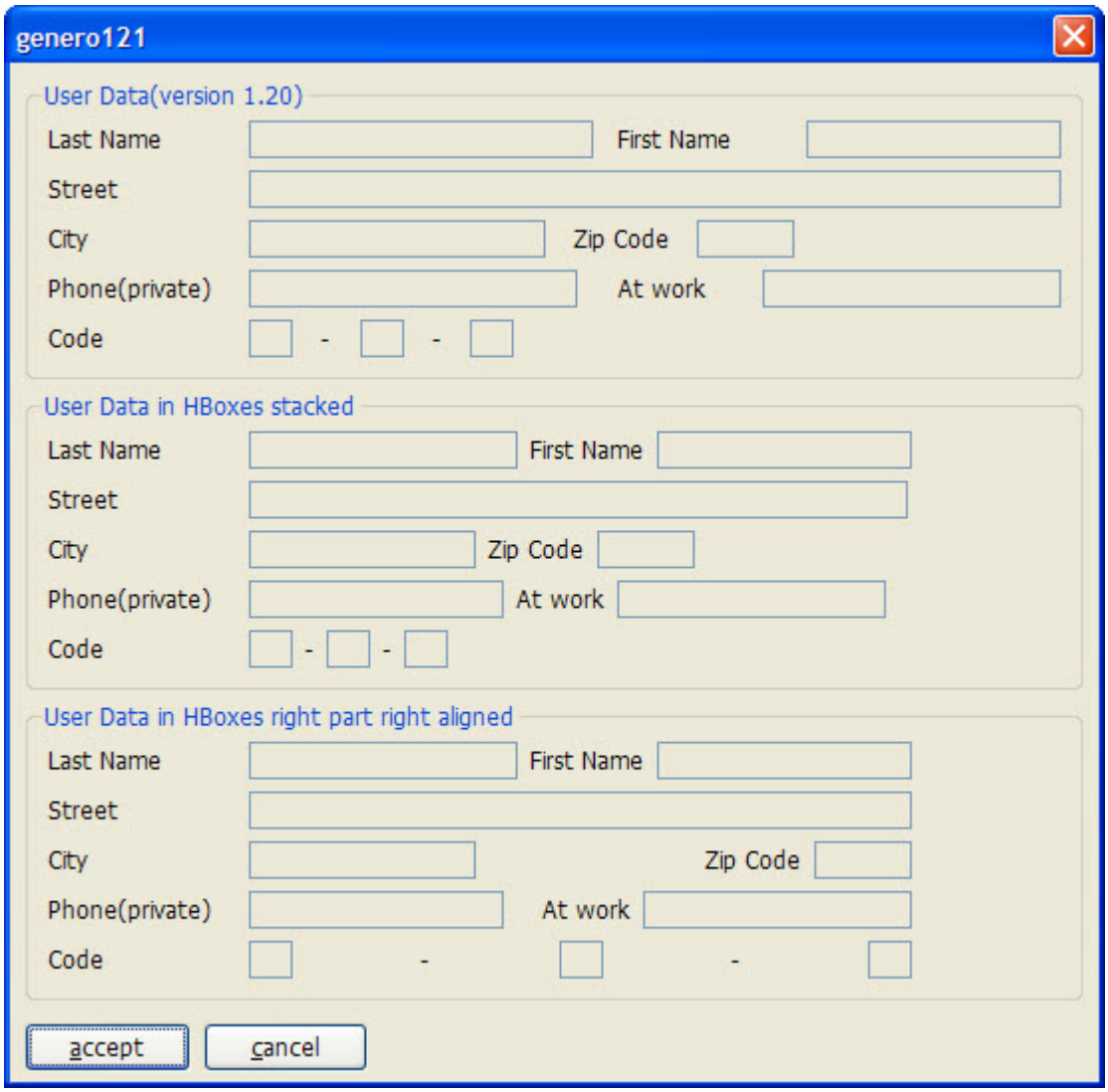
```
<G "User Data in HBoxes right part right aligned"                   >
Last Name     [l_nameh2        : :lfirsth2:f_nameh2          ]
Street        [streeth2                                      ]
City          [cityh2          :                  :lzip:ziph2]
Phone(private)[phoneh2        :      :latw:phonewh2          ]
Code          [ca:   "-"        :cb:        "-"           :cc]
```

Between "l_nameh2" and "lfirsth2" there are two ":" signs with a white space between them. This means: put a Spacer Item between l_nameh2 and lfirsth2, which gets all the additional space if the HBox is bigger than the sum of l_nameh2, lfirsth2 and f_nameh2. The number of spaces, however, has no effect. The spacer item between cityh2 and lzip has the same force as the spacer between l_nameh2 and lfirsth2.

You can treat a spacer item like a spring. The spacer item between cityh2 and lzip presses cityh2 to the left-hand side, and the rest of the fields to the right-hand side. In the "Code" line there is more than one spacer item; they share the additional space among them. (The "Code" HBox sample in the third line is only to show how spacer items work; we always advise using "Code" as in the second Groupbox, or to use a picture)

In general we advise using the approach shown in the second Groupbox: stack the items horizontally by replacing field ends with ":". This is the easy way to remove unwanted horizontal spacing.

Figure 7: HBox tag example screenshot



A big advantage in using elements in an HBox tag is that the fields get their real sizes according to the .per definition.

```
LAYOUT
GRID
{
<G g1 >
[a    ] single Edit Field

<G g2 >
 MMMMM
[b    ] The large label expands the Edit Field

<G g3 >
 MMMMM
[c   :]The large label has no influence on the Edit width

}

END
END
ATTRIBUTES
```

```
EDIT a = formonly.a, sample="0", default="12345";
EDIT b = formonly.b, sample="0", default="12345";
EDIT c = formonly.c, sample="0", default="12345";
END
```

In the second Groupbox, the edit field is expanded to be as large as the label above; using an HBox prevents this.

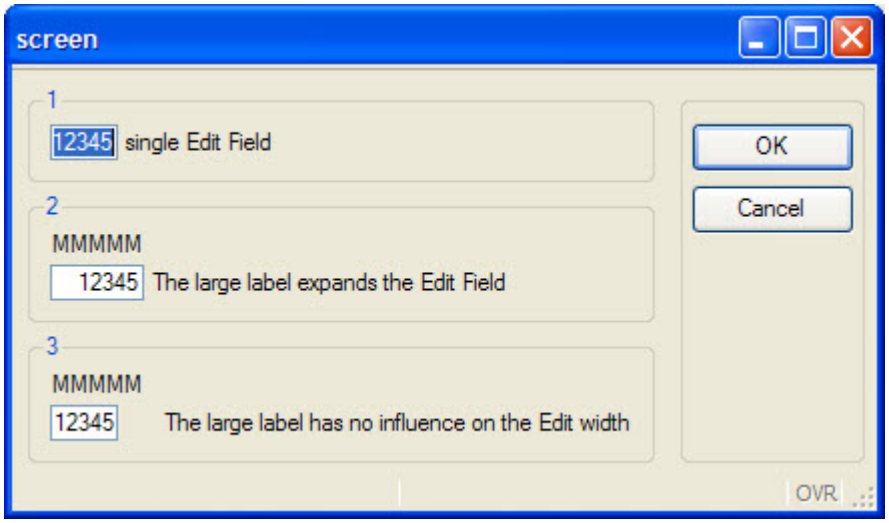


Figure 8: Use of HBox

> **Note:** in this example, we use a sample of "0" to display *exactly* five numbers.

HBox Tags limitations

HBox Tags don't work for fields of Screen Arrays or Tables; you will get a form compiler error. The reason is that the current AUI structure does not allow this. The front end needs a `Matrix` element directly in a `Grid` or a `ScrollGrid` to perform the necessary positioning calculations for the individual fields.

Width of ButtonEdit/DateEdit/ComboBox

When using `BUTTONEDIT`/`COMBOBOX`/`DATEEDIT` fields, you should account for the width of the widget button in addition to the input area.

The problem with BUTTONEDIT, DATEEDIT and COMBOBOX in versions prior to 1.30 is that a field `[b ]` got the width 3, the same width as an edit field with the same layout.

For example:

```
LAYOUT
GRID
{
  [e  ]
  [b  ]
}
END
END
ATTRIBUTES
EDIT e=formonly.e;
BUTTONEDIT b=formonly.b;
END
```

In this example, the outer (visual) width of both elements was the same, but the edit portion of "b" was much smaller, because the button did not count at all. (In practice this meant that on average only one and

a half characters of "b" was visible). However, you could input 3 characters! This made a BUTTONEDIT where you could see only one character and input only one character without tricks impossible.

Starting with version 1.30, for the Button, the Form Compiler subtracts two character positions from the width of BUTTONEDIT/COMBOBOX/DATEEDIT. This is possible because now the form compiler differentiates the width of the widget from the width of the entry part.

In fact, there is no *visual* difference between version 1.20 and 1.30 regarding this example, but in version 1.30 you can only enter one character, which is visually more correct.

In the example the BUTTONEDIT aligns with the Edit; that's why the Edit part of the BUTTONEDIT is usually still a bit bigger than one character (this depends on the button size, but if a button edit is contained by an HBox, it will get the exact size of "width" multiplied by the average character pixel width.

To express the BUTTONEDIT/COMBOBOX/DATEEDIT layout more visually, it is possible to specify:

```
[e  ]
[b- ]
```

the "-" sign marks the end of the edit portion and the beginning of the button portion (edit width ="1", widget width ="3").

The two characters are also subtracted for a BUTTONEDIT which is child of an HBox.

```
 [b  :]
```

gets also width="1" , but no widget width, because the HBox stacks the elements horizontally without needing widget width definition.

The two extra characters are only used to show the real size relations more WYSIWYG, and to have the same calculation as in a field without an HBox parent.

```
[e1:e2:e3:     ]
[b1  :b2  :b3  ]
```

shows that three BUTTONEDIT fields are much larger than three EDIT fields with the same width.

You can even write:

```
[e1:e2:e3:     ]
[b1- :b2- :b3- ]
```

or:

```
[e1:e2:e3:  ]
[b1-:b2-:b3-]
```

to use slim buttons and

```
[e1:e2:e3:        ]
[b1-  :b2-  :b3-  ]
```

if one uses large buttons to get the maximum WYSIWYG effect.

Please note that buttons do not grow if two characters "- " is expanded to three characters "- "; the button always computes its size from the image used, it's just to reserve more space in the form to match the real size.

Form fields default sample

An algorithm is used to compute the field width when no SAMPLE attribute is specified.

Starting with version 1.30, if no SAMPLE attribute is specified in the form files, the client uses an algorithm to compute the field width. In this case, a very pessimistic algorithm is used to compute the field widths: The client assumes a default SAMPLE of "M" for the first six characters and then "0" for the subsequent characters and applies this algorithm to all fields, with some exceptions like DATEEDIT fields.

The default algorithm tends to produce larger forms compared to forms used in BDS V3 and very first versions of Genero. Do not hesitate to modify the SAMPLE attribute in the form file, to make your fields shorter.

If you do not want to touch all your forms, a more tailored automatic solution would be to specify a ui.form.setDefaultInitializer() function, to set the SAMPLE depending on the AUI tag. In this example small UPSHIFT fields get a sample of "M"; all other fields get a sample of "0". This will preserve the original width for UPSHIFT fields, however numeric and normal String fields will get the sample of "0" and make the overall width of the form smaller.

Program:

```
# this demo program shows how to affect the "sample" attribute in a
#  ui.form.setDefaultInitializer function
# the main concern is to set a default sample of "0" and to
# correct the sample attribute for small UPSHIFT fields to "M"
# to be able to display full uppercase letter for fields with a small width

MAIN
  DEFINE three_char_upshift CHAR(3)
  DEFINE three_digit_number Integer
  DEFINE longstring CHAR(100)
  CALL ui.form.setDefaultInitializer("myinit")
  OPEN form f from "sampletest2"
  DISPLAY form f
  INPUT BY NAME three_char_upshift,three_digit_number,longstring
END MAIN

FUNCTION myInit(f)
  DEFINE f ui.Form
  CALL checkSampleRecursive(f.getNode())
END FUNCTION

FUNCTION checkSampleRecursive(node)
  DEFINE node,child om.DomNode
  LET child= node.getFirstChild()
  WHILE child IS NOT NULL
    CALL checkSampleRecursive(child)
    CALL setSample(child)
    LET child=child.getNext()
  END WHILE
END FUNCTION

FUNCTION setSample(node)
  DEFINE node,parent om.DomNode
  LET parent=node.getParent()
  -- only set the "sample" for FormFields in this example
  IF parent.getTagName()<>"FormField" THEN
    RETURN
  END IF
  IF node.getAttribute("shift")="up"
    AND node.getAttribute("width")<=6 THEN
    CALL node.setAttribute("sample","M")
  ELSE
    CALL node.setAttribute("sample","0")
```

```
  END IF
  DISPLAY "set sample attribute of ",node.getId()," to \"",
    node.getAttribute("sample"),"\""
END FUNCTION
```

Form File:

```
LAYOUT(text="sampletest2")
GRID
{
 <G sampletest
   >
  3 Letter Code: [a  ] 3 digit code:[b  ]  Description:[longstring ]

 <G "What can be seen"
   >
  There is no default sample set in this form, but due to a
  ui.form.setDefaultInitializer function, small UPSHIFT fields
  are adjusted to a sample of "M", all other fields get the sample "0"

 ------------------------------------------------------------------------------
  1. The 3 letter code should show up exactly "MMM" because of the applied
     sample="M"
  2. The 3 letter digit code should show up exactly "123" without additional

     spacing
}
END
END
ATTRIBUTES
EDIT a=formonly.three_char_upshift,UPSHIFT,default="MMM";
EDIT b=formonly.three_digit_number,default="123";
EDIT longstring=formonly.longstring,UPSHIFT,
     default="DESCRIPTION OF THE ITEM",SCROLL;
END
```

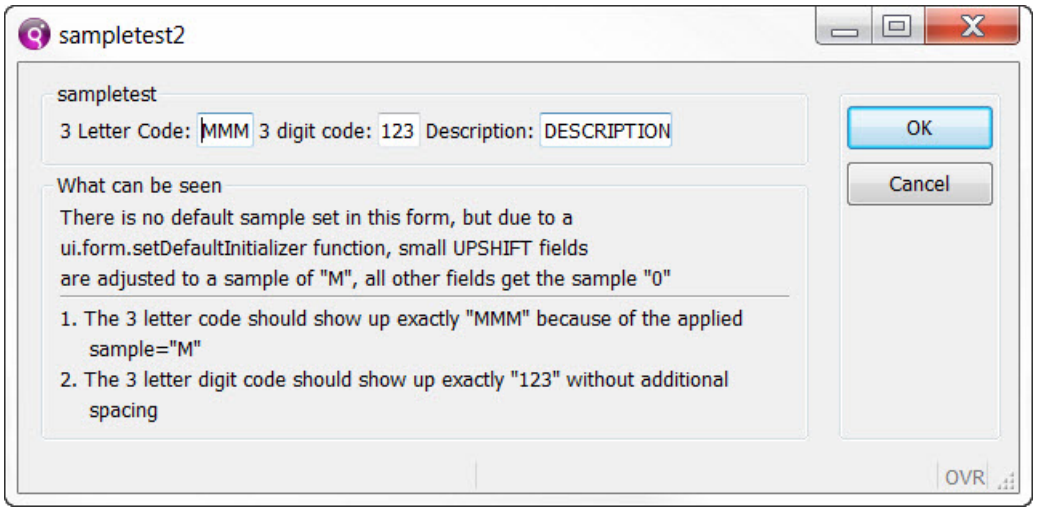


Figure 9: Sample usage in form

Size policy for ComboBoxes
You can use the SIZEPOLICY attribute for a COMBOBOX.

Starting with version 1.30 you can use the SIZEPOLICY attribute for COMBOBOXes.

COMBOBOX form items had a special behavior in versions prior to 1.30, because they adapted their size to the maximum item of the value list. On one hand, this is very convenient because the programmer doesn't have to find the biggest string in the value list, and to estimate how large it will be on the screen (with proportional fonts the string with the highest number of characters is not automatically the largest string). On the other hand, this behavior often led to an unpredictable layout if the programmer didn't reserve enough space for the COMBOBOX.

The SIZEPOLICY attribute gives better control of the result.

```
<G "Combo makes edit2 too big"                    >
 [edit1]
 [combo   ]
 [edit2   ]
...
ATTRIBUTES
EDIT edit1=formonly.edit1;
COMBOBOX combo=formonly.combo,
   ITEMS=((0,"Veeeeeeeery Looooooooooooooooong Item"),(1,"hallo")),
   DEFAULT=0;
EDIT edit2=formonly.edit2;
END
```

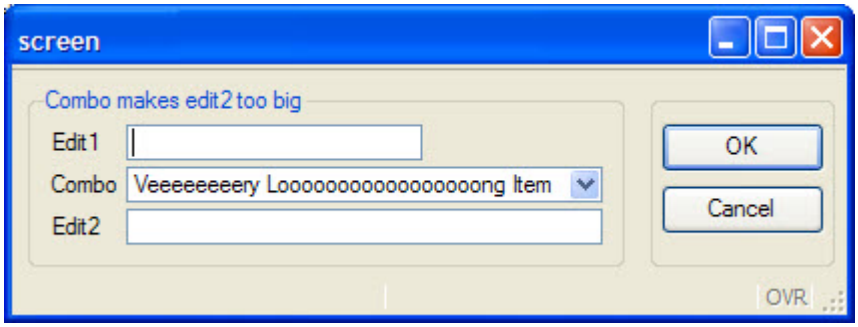


Figure 10: Use of SIZEPOLICY

In this case, the "combo" field gets very large as does "edit2", because it ends in the same grid column. It will confuse the end user if he can input only eight characters and the field is apparently much bigger. Two possibilities exist to surround this:

Use an HBox to prevent the edit2 from growing, and use HBoxes for all fields which start together with combo and are as large or bigger than combo:

```
<G "Edit2 in HBox doesn't grow" >
[edit1]
[combo  :]
[edit2  :]
...
```

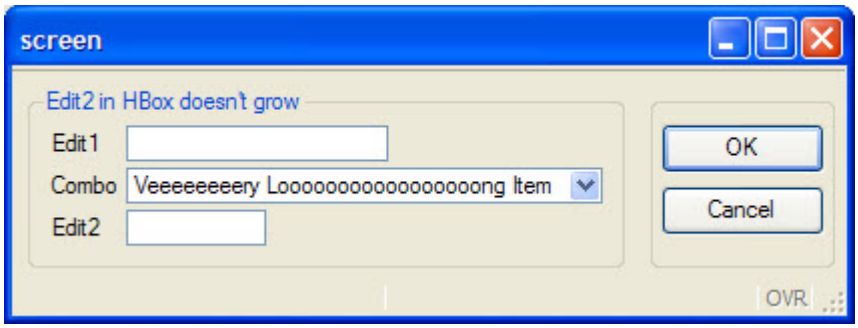


Figure 11: Use of HBox

Use the new `SIZEPOLICY` attribute, and set it to `fixed` to prevent `combo` from getting bigger than the initial six characters (6+Button):

```
 <G "Combo has a fixed size">
 ...
 [combo   ]
 [edit2   ]
 ...
 ATTRIBUTES
 ...
 COMBOBOX combo=formonly.combo,
    ITEMS = ((0,"Veeeeeeeery Looooooooooooooooong Item"),(1,"hallo")),
    DEFAULT=0, SIZEPOLOCY=FIXED ;
....
```

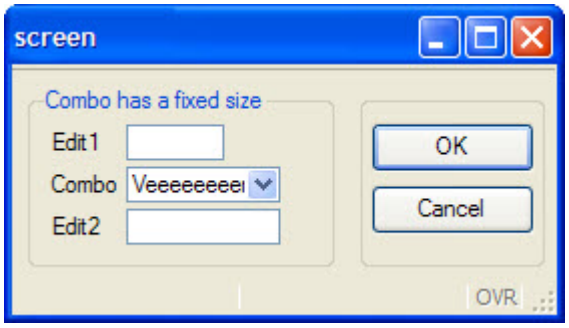


Figure 12: Use of SIZEPOLICY as fixed

In this example the `edit2` dictates the maximum size of `combo`, because even if the SIZEPOLICY is *fixed*, the elements are aligned by the Grid.

To prevent this and have *exactly* six characters (numbers) in the ComboBox, you need to de-couple `combo` from `edit2` by using an HBox.

```
<G "Combo has a fixed size,sample 0,in HBox"
...
Combo [combo  :]
Edit2 [edit2  :]
...
COMBOBOX combo=formonly.combo,
   ITEMS = ((0,"12345678 Looooooooooooooooong Item"),(1,"hallo")),
   DEFAULT=0, SIZEPOLICY=FIXED, SAMPLE="0";
```

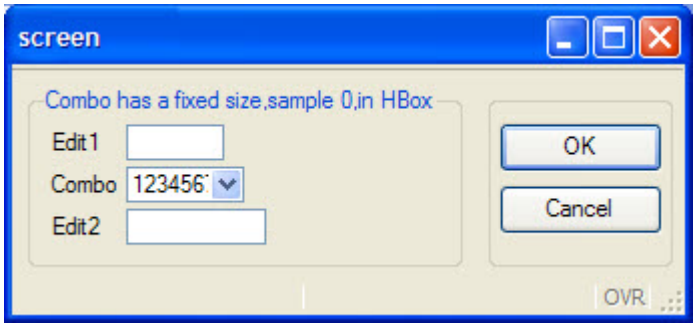


Figure 13: Use of HBox

Now the wanted six numbers are displayed and `combo` does not grow to the size of `edit2`.

Action defaults at form level
You can define action defaults in forms.

Starting with version 1.30 it is now possible to define action defaults in forms. In previous versions you had to define a global action default file; this works for defining common global action attributes, but there is a need to define specific action attributes in some forms. A typical zoom window may have search and navigation actions, while data input windows need to define add/delete/update actions instead.

It is now possible to define an action default section in the form file, and you can also load action defaults with ui.Form.loadActionDefaults().

Compiled string files have now .42s extension
Starting with version 1.30, compiled localized string files use now the file extension `.42s`.

Before version 1.30, the file extension was `.4ls`.

See Localized strings on page 327.

Planned desupport

The features described in this topic will be deprecated or de-supported in the next major release of the product.

Consider reviewing your code now, if you are using one of these features.

Features that will be deprecated in next versions

- Microsoft SQL Server 2005

Features that will be de-supported in next versions

- The `FIELD` form item type, that could be used to specify abstract form fields with attributes defined in the `.val` schema file.
- The GDC WinCOM, WinDDE and WinMail front-call modules.

Migrating from IBM® Informix® 4gl to Genero BDL

- Introduction to I4GL migration on page 140
  - IBM Informix 4GL and Genero BDL products on page 140
  - IBM Informix 4GL reference version on page 140
- Installation and setup topics on page 140
  - Using C extensions on page 140
  - Localization support in Genero on page 141
  - Database schema extractor on page 141
  - Compiling 4GL to C on page 141
- User interface topics on page 141
  - Easy user interface migration with traditional mode on page 141
  - SCREEN versus LAYOUT on page 141
  - Migrating screen arrays to tables on page 143
  - Review TUI specifics on page 144
  - The default SCREEN window on page 145
  - Specifying WINDOW position and size on page 145
  - Right justified field labels on page 145
  - Using widgets instead of multiple text screens on page 146

- Review application ergonomics on page 146
- Subscripted form fields are not supported on page 146
- 4GL programming topics on page 147
  - Dynamic arrays on page 147
  - Debugger command syntax on page 147
  - Mismatching global variable definitions on page 148
  - Strict function signature checking on page 148
  - STRING versus CHAR/VARCHAR on page 150
  - Review user-made C routines on page 150
  - Web Services support on page 150
  - File I/O statements and APIs on page 150
  - OPEN USING followed by FOREACH on page 151

Introduction to I4GL migration

IBM® Informix® 4GL and Genero BDL products

IBM® Informix® 4GL (I4GL) and Genero Business Development Language (BDL) are distinct development tools. The purpose of Genero BDL is to be as compatible as possible with I4GL, and it is very close. The success of Genero BDL depends on the ability to compile and run legacy 4gl code with minimum code changes. For text-mode applications, the migration steps are often reduced to recompile-and-run.

Genero BDL extends the I4GL language with advanced features, such as a Graphical User Interface and SQL access to non-Informix databases. This leads to some differences that you have to deal with, but these incompatibilities are minor compared to the added value.

In some rare cases, the Genero BDL team decided to take a different path to implement an I4GL feature, because we considered that the IBM® Informix® 4gl solution was not adaptable. For example, the dynamic arrays in I4GL and Genero BDL have different semantics.

This guide will help you identify the differences and find solutions to make the migration from IBM® Informix® 4gl easier.

IBM® Informix® 4GL reference version

Several versions of the IBM® Informix® 4GL language have been released. It started in the mid-80s with I4GL version 4.x; then came version 6.x in 1996. I4GL version 7.2 was released in 1998; then versions 7.31, 7.32, and finally the version: 7.50 came out.

There have been several bug fixes and enhancements over the life of I4GL, resulting in releases that slightly differ. Supporting strict compatibility with all versions of I4GL is not possible for Genero BDL.

The Genero BDL compatibility level with IBM® Informix® 4gl is achieved by comparing with the latest version of I4GL, which is version **7.50** at the time of this writing.

Installation and setup topics

Using C extensions

With IBM® Informix® 4GL, you can extend the **fglgo** runtime executable or link your binary programs with **c4gl** by adding your own C functions.

When migrating to Genero Business Development Language, the C-Extensions must be reviewed in order to provide them as shared libraries. Normally, C extensions modules must be specified in .4gl modules with the IMPORT instruction. To simplify migration, the runtime system loads the **userextension** shared library (or DLL) automatically, so you can group all your existing C functions in a unique shared library and use it without changing the source code of your programs.

Localization support in Genero

To support language-specific and country-specific locales, as well as multibyte character sets like BIG5, IBM® Informix® 4GL uses the Informix® GLS library.

For locale support, Genero Business Development Language (BDL) does not use the Informix® GLS library, to be independent from Informix® GLS libraries. Genero uses the standard C library functions for character data hanlding, based on the POSIX setlocale() function.

I4GL uses the CLIENT_LOCALE environment variable to define the locale for the application. With Genero BDL, you must use the LANG/LC_ALL environment variables to specify the locale of the application. Note, however, that CLIENT_LOCALE is still needed to define the locale for the IBM® Informix® database client.

Database schema extractor

Before compiling .4gl or .per files, you must extract the database schema with the **fgldbsch** tool. This will produce an .sch file, and optionally, .val and .att files. The fgldbsch tool can extract database schemas from Informix®, and from other databases such as Oracle and SQL Server, but you must be aware of data type conversion rules.

Compiling 4GL to C

The IBM® Informix® 4GL compilers include a p-code based runtime system called RDS as well as a C-compiled solution, the c4gl compiler. The RDS solution is typically used in a development environment, supporting a debugger, while the Informix® 4GL C compiler is traditionally used to maximize performance on production sites. However, the C compiled binaries need to be built on the same target platform as the production system.

Genero Business Development Language supports a p-code architecture, which is as fast as the C-compiled version of IBM® Informix® 4GL. Since p-code files are portable, you can develop your application on a platform that is different from the production platform, saving porting procedures and simplifying deployment tasks.

User interface topics

Easy user interface migration with traditional mode

IBM® Informix® 4GL (I4GL) and Genero Business Development Language (BDL) handle windows and form content rendering differently. I4GL is designed to write applications for for dumb terminals, while Genero BDL uses real GUI rendering, with resizeable windows and proportional fonts. To simplify migration from TUI-style products, Genero BDL supports the traditional GUI mode.

SCREEN versus LAYOUT

To design a form with IBM® Informix® 4GL, you organize labels and fields in the `SCREEN` section of a **.per** form file. Genero Business Development Language introduced a new `LAYOUT` section to place form elements. The new `LAYOUT` section allows more sophisticated form design than `SCREEN`.

When writing new programs for GUI applications, you should use a `LAYOUT` section instead of `SCREEN`. However, the `SCREEN` section is still supported to be used to design TUI mode forms.

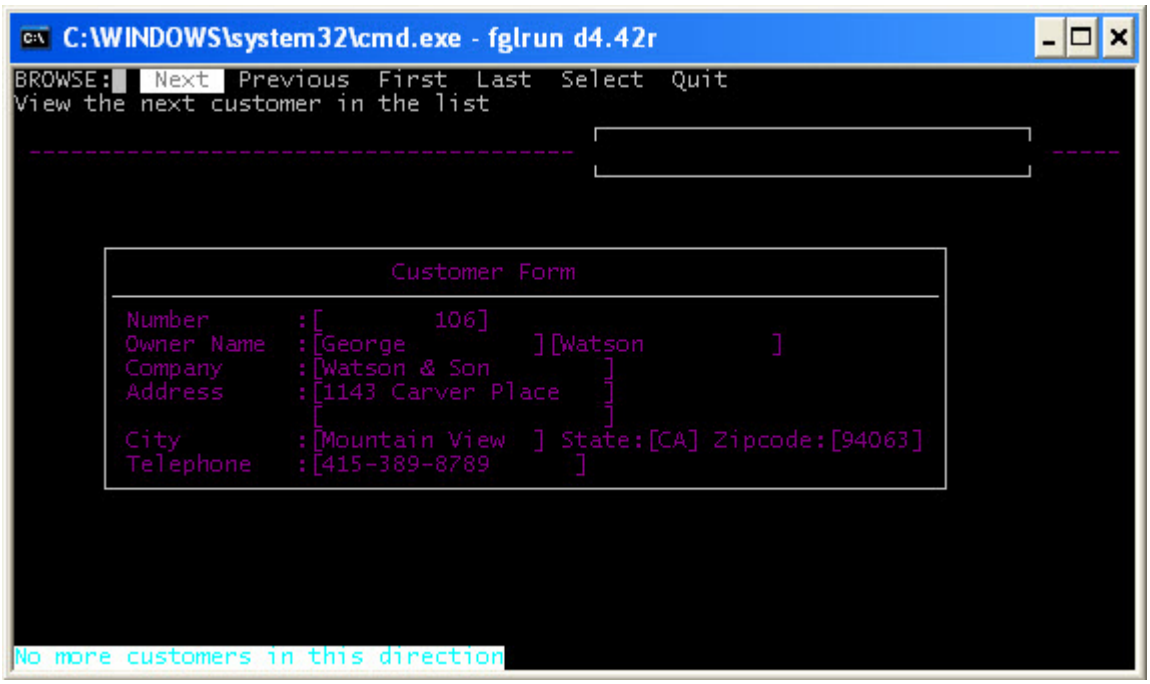


Figure 14: Form using a SCREEN section in TUI mode

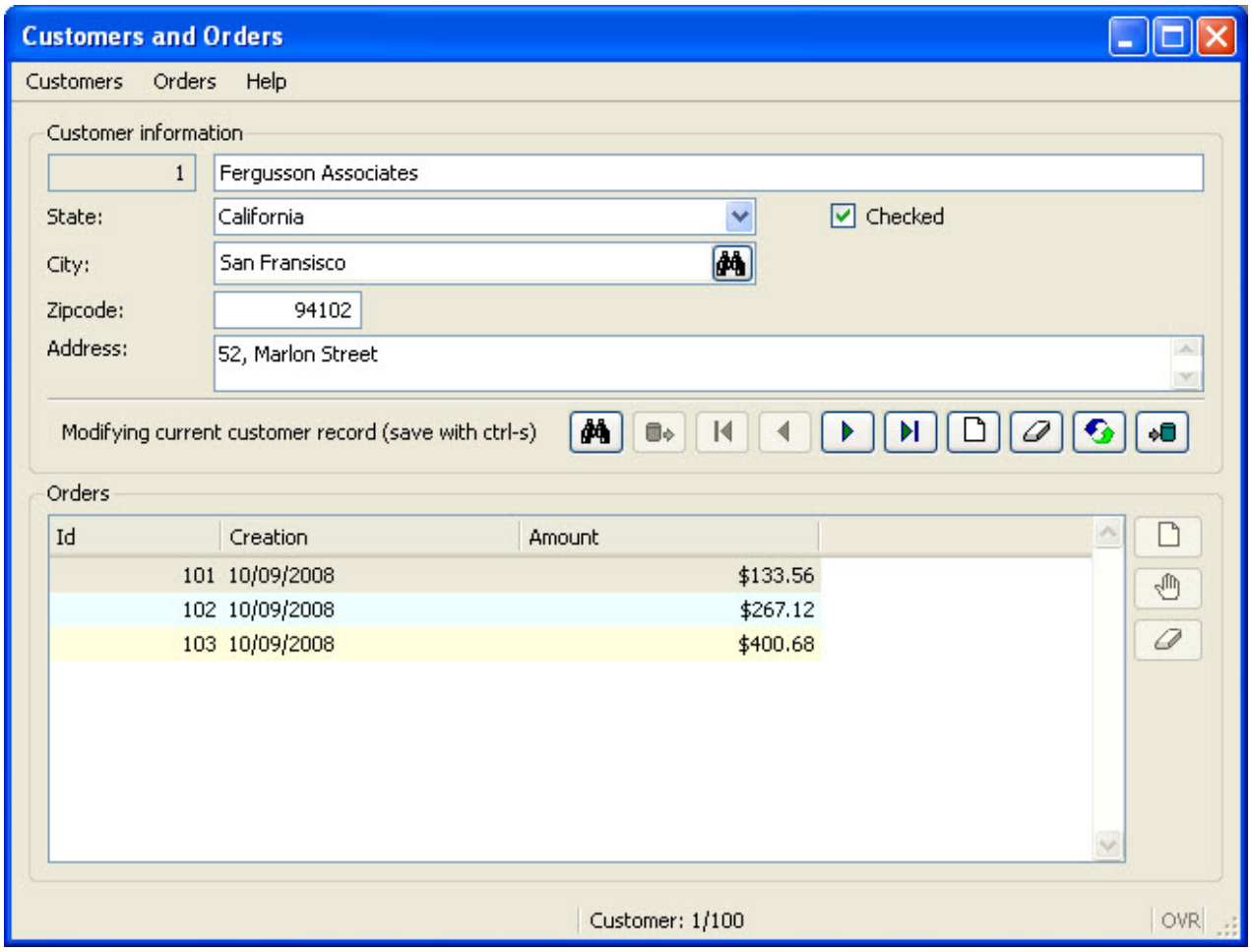


Figure 15: Form using a LAYOUT section in GUI mode

Migrating screen arrays to tables

With IBM® Informix® 4GL, a list of records can be displayed on the screen by using a static screen array in the SCREEN section of the form specification file, with a finite number of lines:

```
DATABASE stores
SCREEN
{
 Id       First name   Last name
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
}
END
TABLES
  customer
END
ATTRIBUTES
  f001 = customer.customer_num ;
  f002 = customer.fname ;
  f003 = customer.lname ;
END
INSTRUCTIONS
  SCREEN RECORD sr_cust[6]( customer.* );
END
```

The display of the form specification file in GUI mode:

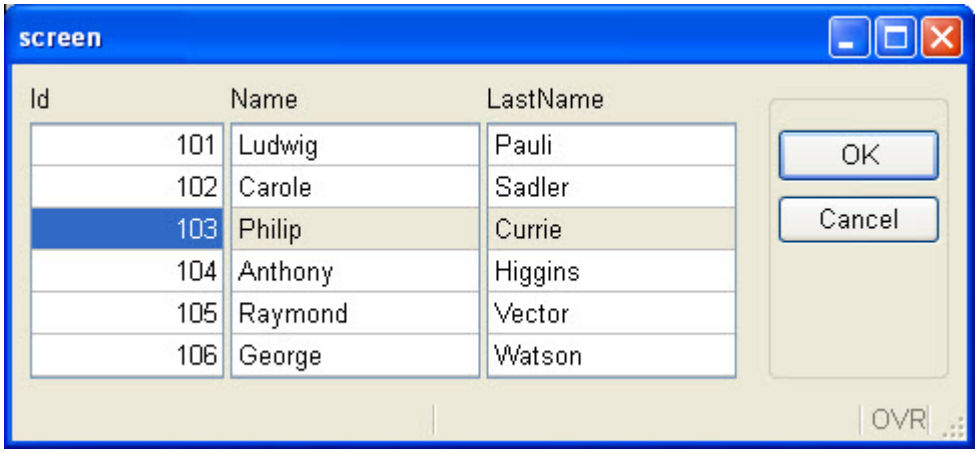


Figure 16: Form displayed not using table widget

With Genero Business Development Language, use a static screen array for applications displayed in dumb terminals, and for GUI applications you can for example use the TABLE container:

```
DATABASE stores
LAYOUT
TABLE
{
 Id       First name   Last name
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
```

```
[f001    |f002        |f003        ]
}
END
END
TABLES
 customer
END
ATTRIBUTES
 f001 = customer.customer_num ;
 f002 = customer.fname ;
 f003 = customer.lname ;
END
INSTRUCTIONS
 SCREEN RECORD sr_cust( customer.* );
END
```

The display of the form specification file is a real table widget, which is resizeable. The .4gl source is untouched.

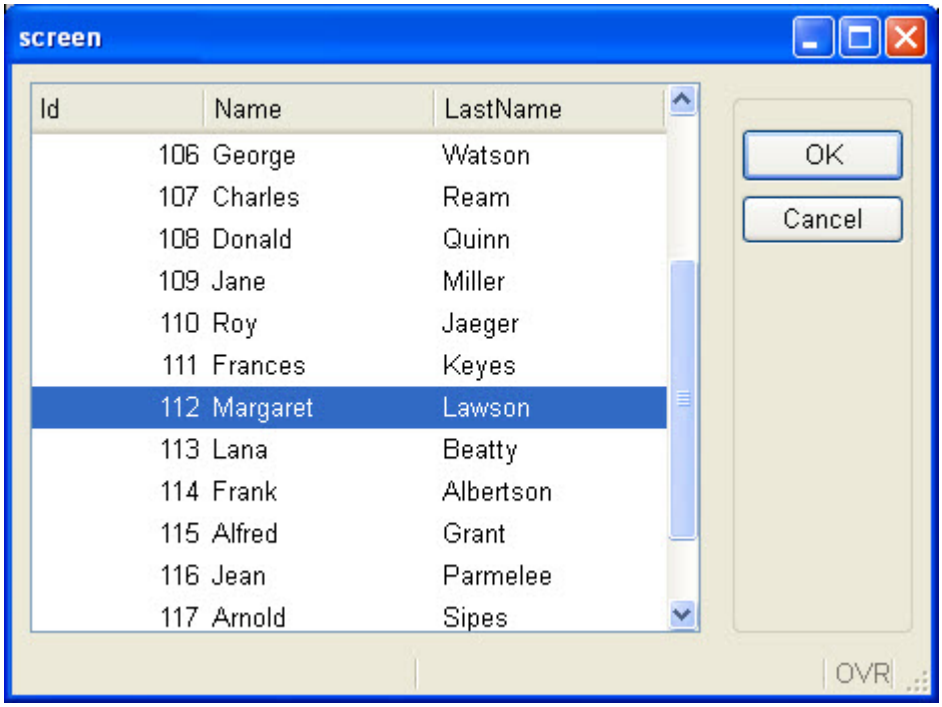


Figure 17: Form displayed as table widget

Review TUI specifics

Typical IBM® Informix® 4GL programs use the TUI mode and often exploit all the display possibilities of the language for dumb terminals. Some instructions are specific to TUI mode and should be reviewed when redesigning the application for GUI mode.

For example, data records can be displayed in a screen array with a `DISPLAY`*`array`*`[`*`array-index`*`].* TO `*`screen-array`*`[`*`screen-line`*`]` instruction, optionally with the `ATTRIBUTES()` clause to use some TTY attributes like colors, reverse and bold effects. When scrolling a list, I4GL actually uses the terminal scrolling capabilities to preserve the TTY attributes in each row. This applies only to the current rows visible on the screen, but it was a commonly used feature.

In order to display application screens on different types of front-ends, Genero Business Development Language (BDL) handles user interface elements in a more abstract way. Therefore, dumb terminal specifics as described above cannot be supported. A good replacement for `DISPLAY ... TO ... ATTRIBUTES()` in `DISPLAY ARRAY` or `INPUT ARRAY` is to use the DIALOG.setArrayAttributes() method.

Genero BDL supports TUI-specific instructions such as `DISPLAT AT`, `CLEAR SCREEN`, `CLEAR WINDOW`, as well as TTY attributes such as BLUE, RED, REVERSE, but you should use those instructions for TUI programs only. New GUI programs should use graphical user interface possibilities. For example, a good replacement for TTY attributes is to use presentation styles.

The default SCREEN window

When the first interactive instruction is reached in a Genero BDL program, a default window named `SCREEN` is created.

The default `SCREEN` window can be used to open one or more successive forms; it can also be closed, with the `CLOSE WINDOW SCREEN` instruction. If the default `SCREEN` window is not cloase, and a new window is created with the `OPEN WINDOW` command, an empty default `SCREEN` window will be displayed.

When writing a GUI application, you typically open the main form in the `SCREEN` window, and display other forms with the `OPEN WINDOW` name `WITH FORM` instruction:

```
MAIN
  DEFER INTERRUPT
  OPTIONS INPUT WRAP
   ...
  OPEN FORM f_main FROM "custfrm"
  DISPLAY FORM f_main
   ...
END MAIN
```

The `SCREEN` window is not visible in TUI mode because program windows are rendered as simple boxes and `SCREEN` is created without borders. The size of the `SCREEN` window is 80x25 in TUI mode.

Specifying WINDOW position and size

When writing a program for TUI mode, the windows can be created with the `OPEN WINDOW name AT x,y` instruction, specifying an position on the screen; sometimes even the width and height of the window is specified, for example when you don't use a form to create the window. Window position and size is allowed by Genero Business Development Language for TUI mode applications.

However, the window position and sizes are ignored in GUI mode. In GUI mode, the window position is defined by the window manager, and the size adapts to the form displayed. In this mode, the preferred way to display application forms is to use the `OPEN WINDOW` name `WITH FORM` instruction.

Right justified field labels

If the application forms define right-justified labels and use a proportional font in GUI mode, the text will no longer be aligned as on a dumb terminal. Form layout must be reviewed to replace any right-justified text with `LABEL` form items. Migration to GUI mode can also be easier achieved with the traditional mode, to leave TUI-style forms untouched.

Example of right-justified static form labels

```
DATABASE FORMONLY
SCREEN
{
     Customer id: [f01       ]
           Name: [f02                          ]
         Zipcode: [f03     ]
         Address: [f04                                 ]
}
END
ATTRIBUTES
EDIT f01 = FORMONLY.cust_id;
EDIT f02 = FORMONLY.cust_name;
EDIT f03 = FORMONLY.cust_zipcode;
```

```
EDIT f04 = FORMONLY.cust_address;
END
```

Example of form label items with localized text

```
LAYOUT
GRID
{
     [l01         |f01       ]
     [l02         |f02                          ]
     [l03         |f03     ]
     [l04         |f04                                  ]
}
END
END
ATTRIBUTES
LABEL l01: TEXT=%"customer.id";
LABEL l02: TEXT=%"customer.name";
LABEL l03: TEXT=%"customer.zipcode";
LABEL l04: TEXT=%"customer.address";
EDIT f01 = FORMONLY.cust_id;
EDIT f02 = FORMONLY.cust_name;
EDIT f03 = FORMONLY.cust_zipcode;
EDIT f04 = FORMONLY.cust_address;
END
```

Using widgets instead of multiple text screens

Applications designed for dumb terminals (TUI mode) use various techniques to ensure that all display fits in an 80x25 screen. This may mean iterating through a number of dialogs using different forms, only displaying certain columns in an record list, using abbreviations for labels, etc.

With a graphical user interface (GUI mode), windows are wider, re-sizable and can contain different sort of layout elements and widgets, displaying much more information as in a simple dumb terminal. For example, TABLE containers display record lists and have the ability to scroll horizontally so that you can show more than 78 characters of data.

Dump-terminal oriented programs should be reviewed to take advantage of the GUI possibilities. However, do not end up with over-crowded screens, that will be unreadable to the end user.

Review application ergonomics

With IBM® Informix® 4GL, programs can only execute a single `MENU`, `INPUT`, `CONSTRUCT`, `DISPLAY ARRAY` or `INPUT ARRAY` instruction at a time. This may be sufficient for dumb-terminal applications, but is not adapted for a graphical user interface.

Genero Business Development Language (BDL) introduces the concept of multi-dialog, where multiple interactive instructions control several form areas at the same time. Typical GUI concepts such as Drag and Drop and Tree Views are supported as well. You may wish to review your code to take advantage of these features.

Subscripted form fields are not supported

IBM® Informix® 4GL forms can define subscripted fields with multiple field definition entries in the `ATTRIBUTES` section, each defining a piece of the data displayed by the field, as in this example:

```
DATABASE stores
SCREEN
{
   1234567890
  [f01       ]
```

```
  [f02       ]
}
END
ATTRIBUTES
f01 = customer.cust_name[1,10];
f02 = customer.cust_name[11,20];
END
```

In the ATTRIBUTES section, the name of the field is immediately followed by a subscript specification defining the piece of sub-data the screen tag displays and allows to input.

This feature is not supported at all by Genero BDL, all fields must be defined as a whole.

4GL programming topics

Dynamic arrays

Both IBM® Informix® 4GL (I4GL) and Genero Business Development Language (BDL) implement static arrays with a fixed size. Static arrays cannot be extended:

```
DEFINE arr ARRAY[100] OF RECORD LIKE customer.*
```

I4GL introduced dynamic arrays in version **7.32**. Unlike Genero BDL, I4GL requires explicitly to associate memory storage with a dynamic array by using the ALLOCATE ARRAY statement, and memory must be freed with DEALLOCATE ARRAY. I4GL dynamica arrays can be resized with the RESIZE ARRAY statement. I4GL dynamic arrays cannot be used in a interactive instructions such as DISPLAY ARRAY.

```
DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
ALLOCATE ARRAY arr[10]
RESIZE ARRAY arr[100]
LET arr[50].cust_name = "Smith"
DEALLOCATE ARRAY arr
```

Genero BDL supports dynamic arrays in a slightly different way than I4GL. There are no allocation, resizing, or deallocation instructions, because the memory for element storage is automatically allocated when needed. Further, you can use dynamic arrays with interactive instructions, making a DISPLAY ARRAY or INPUT ARRAY unlimited.

```
DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
LET arr[50].cust_name = "Smith"
DISPLAY ARRAY arr TO sr.*
```

In Genero BDL, the main difference between static arrays and dynamic arrays is the memory usage; when you use dynamic arrays, elements are allocated on demand. With static arrays, memory is allocated for the complete array when the variable is created.

> **Important: The semantics of dynamic arrays is very similar to static arrays, but there are some small differences. Keep in mind that the runtime system automatically allocates a new element for a dynamic array when needed. For example, when a `DISPLAY arr[100].*` is executed with a dynamic array, the element at index 100 is automatically created if does not exist.**

Debugger command syntax

IBM® Informix® 4GL (I4GL) provides a program debugger. Genero Business Development Language provides a program debugger with a different set of commands as I4GL, compatible with the well-known gdb tool. This debugger can be used alone in command line mode, or with a graphical shell compatible with gdb, such as **ddd**:

```
ddd --debugger "fglrun -d myprog"
```

Mismatching global variable definitions

The c4gl C-code compiler of IBM® Informix® 4GL has a weakness that allows global variable declarations of the same variable with different data types. Each different declaration found by the c4gl compiler defines a distinct global variable, which can be used separately. This can actually be very confusing (the same global variable name can, for example, reference a `DATE` value in module A and an `INTEGER` value in module B).

IBM® Informix® 4GL RDS (fglpc / fglgo) does not allow multiple global variable declaration with different types. The fglgo runner raises error -1337 if this happens.

The next code example shows two .4gl modules defining the same global variable with different data types:

Main.4gl:

```
GLOBALS
  DEFINE v INTEGER
END GLOBALS
...
MAIN
  ...
  LET v = 123
  ...
END MAIN
```

Module.4gl:

```
GLOBALS
  DEFINE v DATE
END GLOBALS
...
FUNCTION test()
  ...
  LET v = TODAY
  ...
END FUNCTION
```

The fglcomp tool compiles both modules separately without problem, but when linking with fgllink, the linker raises error -1337.

You must review your code and use the same data type for all global variables having the same name.

Strict function signature checking

IBM® Informix® 4GL (I4GL) is not very strict regarding function signature. With I4GL, you can, for example, define a function in module A that returns three values, and call that function in module B with a returning clause specifying two variables:

Module A:

```
FUNCTION func()
   RETURN "abc", "def", "ghi"
END FUNCTION
```

Module B (main):

```
MAIN
   DEFINE v1, v2 VARCHAR(100)
   CALL func() RETURNING v1, v2
END MAIN
```

The c4gl compiler (7.32) compiles and links these modules without error, but at execution time you get the following runtime error:

```
Program stopped at  "main.4gl", line number 3.
FORMS statement error number -1320.
A function has not returned the correct number of values
expected by the calling function.
```

With Genero Business Development Language (BDL), the mistake will be detected at link time:

```
$ fgllink -o prog.42x main.42m module_a.42m
ERROR(-6200): Module 'main': The function module_a.func(0,3) will be
called as func(0,2).
```

Similarly, I4GL does not detect an invalid number of parameters passed to a function defined in a different module:

Module A:

```
FUNCTION func( p )
   DEFINE p INTEGER
   DISPLAY p
END FUNCTION
```

Module B (main):

```
MAIN
   CALL func(1,2)
END MAIN
```

The c4gl compiler (7.32) compiles and links these modules without error, but at execution time, you get the following runtime error:

```
Program stopped at  "main.4gl", line number 2.
FORMS statement error number -1318.
A parameter count mismatch has occurred between the calling
function and the called function.
```

When using Genero BDL, the error will be detected at link time:

```
$ fgllink -o prog.42x main.42m module_a.42m
ERROR(-6200): Module 'main': The function module_a.func(1,0) will be
called as func(2,0).
```

Note, however, that Genero BDL does not check function signatures when several RETURN instructions are found by the compiler. This is necessary in order to be compatible with I4GL. The next code example compiles and runs with both I4GL and BDL:

```
MAIN
   DEFINE v1, v2 VARCHAR(100)
   CALL func(1) RETURNING v1
   DISPLAY v1
   CALL func(2) RETURNING v1, v2
   DISPLAY v1, v2
END MAIN

FUNCTION func( n )
   DEFINE n INTEGER
   IF n == 1 THEN
      RETURN "abc"
   ELSE
```

```
      RETURN "abc", "def"
   END IF
END FUNCTION
```

However, this type of programming is not recommended.

STRING versus CHAR/VARCHAR

Genero Business Development Language (BDL) introduces a new data type named `STRING`, which is similar to `VARCHAR`, but without a size limit. The `STRING` data type does not exist in IBM® Informix® 4GL. The `STRING` data type implementation is optimized for memory usage; unlike `CHAR/VARCHAR`, BDL will only allocate the memory needed to hold the actual character string value in a `STRING` variable.

A `STRING` variable is typically used within utility functions (for example, to hold the path to a file). Another typical usage is with CONSTRUCT, to hold the SQL condition. The `STRING` variable can then be completed to build the SQL text and passed to the PREPARE or DECLARE instruction.

However, because of SQL assignment and comparison rules, the `STRING` variables cannot be used as SQL parameters in the `USING` clause of EXECUTE or OPEN/ FOREACH, not can it be used to receive fetched data with the FETCH instruction: For SQL statements, use CHAR or VARCHAR data types.

The `STRING` data type has a number of built-in methods e.g. getLength() that will are very useful and will reduce source code.

Review user-made C routines

IBM® Informix® 4GL (I4GL) applications often need additional utility C routines implemented in C-Extensions, for example to access the file system and read the content of a directory. Writing C-Extensions is an important cost in cross-platform portability and maintenance.

Genero Business Development Language (BDL) provides a set of utility libraries that include functions and classes which can probably replace some of the routines written for I4GL application. For example, BDL implements typical file management functions to search directories and files.

If portability is a concern (for example if you want to move from a UNIX™ platform to a Microsoft™ Windows™ or Mac OS-X™ platform), review your C routines and check whether there is a replacement built into the language or in one of the libraries provided.

Genero BDL even allows to use to the huge Java™ class library with the Java™ Interface .

Web Services support

Starting with IBM® Informix® 4GL version **7.50**, I4GL functions can be deployed as Web Services. The published functions can be subscribed from programs that run on a Web client in another programming language.

Web Services support was introduced in Genero Business Development Language before I4GL 7.50 was released. Each implementation is quite different, but the basic principles are the same: publishing 4gl functions as Web Services, by handling WS requests and supporting easy input and output parameter conversions between WS data formats and 4gl program variables.

File I/O statements and APIs

Enhancement reference: BZ#19156

IBM® Informix® 4GL version **7.50.xC4** introduced file manipulation instructions to access files on the operating system running the application. These instructions can be used to open, read, write, seek and close files:

```
MAIN
   DEFINE fd1, fd2 INTEGER, v1,v2 VARCHAR(10)
   OPEN FILE fd1 FROM "/tmp/file1" OPTIONS (READ, FORMAT="CVS")
```

```
   OPEN FILE fd2 FROM "/tmp/file2" OPTIONS (WRITE, APPEND, CREATE,
 FORMAT="CVS")
   READ FROM fd1INTO v1, v2
   SEEK ON fd2TO 0FROM LAST INTO v1
   WRITE TO fd2USING v1, v2
   CLOSE FILE fd1
   CLOSE FILE fd2
END MAIN
```

Genero Business Development Language (BDL) implements file I/O support with the *base.Channel* built-in class. This class implements file access, but it can also open streams to subprocesses (i.e. pipes) and sockets.

OPEN USING followed by FOREACH

In earlier versions of IBM® Informix® 4GL (I4GL), the `FOREACH` instruction had no a `USING` clause to pass SQL parameters to the prepared statement. SQL Parameters could be specified in a `OPEN USING` instruction, and were re-used by the next `FOREACH` instruction:

```
PREPARE st1 FROM "SELECT * FROM tab WHERE col>?"
DECLARE cu1 CURSOR FOR st1
OPEN cu1 USING var
FOREACH cu1 INTO rec.*
  DISPLAY rec.*
END FOREACH
```

This feature is supported by Genero Business Development Language, but can lead to defects with some versions of the Informix® database client. Review your code to avoid the `OPEN` statement by moving the `USING` clause to the `FOREACH` instruction.

Migrating from Four Js BDS to Genero BDL

These topics describe product changes you must be aware of when migrating from Four Js BDS 3.xx to the most recent Genero Business Development Language version.

- Installation and setup topics on page 152
  - License controller on page 152
  - Runner linking is no longer needed on page 152
  - Localization support on page 152
  - Database schema extractor on page 152
  - C-Code compilation is desupported on page 153
  - Desupported environment variables on page 153
  - Desupported FGLPROFILE entries on page 153
- User interface topics on page 153
  - Easy user interface migration with traditional mode on page 153
  - Front-end compatibility on page 154
  - FGLGUI is 1 by default on page 154
  - FGLPROFILE: GUI configuration on page 154
  - Key labels versus action defaults on page 158
  - Migrating form field widgets on page 159
  - SCREEN versus LAYOUT on page 160
  - Migrating screen arrays to tables on page 160
  - Review TUI specifics on page 160
  - The default SCREEN window on page 160
  - Specifying WINDOW position and size on page 160

  - Front-end configuration tools on page 160
  - Function key mapping on page 161
- 4GL Programming topics on page 161
  - FGLPROFILE: VM configuration on page 161
  - Calling fgl_init4gl() initialization function on page 162
  - Static versus Dynamic Arrays on page 162
  - Debugger syntax changed on page 162
  - fgl_system() function on page 162
  - The Channel:: methods on page 162
  - STRING versus CHAR/VARCHAR on page 162
  - Review user-made C routines on page 162
  - Strict variable identification in SQL statements on page 162
  - Default action of WHENEVER ANY ERROR on page 162

Installation and setup topics

License controller

With Four Js Business Development Suite (BDS), you must license the product with the `licencef4gl` command line tool. Starting with Genero Business Development Language, the command line tool to license the product is `fglWrt`. Run fglWrt with the -h option for the possible options.

Runner linking is no longer needed

With Four Js Business Development Suite (BDS), you need to create the fglrun binary with the fglmkrun tool, by specifying the type of the database driver and C extensions libraries to be linked with the runtime system. Since Genero Business Development Language version 2.00, you do not more need to link the runtime system.

The database drivers are provided as shared libraries ready to use; you just need to specify the driver to be loaded.

However, C extensions must be provided shared libraries for Genero BDL. To easy migration, the runtime system loads automatically the `userextension` share library (or DLL).

Localization support

IBM® Informix® 4GL (I4GL) and Four Js Business Development Suite (BDS) use the Informix® GLS library for localization support (i.e. to support non-ASCII character sets such as BIG5). This implies a strong dependency to the proprietary GLS library.

Genero Business Development Language (BDL) does not use the GLS library; Genero BDL uses the standard C library functions for character set handling, based on the setlocale() POSIX conformant function.

While I4GL/BDS need the CLIENT_LOCALE environment variable to define the locale for the application, you must now use the LANG/LC_ALL environment variables to specify the locale of the Genero application. Note, however, that CLIENT_LOCALE is still needed when connecting to an Informix® database.

In Four Js BDS, you could select the locale library with the fglmode tool, to select either GLS or ASCII mode. This tool is no longer needed in Genero.

Database schema extractor

Before compiling .4gl or .per files with Four Js Business Development Suite (BDS) or with Genero Business Development Language (BDS), you need to extract the database schema as a .sch file.

In DBS, the name of the schema extraction tool fglschema, while Genero BDL provides the fgldbsch tool. The fglschema tool could only extract schemas from Informix® databases; fgldbsch can extract database schemas from Informix®, and from other databases like Oracle, SQL Server, DB2®, PostgreSQL, MySQL and Genero db. The fglschema tool is still supported in Genero BDL for backward compatibility, but fglschema actually calls fgldbsch.

Note that Genero BDL allows you to centralize new widget types and attributes in the .val file.

C-Code compilation is desupported

Four Js Business Development Suite (BDS) fglcomp could compile to P-Code or C-Code. The compiler of Genero Business Development Language does not support C-Code generation. Only P-Code generation is supported by Genero BDL.

If you experience performance problems when comparing Genero BDL to Four Js BDS, please contact your local support center.

Desupported environment variables

This table lists the Four Js Business Development Suite (BDS) environment variables that are no longer supported (or replaced) in Genero Business Development Language:

Table 87: Desupported environment variables

Entry	Description of the BDS environment variable	Genero equivalent
FGLDBS	FGLDBS defines the type and version of the database driver, used when linking fglrun with fglmkrun.	Database drivers are loaded dynamically by fglrun.
FGLCC	FGLCC defines the name of the C compiler.	The fglrun tool does not need to be created, it's fully dynamic.
FGLLIBSQL	FGLLIBSQL defines the list of database client software libraries to be used to link fglrun with fglmkrun.	Database drivers are loaded dynamically by fglrun.
FGLLIBSYS	FGLLIBSYS defines the list of system libraries to be used to link fglrun with fglmkrun.	The fglrun tool does not need to be created, it's fully dynamic.
FGLSHELL	FGLSHELL defined the name of the fglrun program, for example when using tools like fglschema.	The name of the runtime system tool is fglrun and does not need to be changed.

Desupported FGLPROFILE entries

Genero Business Development Language comes with redesigned software components and features. Some FGLPROFILE entries have been desupported. This section describes what configurations settings are no longer supported, and point to Genero equivalent features if they exist.

User interface topics

Easy user interface migration with traditional mode

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Front-end compatibility

When migrating to Genero Business Development Language (BDL), you must use one of the Genero Front Ends; the **WTK**, **WebFE** and **JavaFE** front-ends are not compatible with the Genero fglrun runtime system. Note also that the UNIX™ version of Genero does not include the **fglX11d** front-end any longer. You must use the GDC front-end on UNIX™.

FGLGUI is 1 by default

In Four Js Business Development Suite (BDS), when the FGLGUI environment variable is not set, the application starts in TUI mode (FGLGUI=0). With Genero BDL, the default is GUI mode (FGLGUI=1). Therefore, when migrating from Informix® 4GL, you should set FGLGUI=0 to run the application in text mode as a first step.

FGLPROFILE: GUI configuration

This table shows Four Js Business Development Suite (BDS) FGLPROFILE entries related to GUI configuration which are desupported in Genero Business Development Language.

Table 88: BDS/WTK FGLPROFILE entries related to GUI configuration which are desupported in Genero

Entry	Description of the BDS feature	Genero equivalent
`fglrun.interface`, `fglrun.scriptName`	These entries defined the TCL configuration and script to be send to the WTK front-end.	There is no equivalent in Genero.
`fglrun.guiProtocol.*`	These entries could be used to configure the communication protocol with WTK front-end.	In Genero you can control this with gui.protocol.* entries.
`fglrun.error.line.number`	This entry was used to define the number of lines to be displayed in the error message line.	You can control the aspect of the error line with the Window style attribute called `statusBarType`.
`gui.useOOB.interrupt` `fglrun.signalOOB`	These entries could be used to configure or disable Out Of Band signal on the GUI protocol socket to avoid problems on platforms not supporting that feature. OOB signal was used to send interruption events the program executed is processing.	Genero supports interruption event handling with a predefined action name called **interrupt**. You can bind any sort of action view (button in form, toolbar or topmenu item) with this name. Interrupt events are sent asynchronously with the new Genero GUI protocol and don't use OOB signals any longer. See User interruption handling on page 1252 for more details.
`Sleep.minTime`	This entry was used to define the number of seconds before the interrupt key button appeared on the screen window when the program is processing.	Genero supports interruption event handling with a predefined action name called **interrupt**. You can bind any sort of action view (button in form, toolbar or topmenu item) with this name. Interrupt events are sent asynchronously with the new

Entry	Description of the BDS feature	Genero equivalent
		Genero GUI protocol and don't use OOB signals any longer. See User interruption handling on page 1252 for more details.
`gui.watch.delay`	This entry was used to define the number of seconds before the mouse cursor displays as a wait cursor, when the program is processing.	Genero supports interruption event handling with a predefined action name called **interrupt**. You can bind any sort of action view (button in form, toolbar or topmenu item) with this name. Interrupt events are sent asynchronously with the new Genero GUI protocol and don't use OOB signals any longer. See User interruption handling on page 1252 for more details.
`gui.bubbleHelp.*`	These entries could be used to enable and configure tooltips displaying field COMMENT text.	Genero front-ends display bubble-help with field COMMENT text by default.
`gui.controlFrame.scroll.*`	These entries could be used to show and configure a scrollbar in the control frame displaying ON KEY or COMMAND buttons.	Genero front-ends display control frame scrolling buttons by default when needed. See also Window style attributes like `ringMenuScroll`.
`screen.scroll`	This entry could be used to get scrollbars in the main window when the form was too big for the screen resolution of the workstation.	With Genero, by default, each program window is rendered as a distinct GUI window by the front-end. Window aspect can be controlled with style attributes. See Window style attributes for more details.

`gui.screen.size.x`
`gui.screen.size.y`
`gui.screen.x`
`gui.screen.y`
`gui.screen.incrx`
`gui.screen.incry`
	These entries could be used to configure the size and position of the main screen window with the WTK front-end.	In Genero, each program window is rendered as a distinct GUI window by the front-end. There is no equivalent for these options. However, you can use the traditional mode to render program windows in a single parent screen window and with BDS/WTK.

`gui.screen.withvm`	This entry could be used to integrate with the X11 window manager (allowing move and resize actions).	There is no equivalent in Genero.
`gui.preventClose.message`	This entry could be used to display an error message to the user attempting to close the main GUI	In Genero, each program window is rendered as a distinct GUI window by the front-end. You can use the

Entry	Description of the BDS feature	Genero equivalent
	window with CTRL-F4 or the cross-button on the right of the GUI window title bar.	**close** action to control window close events. See Implementing the close action on page 1337 for more details. See also ON CLOSE APPLICATION program option.
`gui.key.doubleClick.left`	This entry could be used to define the key to be returned to the program when the user double-clicks on the left button of the mouse.	You can use the DOUBLECLICK attribute to define the action to be invoked when the user double-clicks on a Table container.
`gui.key.click.right`	This entry could be used to define the key to be returned to the program when the user clicks on the right button of the mouse.	You can configure contextual menus with the **CONTEXTMENU** attribute in action attributes.
`gui.key.add_function`	Could be used to define the offset to identify SHIFT+Fx keys.	There is no equivalent in Genero.
`gui.key.`*`x`*`.translate`	These entries could be used to map keys. For example, when the user pressed Control-U, it could be mapped to F5 for the program.	There is no equivalent in Genero.
`gui.key.radiocheck.invokeexit`	Could be used to define the key to select the RADIO or CHECK field and move to the next field.	There is no equivalent in Genero.
`gui.mswindow.button`	This entry defined the aspect of buttons on Windows™ platforms.	There is no equivalent in Genero: Front-ends will use the current platform theme when possible.
`gui.mswindow.scrollbar`	Could be used to get MS Windows™ scrollbar style.	There is no equivalent in Genero: Front-ends will use the current platform theme when possible.
`gui.scrollbar.expandwindow`	When set to true, the WTK front-end expanded the window automatically if scrollbars are needed in a screen array.	There is no equivalent in Genero.
`gui.fieldButton.style`	Could be used to define the style of BMP field buttons.	There is no equivalent in Genero.
`gui.BMPbutton.style`	Could be used to define the style of FIELD_BMP field buttons.	There is no equivalent in Genero.
`gui.entry.style`	This entry defines the underlying widgets to be used to manage form fields.	There is no equivalent in Genero.
`gui.user.font.choice`	This entry could be set to true to let the end user change the font of the application screen window.	Genero front-ends allow the user to change the font.

Entry	Description of the BDS feature	Genero equivalent
		See front-end specific documentation for option configuration.
`gui.interaction.inputarray.usehighlightcolor`	This entry could be used to highlight the current row during an INPUT ARRAY.	The current row highlighting can be controlled in Genero with the Table style attribute `highlightCurrentRow`.

`gui.form.foldertab.multiline`
`gui.folderTab.input.sendNextField`
`gui.folderTab.`*`x`*`.selection`
	These entries could be used to configure folder tabs and define the keys to be sent when a page is selected by the user.	Genero supports folder tabs with the FOLDER container in LAYOUT. An action can be defined for each folder PAGE.

`gui.keyButton.position`
`gui.keyButton.style`
`gui.button.width`
	These entries could be used to define the aspect of control frame buttons associated to ON KEY actions in dialogs like INPUT.	Default action views aspect and position can be controlled with Action Defaults attributes and with Window style attributes.

`Menu.style`
`gui.menu.timer`
`gui.menu.horizontal.*`
`gui.menu.showPagerArrows`
`gui.menuButton.position`
`gui.menuButton.style`
	These entries could be used to define the aspect of control frame buttons associated to COMMAND [KEY] actions in MENU.	Default action views aspect and position can be controlled with action attributes with window style attributes.

`gui.empty.button.visible`	This entry could be used to hide control frame buttons without text. By default, the empty buttons are visible but disabled.	Default action views aspect can be controlled with action attributes. Use for example the `defaultView` attribute to display a default button for an action.

`gui.containerType`
`gui.containerName`
`gui.mdi.*`
	These entries could be used to configure WCI windows in BDS.	To define WCI containers and children in Genero, use the ui.Interface methods.
		See Window containers (WCI) on page 1458 for more details.

`gui.toolBar.*`	These entries define the toolbar aspect in BDS.	Toolbar definition has been extended in Genero. See ToolBars for more details.
`gui.statusBar.*`	These entries define the status aspect in BDS.	The StatusBars are defined with Window presentation style attributes. See Presentation Styles for more details.
`gui.directory.images`	This entry defines the path to the directories where images (toolbar	See front-end documentation for image files located on the workstation. With Genero, image

Entry	Description of the BDS feature	Genero equivalent
	icons) are located, on the front-end workstation.	files can be located on the application server and automatically transmitted to the front-end according to the FGLIMAGEPATH environment variable.
`gui.display.<source>`	These entries could be used to redirect the ERROR / MESSAGE / COMMENT text to a specific place on the GUI screen.	The rendering of ERROR, MESSAGE or COMMENT can be configured with Window style attributes in Genero. However, it is not possible to customize keyboard NumLock / CapsLock status in Genero. See Presentation Styles for more details.

`gui.local.edit`
`gui.local.edit.error`
`gui.key.cut`
`gui.key.copy`
`gui.key.paste`
	These entry could be used to enable and configure cut/copy/ paste local keys in WTK.	Cut/Copy/Paste are defined as front-end local actions in Genero. You can bind action views with `editcut`, `editcopy`, `editpaste` predefined action names.
		See Dialog actions on page 1276 for more details.

`gui.key.*`	These entries were used to map physical key to a virtual key used in programs. For example: `gui.key.interrupt = "control-c"`	Cut/Copy/Paste are defined as front-end local actions in Genero. You can bind action views with editcut, editcopy, editpaste predefined action names. See Dialog actions on page 1276 for more details.
`gui.workSpaceFrame.nolist`	This entry could be used to define the aspect of fixed size screen arrays in forms, to render each array cell as an individual edit field.	*There is no equivalent in Genero.*

Key labels versus action defaults

In Four Js Business Development Suite (BDS), labels can be defined for keys such as **accept**, **F10** or **Control-Z**. With this feature, it is possible to easily decorate ON KEY or COMMAND KEY blocks with a button in the control frame.

With Genero Business Development Language (BDL), interaction statements can define actions with the ON ACTION blocks. These action handlers are more abstract than `ON KEY`: You identify an action by a name, while decoration is defined in form files (ACTION DEFAULTS section) or in global configuration files (.4ad file).

When adapting your code for Genero, you are free to use the traditional `ON KEY` blocks or the new `ON ACTION` blocks. Genero still supports the key label settings as in Four Js BDS. Note however that key label settings will overwrite Action Defaults settings. Additionally, if the name of the key specified in the `ON KEY` clause does not only contain alphanumeric characters (such as Control-Z), it will not be possible to define action defaults attributes for these action handlers, as action names must be simple identifiers. This is also true for Menu COMMAND labels, for example with `COMMAND"Exit program"`.

Migrating form field widgets

To get combo-boxes or check-boxes in Four Js Business Development Suite (BDS), **.per** forms could define fields with the WIDGET attribute. To ease migration, the `WIDGET` attribute and the corresponding form field widgets are still supported in Genero Business Development Language (BDL), but these are now deprecated: You should use new BDL form item types instead.

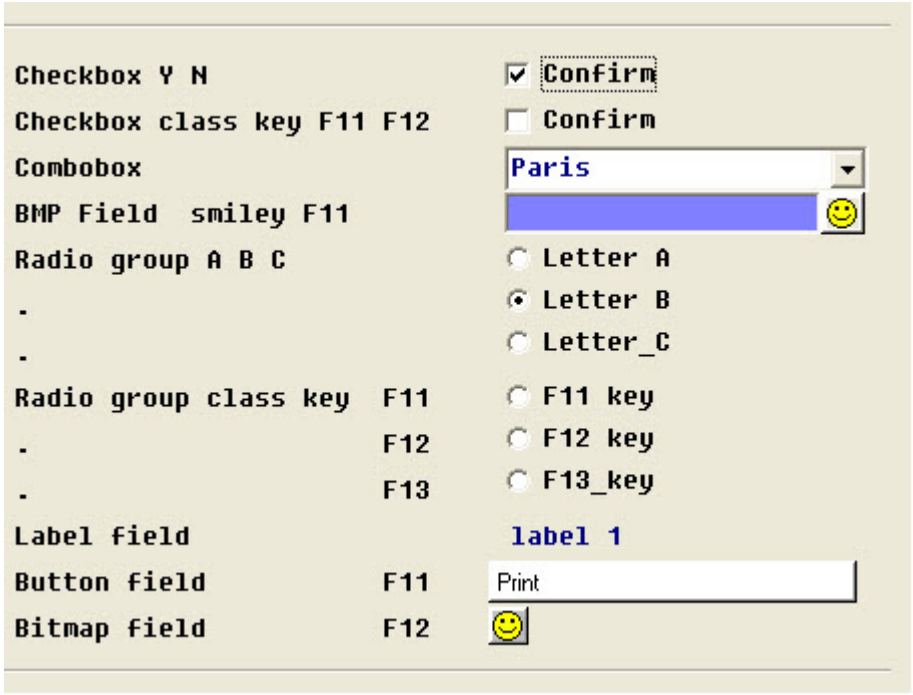


Figure 18: Four Js BDS-specific widgets

This table shows new Genero BDL form item types corresponding to old BDS WIDGET fields:

Table 89: Genero form item types corresponding to old BDS WIDGET fields

WIDGET=	Description	Genero equivalent
`WIDGET="Canvas"`	Drawing area for fgldraw functions	CANVAS item type
`WIDGET="BUTTON"`	Text push button firing key event	BUTTON item type
`WIDGET="BMP"`	Image push button firing key event	BUTTON item type
`WIDGET="CHECK"`	Checkbox field	CHECKBOX item type
`WIDGET="CHECK" + CLASS="KEY"`	Checkbox field firing key event	CHECKBOX item type + ON CHANGE trigger in program
`WIDGET="COMBO"`	Combobox field	COMBOBOX item type
`WIDGET="FIELD_BMP"`	Edit field with push button	BUTTONEDIT item type
`WIDGET="LABEL"`	Label field (no input)	LABEL item type
`WIDGET="RADIO"`	Radio group field	RADIOGROUP item type
`WIDGET="RADIO" + CLASS="KEY"`	Radio group field firing key event	RADIOGROUP item type + ON CHANGE trigger in program

Genero introduced more form item types like DATEEDIT, PROGRESSBAR.

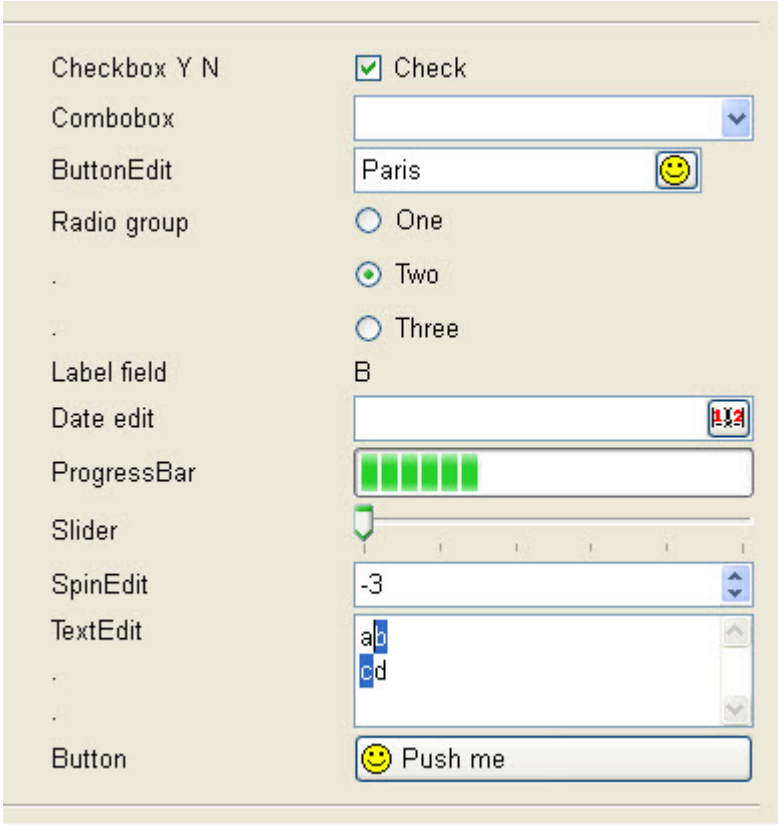


Figure 19: New form types in Genero

SCREEN versus LAYOUT

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Migrating screen arrays to tables

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Review TUI specifics

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

The default SCREEN window

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Specifying WINDOW position and size

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Front-end configuration tools

Four Js Business Development Suite (BDS) provided WTK front-end and X11 front-end specific configuration tools called "Configuration Manager" / confdesi. These tools could be used to define widget aspect (color, borders, fonts).

In Genero Business Development Language, the form items can be decorated with presentation styles for all sorts of front-ends.

Function key mapping

With Four Js Business Development Suite (BDS), when the user pressed a key modifier plus a function key (like Shift-F4 or Ctrl-F6), the key combination was mapped to a regular function key F(n+offset), because *Shift* and *Control* key modifiers are not handled in the 4GL language. The number of function keys of the keyboard was defined by the gui.key.add_function FGLPROFILE entry. For example, when this entry is set to 12 (the default), a Shift-F4 was received as F16 (4 + 12) in the program.

This feature and FGLPROFILE entry is still supported when using the traditional mode.

4GL Programming topics

FGLPROFILE: VM configuration

Genero Business Development Language (BDL) comes with redesigned software components and features. Some Four Js Business Development Suite (BDS) specific FGLPROFILE entries have been desupported. This section describes what configurations settings are no longer supported, and point to Genero equivalent features if they exist.

This table shows BDS FGLPROFILE entries related to runtime system configuration which are desupported in Genero. See the FGLPROFILE description page for supported entries:

Table 90: BDS FGLPROFILE entries related to runtime system configuration which are desupported in Genero

Entry	Description of the BDS feature	Genero BDL equivalent
`fglrun.checkDecimalPrecision`	Controls decimal variable assignment when overflow occurs. For example, a value of 1000.0 does not fit in a DECIMAL(2,0). Is false by default = no overflow error, value assigned.	*There is no equivalent in Genero.* *By default Genero assigns NULL to a decimal when overflow occurs. Can be trapped by WHENEVER ANY ERROR.*
`fglrun.ix6`	Controls Informix® version 6.x compatibility. By default BDS is compatible with I4GL 4.x	*There is no equivalent in Genero.* *By default Genero is compatible to Informix® 4gl 7.32.*
`fglrun.cmd.winnt`, `fglrun.cmd.win95`	Defines the command line to be executed for a RUN WITHOUT WAITING on Windows™ platforms.	With Genero the command program can be defined with the COMSPEC environment variable.
`fglrun.database.listvar`, `fglrun.remote.envvar`	Was used by Informix® driver to set environment variables with the ifx_putenv() function on Windows™ platforms.	*There is no equivalent in Genero.*
`fglrun.setenv.*`, `fglrun.defaultenv.*`	These entries could be used to define environment variables for all programs.	*There is no equivalent in Genero.*
`fgllic.*`	License controller related entries	With Genero you configure license settings with the `flm.*` entries. See license manager documentation for more details.

Entry	Description of the BDS feature	Genero BDL equivalent
`fglrun.server.*`	These entries could be used to define X11 front-end automatic startup.	In Genero this can be configured with `gui.server.autostart.*` entries. See Automatic front end startup on page 761 for more details.

Calling fgl_init4gl() initialization function

Four Js Business Development Suite (BDS) provided a few utility functions in the libfgl4js.42x library. This library had to be initialized with a call to fgl_init4js():

```
MAIN
   ...
  CALL fgl_init4js()
   ...
END MAIN
```

Genero Business Development Language still supports the fgl_init4js() function, but only for backward compatibility. Calling this function has no effect in Genero.

Static versus Dynamic Arrays

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for more details.

Debugger syntax changed

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for more details.

fgl_system() function

The fgl_system() function is still supported in Genero Business Development Language, but it does not raise a terminal window on the front-end as with Four Js Business Development Suite (BDS). However, some front-ends implement a workaround for this feature, based on the detection of special strings displayed to stdout by fglrun. See front-end documentation for more details.

The Channel:: methods

Genero Business Development Language provides file, socket and process I/O with the Channel built-in class, while Four Js Business Development Suite (BDS) has the `Channel::` functions. You must review your code and replace `Channel::` calls with the new API.

STRING versus CHAR/VARCHAR

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Review user-made C routines

This topic also concerns IBM® Informix® 4GL migration, see the I4GL Migration page for mode details.

Strict variable identification in SQL statements

This topic applies also to older Genero Business Development versions, see the Genero 2.20 Migration page for more details.

Default action of WHENEVER ANY ERROR

With old Four Js Business Development Suite (BDS) versions like 2.10, expression evaluation errors such as a division by zero stop the program with an error message. Genero Business Development Language behaves like IBM® Informix® 4GL and recent BDS versions like 3.55: By default, the WHENEVER ANY

ERROR action is to CONTINUE the program flow. You can change this behavior by setting the next FGLPROFILE entry to true:

```
fglrun.mapAnyErrorToError = true
```

Configuration

These topics cover configuration options of the Genero Business Development Language.

- The FGLPROFILE file on page 164
- Environment variables on page 169
- Configuring the database server connections on page 186
- Configuring the front-end connection on page 186

The FGLPROFILE file

- Understanding FGLPROFILE on page 164
- FGLPROFILE entry syntax on page 165
- List of FGLPROFILE entries on page 166

Understanding FGLPROFILE

The runtime system uses one or more configuration files in which you can define options and parameters to change the behavior of the programs.

Loading FGLPROFILE files

There are three different levels to specify a configuration file, and these files are loaded in the following order:

1. First, the runtime system reads the default configuration file provided in `FGLDIR/etc/fglprofile`. This file contains all supported entries, identifies the possible values for an entry, and documents default values. You should not modify this default configuration file.
2. Then, if the FGLPROFILE environment variable is set, the runtime system reads entries from the files specified by this environment variable. A list of files can be provided with FGLPROFILE. Files must be separated by the operating system specific path separator.

   Note: On mobile devices, it is not possible to define environment variables. To specify a custom fglprofile file for a mobile application, you must deploy a file with the name "`fglprofile`" in the *appdir* directory, along with the other application program files (`.42m`, `.42f`, and so on). Only one custom fglprofile file can be deployed for a given mobile application.
3. After loading and merging the two previous levels, the runtime system checks whether the `fglrun.defaults` entry is set. This entry defines the *program-specific profile directory*. If this directory contains a file with the same name as the current program (without a `.42r` extension), the runtime system reads the entries from that file.

The runtime system merges the different configuration files found at the three levels. If the same entry is defined in several files, the last loaded entry wins. This means that the order of precedence is:

1. Program-specific configuration file (if `fglrun.defaults` is defined in one of the other levels).
2. Configuration files defined by the FGLPROFILE environment variable, or *`appdir`*`/fglprofile`, for mobile applications.
3. The default configuration file `FGLDIR/etc/fglprofile`.

The default FGLPROFILE file

It is recommended that you NOT change the default configuration file in `FGLDIR/etc/fglprofile`. This file will be overwritten by a new installation and your changes will be lost. It is recommended that you make a copy and define your private configuration file using the FGLPROFILE environment variable.

FGLPROFILE file name

For non-mobile apps, there is no specific naming convention for FGLPROFILE configuration files. You can use a file name without an extension, or the `.txt` extension for simple text file.

On mobile devices, the name of the custom fglprofile file must be "`fglprofile`", and must be deployed under the *appdir* directory.

FGLPROFILE file encoding

The character encoding of FGLPROFILE files must match the application locale.

Defining your own FGLPROFILE entries

User-defined entries can be read with the `FGL_GETRESOURCE()` built-in function.

FGLPROFILE entry syntax

Description of the syntax of FGLPROFILE entries.

Syntax

```
  # comment
| entry-definition
```

where *entry-definition* is:

```
entry = value
```

where *entry* is:

```
ident [.ident [.ident] [...] ] ]
```

and *value* is:

```
  [-]{ digit [...] }[ . digit [...] ]
| " alphanum [...] "
| {true|false}
```

1. *comment* is a line of text that is started by a # sharp.
2. *entry* identifies the name of the entry. This can be a dot-separated list of identifiers.
3. *value* is the value of the entry; it might be a numeric value, a string literal, or a boolean value (true/false).

Usage

An FGLPROFILE entry is a line in the configuration file associating a parameter name to a value that can be specified as a numeric, string or boolean.

> **Important:** The encoding of FGLPROFILE files must match the application locale of the program. For more details about locale definition, see Application locale on page 307.

The entries are defined by a name composed of a list of identifiers separated by a dot character.

> **Note:** FGLPROFILE entry names are case insensitive. In order to avoid any confusion, it is recommended to write FGLPROFILE entry names in lower case.

If an entry is defined several times in the same file, the last entry found in the file is used. No error is raised.

The value can be a numeric literal, a string literal, or a boolean (true/false).

Numeric values are composed by an optional sign, followed by digits, followed by an optional decimal point and digits:

```
my.numeric.entry = -1566.57
```

String values must be delimited by single or double quotes. The escape character is backslash, `\t \n \r \f` are interpreted as TAB, NL, CR, FF. Double the backslash to write a backslash character (`\\`):

```
my.string.entry = "C:\\data\\test1.dbf"
```

Boolean values must be either the `true` or `false` keyword:

```
my.boolean.entry = true
```

Example

```
# Last modification: 2013-03-12/mike
report.aggregatezero = true
gui.connection.timeout = 100
dbi.database.stores.source = "C:\\data\\test1.dbf"
dbi.database.stores.prefetch.rows = 200
```

List of FGLPROFILE entries

This is a summary of supported FGLPROFILE entries.

Find more information for an FGLPROFILE entry by following the documentation link in the description of the entry.

Table 91: Partial list of supported FGLPROFILE entries

Entry	Values	Default	Description
`Dialog.currentRowVisibleAfterSort`	boolean	false	Forces current row to be shown after a sort in a table. See Dialog configuration with FGLPROFILE on page 1251.
`Dialog.fieldOrder`	boolean	false	Defines if the intermediate field triggers must be executed when a new field gets the focus with a mouse click. See Dialog configuration with FGLPROFILE on page 1251.
`dbi.default.driver`	string	NULL	Defines the default database driver. See Default database driver on page 464.
`dbi.database.`*`dbname`*`.driver`	string	NULL	Defines the database driver for a database name. See Database driver specification (driver) on page 462.

Entry	Values	Default	Description
`dbi.database.`*`dbname`*`.source`	string	NULL	Defines the data source for a database name. See Database source specification (source) on page 461.
`dbi.*`	N/A	N/A	Database interface configuration. See Connections.
`fglrun.arrayIgnoreRangeError`	boolean	false	Controls runtime behavior when array index is out of bounds. See Arrays on page 296 for more details.
`fglrun.decToCharScale2`	boolean	false	Formats DECIMAL(P) with 2 digits after the decimal point. See Floating point to string conversion on page 107.
`fglrun.defaults`	string	NULL	Defines the directory where program specific configuration files are located. See Understanding FGLPROFILE on page 164.
`fglrun.ignoreDebuggerEvent`	boolean	false	Defines whether the runtime system can swtich to debug mode. See The debugger on page 1531.
`fglrun.ignoreLogoffEvent`	boolean	false	Defines whether the runtime system ignores a CTRL_LOGOFF_EVENT on Windows™ platforms. See Responding to CTRL_LOGOFF_EVENT on page 386.
`fglrun.localization.*`	N/A	N/A	Defines load parameters for localized string resource files. See Localized strings on page 327.
`fglrun.mapAnyErrorToError`	boolean	false	Controls default action of `WHENEVER ANY ERROR`. See Exceptions on page 340.
`fglrun.mmapDisable` **Note:** This entry may be removed in a future version: It is only provided to solve file overwrite issues when doing live program files updates on Windows™ platforms.	boolean	false	Turns program files memory mapping off on Windows™ platforms. See Dynamic module loading on page 1560.
`flm.*`	N/A	N/A	License management related entries.

Entry	Values	Default	Description
			See licensing documentation.
`gui.connection.timeout`	integer	30	Defines the timeout delay (in seconds) the runtime system waits when it establishes a connection to the front-end. After this delay the program stops with an error. See Configure the GUI connection timeout on page 757.
`gui.key.add_function`	integer	none	If set, this entry defines the offset for function key mapping when using Shift-Fx and Control-Fx key modifiers. See Traditional GUI mode on page 753.
`gui.protocol.pingTimeout`	integer	600	Defines the timeout delay (in seconds) the runtime system waits for a front-end ping when there is no user activity. After this delay the program stops with an error. See Wait for front end ping timeout on page 757.
`gui.protocol.format`	string	default	Controls Front-End protocol format. Possible values are: "block", "zlib". Default is "block" (encapsulation only). See GUI protocol compression on page 758.
`gui.server.autostart.*`	N/A	N/A	Defines automatic front-end startup parameters. See Automatic front end startup on page 761.
`gui.uiMode`	string	NULL	Defines the user interface mode, to render windows in traditional I4GL mode. Possible values are: "default" or "traditional". Default is the new Genero GUI mode with real resizeable windows. See Traditional GUI mode on page 753.
`key.`*`key-name`*`.text`	string	N/A	Defines a label for an action defined with an ON KEY clause. *Provided for V3 compatibility only.* See Setting key labels on page 759.

Entry	Values	Default	Description
`mobile.environment.`*`name`* `= "`*`value`*`"`	N/A	N/A	Define environment variable values in FGLPROFILE for mobile applications. See Setting environment variables in FGLPROFILE (mobile) on page 170.
`Report.aggregateZero`	boolean	false	Defines if the report aggregate functions must return zero or NULL when all values are NULL. *Provided for V3 compatibility only.* See Report engine configuration on page 1492.
`authenticate.*`	N/A	N/A	Web services configuration. See Basic or digest HTTP authentication on page 2512
`proxy.*`	N/A	N/A	Web services configuration. See Proxy configuration on page 2512
`security.*`	N/A	N/A	Web services configuration. See HTTPS and password encryption on page 2510
`ws.*`	N/A	N/A	Web services configuration. See Server configuration on page 2513
`xml.*`	N/A	N/A	Web services configuration. See XML configuration on page 2514

Environment variables

- Setting environment variables on UNIX on page 169
- Setting environment variables on Windows on page 170
- Setting environment variables in FGLPROFILE (mobile) on page 170
- Operating system environment variables on page 171
- Database client environment variables on page 173
- Genero environment variables on page 173

Setting environment variables on UNIX™

On UNIX™ platforms, environment variables can be set through the following methods, depending on to the command interpreter used:

Bourne shell:

```
VAR=value; export VAR
```

Korn shell:

```
export VAR=value
```

C shell:

```
setenv VAR=value
```

For more details, refer to the documentation for your UNIX™ system.

Setting environment variables on Windows™

On Windows™ platforms, environment variables can be set by one of the following methods:

- In a command window, with the SET command.
- In the registry, for the current user in HKEY_CURRENT_USER or a global setting in HKEY_LOCAL_MACHINE.

For more details, refer to the documentation of your Windows™ system.

On Windows™, double quotes do not have the same meaning as on UNIX™ systems. For example, if you set a variable with the command SET VAR="abc", the value of the variable will be "abc" (with double quotes), and not abc.

When using Informix®, some variables related to the database engine must be set using the SETNET32 utility.

Setting environment variables in FGLPROFILE (mobile)

When executing applications on mobile devices, you can configure environment settings with FGLPROFILE entries. Setting an environment variable with an FGLPROFILE entry is equivalent to setting the environment variable before running the fglrun VM process on a server.

> **Note:** Environment variables set in an FGLPROFILE file are only read when the deployed application runs the mobile device. They are not read during development mode (i.e. when the VM runs on the development machine and the mobile client displays on the device). The FGLPROFILE environment variable settings are only for the VM component and are ignored by the GMA/GMI front-end component.

FGLPROFILE environment variables settings can be used to define DBDATE and DBFORMAT, if the default regional settings on the mobile must be ignored for date and numeric value formatting. Note that defining DBMONEY will have no effect, because DBFORMAT is defined automatically by the GMI or GMA front-end component before starting the VM component. Since DBFORMAT takes precedence over DBMONEY, setting DBMONEY in FGLPROFILE is useless.

> **Important:** C-runtime library variables such as LANG/LC_ALL cannot be set with FGLPROFILE entries, because the C-runtime library is (and must be) initialized before reading FGLPROFILE files.

The syntax is:

```
mobile.environment.env_name = "env_value"
```

where:

1. *env_name* is the name of the environment variable to be set.
2. *env_value* is the value for the *env_name* environment variable.

For example:

```
mobile.environment.MY_ENV_VAR = "my value"
```

The value specified in a mobile.environment entry can contain $*NAME* placeholders, that will be replaced by the actual value of the *NAME* environment variable. The *NAME* environment variable will typically be set by the front-end component, before starting the runtime system component, for example to define FGLDIR and FGLAPPDIR values.

If the environment variable contains directory or file pathes, use the UNIX path notation with `/` slashes as directory name separator, and the `:` colon as path separator.

The next example defines the FGLIMAGEPATH environment variable for the mobile app, using FGLAPPDIR and FGLDIR predefined environment variables:

```
mobile.environment.FGLIMAGEPATH = "$FGLAPPDIR/myimages:$FGLAPPDIR/icons/
myimage2font.txt:$FGLDIR/lib/image2font.txt"
```

> **Note:** During development (when executing programs on a server), consider defining environment variables such as FGLAPPDIR in the shell environment, along with the other environment variables that are defined with `mobile.environment` entries, as these are only read when executing on mobile devices.

Operating system environment variables

This section describes some well-known system environment variables that are used by Genero software components.

- LC_ALL (or LANG) on page 171
- LD_LIBRARY_PATH on page 171
- PATH on page 172
- TERM on page 172
- TERMCAP on page 172
- TERMINFO on page 172
- TMPDIR, TEMP, TMP on page 173

LC_ALL (or LANG)

Defines the current application locale on UNIX™ platforms.

The LC_ALL (or LANG) environment variable defines language, territory and codeset for programs running on UNIX™ platforms.

The codeset defined in LC_ALL is used by the runtime system to handle character strings.

It is important to set this variable properly according to the character set used by your application.

If LC_ALL is not defined, LANG is used instead.

Read the UNIX™ man page of the `setlocale()` C function for more details about this variable.

LD_LIBRARY_PATH

Defines search paths to find shared libraries on UNIX™ platforms.

The `LD_LIBRARY_PATH` environment variable defines the list of search paths for shared libraries loaded by the *dynamic linker* on UNIX™ platforms.

On some operating systems, the environment variable defining the shared library search path may have a different name.

- On a system where a 32-bit and a 64-bit environment coexist, you may need to set `LD_LIBRARY_PATH_64` to execute the 64-bit programs.
- On HP/UX, set `SHLIB_PATH`.
- On AIX®, set `LIBPATH`.
- On Mac OS X®, the usage of `DYLD_LIBRARY_PATH` is discouraged. Therefore, shared libraries that are not part of the Genero runtime system (such as database client libraries) must be found in the standard system directories (`/usr/lib`, `/usr/local/lib`)

PATH
Defines the list of search paths to find executable files.

The PATH environment variable defines the list of search paths for executable files.

On UNIX™ platforms, PATH defines the search path list for executable programs.

On Windows™ platforms, PATH defines the search path for programs and DLLs.

The path separator is a colon (`:`) on UNIX™ and a semicolon (`;`) on Windows™.

TERM
Defines the type of terminal on UNIX™ platforms.

The TERM variable is used by UNIX™ and Genero applications to identify the terminal type when running in TUI mode.

By default or when INFORMIXTERM equals *termcap*, Genero reads terminal capabilities from the file defined by the TERMCAP environment variable. When INFORMIXTERM is set to *terminfo*, Genero reads terminal capabilities from the terminfo database of the system.

TERMCAP is the older implementation of terminal capabilities database. you should set INFORMIXTERM=terminfo.

It is important to define this variable properly to match the text terminal hardware or the terminal emulation you are using.

TERMCAP
Defines the *termcap* terminal capabilities database on UNIX™ platforms.

Usage

For UNIX™ platforms, TERMCAP is an environment variable that defines to the terminal capabilities file. This variable must be used in conjunction with TERM, when INFORMIXTERM is set to *termcap*, or when INFORMIXTERM is not set.

If the TERMCAP variable is not defined, Genero tries to open `/etc/termap`. If no `/etc/termcap` file exists, the runtime system uses `$FGLDIR/etc/termcap`. You can add more terminal definitions in this file.

TERMCAP is the older implementation of terminal capabilities database. you should set INFORMIXTERM=terminfo.

It is important to define terminal capabilities properly according to the text terminal hardware or the terminal emulation you are using. Especially function keys (F1, F16) and display attributes (bold, reverse, colors) may not work if the escape sequences do not correspond to the terminal used.

For more details about the TERMCAP environment variable, please refer to your UNIX™ operating system manual.

TERMINFO
Defines the *terminfo* terminal capabilities database.

On UNIX™ platforms, the TERMINFO environment variable points to the terminal capabilities database. This variable must be used in conjunction with TERM, when INFORMIXTERM is set to *terminfo*.

You should not have to modify or set this environment variable. The default is defined by the UNIX™ system, it can be for example `/etc/terminfo`, `/usr/lib/terminfo`, or `/lib/terminfo`.

It is important to define terminal capabilities properly according to the text terminal hardware or the terminal emulation you are using. In particular, function keys (F1, F16) and display attributes (bold, reverse, colors) may not work if the escape sequences do not correspond to the terminal used.

For more details about the TERMINFO environment variable, please refer to your UNIX™ operating system manual.

TMPDIR, TEMP, TMP
Defines the directory for temporary files.

The TMPDIR, TEMP and TMP environment variables define the directory where temporary files are created by the operating system and by some other software (TMPDIR is typically used on UNIX™ platforms, TEMP and TMP are used on Windows™)

On desktop and server platforms, consider using DBTEMP to define the temp file directory for runtime system temporary files.

On mobile devices, there is no need to define the TMPDIR (or DBTEMP) environment variable: The runtime system will automatically use the appropriate temporary directory within the app sandbox file system.

Database client environment variables

Programs connecting to a database server use a database driver that in turn uses a database client library. The database client software usually needs configuration settings that are defined with environment variables. Database client environment variable define information such as installation directory of the client software, localization settings, temporary directory, and more.

Refer to the database client software documentation for the required environment variable settings.

Genero environment variables

This section lists and describes in detail all Genero specific environment variables.

- DBCENTURY on page 174
- DBDATE on page 174
- DBDELIMITER on page 175
- DBEDIT on page 175
- DBFORMAT on page 176
- DBMONEY on page 178
- DBPATH on page 179
- DBPRINT on page 179
- DBSCREENDUMP on page 180
- DBSCREENOUT on page 180
- DBTEMP on page 180
- FGL_LENGTH_SEMANTICS on page 180
- FGLAPPDIR on page 181
- FGLAPPSERVER on page 181
- FGLDBPATH on page 181
- FGLDIR on page 181
- FGLGUI on page 181
- FGLGUIDEBUG on page 181
- FGLIMAGEPATH on page 182
- FGLLDPATH on page 184
- FGLPROFILE on page 184
- FGLRESOURCEPATH on page 184
- FGLSERVER on page 185
- FGLSOURCEPATH on page 185
- FGLSQLDEBUG on page 185
- FGLWRTUMASK on page 186

- FGLWSDEBUG on page 186
- INFORMIXTERM on page 186

DBCENTURY
Specifies the expansion for the century in DATE and DATETIME values.

The DBCENTURY environment variable specifies how to expand abbreviated one- and two-digit *year* specifications within DATE and DATETIME values, especially during field input.

> **Important:** The DBCENTURY environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

Default value is "R" (prefix the entered value with the first two digits of the current year).

Values are case sensitive; only the four uppercase letters are valid.

Table 92: DBCENTURY valid values

Symbol	Algorithm for Expanding Abbreviated Years
C	Use the past, future, or current year closest to the current date.
F	Use the nearest year in the future to expand the entered value.
P	Use the nearest year in the past to expand the entered value.
R	Prefix the entered value with the first two digits of the current year.

If a year is entered as a single digit, it is first expanded to two digits by prefixing it with a zero; DBCENTURY then expands this value to four digits.

Three-digit years are not expanded.

Years before 99 AD (or CE) require leading zeros (to avoid expansion).

If the database server and the client system have different settings for DBCENTURY, the client system setting takes precedence for abbreviations of years in dates entered through the application. Expansion is sensitive to the time of execution and to the accuracy of the system clock-calendar. You can avoid the need to rely on DBCENTURY by requiring the user to enter four-digit years or by setting the CENTURY attribute in the form specification of DATE and DATETIME fields.

DBDATE
Defines the default display and input format for DATE values.

The DBDATE environment variable defines the default display and input format for DATE values.

> **Important:** The DBDATE environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

DBDATE defines the order of the month, day, and year time units within a string representing a date with numeric month and day such as "24/04/2014".

Values of DBDATE must be a restricted combination of symbols representing the position of the year (Yn), month (M) and day (D), the separator and some optional configuration options. For example, DMY4/ defines a date format with the day unit at the first position, followed by the month and the year (on 4 digits): "dd/mm/yyyy".

The separator always goes at the end of the format string (for example, DMY2/). If no separator or an invalid character is specified, the slash (/) character is the default. Specifying a 0 (zero) as separator indicates that no separator is used.

The default value of DBDATE depends on the type of platform: On destkop/server platforms, the default setting for DBDATE is: `MDY4/`. On mobile platforms, DBDATE defaults to the regional settings defined on the device.

Table 93: Valid DBDATE symbols

Symbol	Meaning in DBDATE format string
`D`	Day of month as one or two digits
`M`	Month as one or two digits
`Y2`	Year as two digits
`Y3`	Year as three digits *(Ming Guo format only)*
`Y4`	Year as four digits
`/`	Default time-unit separator for the default locale
`C1`	Ming Guo format modifier (years as digits)
`-`	Hyphen time-unit separator
`.`	Period time-unit separator
`0`	Indicates no time-unit separator

The combinations must follow a specific order:

```
{ DM | MD } { Y2 | Y3 | Y4 } { / | - | . | 0 } [C1]
{ Y2 | Y3 | Y4 } { DM | MD } { / | - | . | 0 } [C1]
```

When a form field and its corresponding variable are defined with the `DATE` type, values will be displayed according to the DBDATE format, except if a `FORMAT` attribute is defined.

The DBDATE format is also used to automatically convert a character string to/from a DATE value in programs.

Note that DBDATE takes also effect when fetching DATE values from the database into `CHAR`/`VARCHAR` program variables. However, it is not recommended to fetch date information into string variables, you should use `DATE` or `DATETIME` variables instead.

The C1 modifier can be used at the end of the DBDATE value in order to use Ming Guo date format with digit-based years. When using C1, you can use one of the Y4, Y3 or Y2 symbols for the year.

A Gregorian date format can look like `"DMY4/"`, while a Ming Guo date format would look like `"Y3MD/C1"`.

Date formatting specified in a `USING` clause or `FORMAT` attribute overrides the formatting specified in DBDATE.

DBDELIMITER
Defines the value separator for unload data files.

The DBDELIMITER environment variable defines the value delimiter for `LOAD` and `UNLOAD` instructions.

If DBDELIMITER is not defined, the default delimiter is a (`|`) pipe.

Do not use backslash or hex digits (0-9, A-F, a-f).

DBEDIT
Defines the editor program for `TEXT` fields in TUI mode.

The DBEDIT environment variable defines the editor program to modify the values of form fields defined with the `TEXT` data type, when running programs on dumb terminals.

DBFORMAT
Defines currency symbol, decimal and thousands separator for input and display of numeric values.

The DBFORMAT environment variable defines the input and display format for numeric values.

Important:

- When defined, the DBFORMAT environment variable takes precedence over DBMONEY.
- The DBFORMAT environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

The value of a DBFORMAT variable must use the following syntax:

```
front:thousands:decimal:back
```

1. *front* is the leading currency symbol, can be an asterisk (*).
2. *thousands* is a character that you specify as a valid thousands separator, can be an asterisk (*).
3. *decimal* is a character that you specify as a valid decimal separator.
4. *back* is the trailing currency symbol, can be an asterisk (*).

DBFORMAT takes precedence over DBMONEY.

If neither DBMONEY, nor DBFORMAT are defined, the default numeric formatting depends on the type of platform where the runtime system executes:

- On destkop/server platforms, the default numeric format defines the (`,`) comma as thousands separator, the (`.`) dot as decimal separator, and the ($) dollar sign as front currency symbol for MONEY values. This corresponds to `DBMONEY="$."`, or `DBFORMAT="$:,:.:"`.
- On mobile platforms, the numeric format defaults to the regional settings defined on the device. Normally, there is no need to modify these defaults.

DBFORMAT can be set to define the input and display format for values of these types:

- `MONEY` (thousands separator, decimal separator and currency symbol)
- `DECIMAL` (thousands separator, decimal separator)
- `SMALLFLOAT` (thousands separator, decimal separator)
- `FLOAT` (thousands separator, decimal separator)
- `SMALLINT` (thousands separator)
- `INTEGER` (thousands separator)
- `BIGINT` (thousands separator)

DBFORMAT can specify the leading and trailing currency symbols (but not their default positions within a monetary value) and the decimal and thousands separators. The decimal and thousands separators defined by DBFORMAT apply to both monetary and other numeric data.

The instructions affected by the setting in DBFORMAT include (but are not restricted to) these items:

- `USING` operator.
- `FORMAT` field attribute.
- `DISPLAY` or `PRINT` statement (default formatting of numeric values).
- `LET` statement, where a `CHAR`, `VARCHAR` or `STRING` variable is assigned a monetary or number value.
- `LOAD` and `UNLOAD` statements that use ASCII files (or whatever the locale regards as a *flat* file) to pass data to or from the database.

The asterisk (*) specifies that a symbol or separator is not applicable; it is the default for any *front*, *thousand*s, or *back* term that you do not define.

If you specify more than one character for *decimal* or *thousand*s, the values in the *decimal* or *thousands* list cannot be separated by spaces (nor by any other symbols). However, only the first character will be used to display numeric or currency values, when converting strings to numbers and when entering values in form fields.

Any printable character that your locale supports is valid for the thousands separator or for the decimal separator, except:

- Digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
- <, >, |, ?, !, =, [,]

The same character cannot be both the thousands and decimal separator. A blank space (ASCII 32) can be the thousands separator (and is conventionally used for this purpose in some locales). The asterisk (*) symbol is valid as the decimal separator, but is not valid as the thousands separator.

Enclosing the DBFORMAT specification in a pair of single quotation marks is recommended to prevent the shell from attempting to interpret (or execute) any of the DBFORMAT characters.

The setting in DBFORMAT affects how formatting masks of the `FORMAT` attribute and `USING` operator are interpreted. In formatting masks of `FORMAT` and `USING`, these symbols are not literal characters but are placeholders for what DBFORMAT specifies:

- The dollar ($) sign is a placeholder for the *front* currency symbol.
- The comma (,) is a placeholder for the thousands separator.
- The period (.) is a placeholder for the decimal separator.
- The at (@) sign is a placeholder for the *back* currency symbol.

This table illustrates the results of different combinations of DBFORMAT setting and format string on the same value.

Table 94: Results of combinations of DBFORMAT setting and format string on the same value

Value	Format String	DBFORMAT	Result
`1234.56`	`$#,###.##`	`$:,:.:`	`$1,234.56`
`1234.56`	`$#,###.##`	`:.:,:DM`	`1.234,56`
`1234.56`	`#,###.##@`	`$:,:.:`	`1,234.56`
`1234.56`	`#,###.##@`	`:.:,:DM`	`1.234,56DM`

When the user enters numeric or currency values in fields, the runtime system behaves as follows:

- If a symbol is entered that was defined as a decimal separator in DBFORMAT, it is interpreted as the decimal separator.
- For `MONEY` fields, it disregards any *front* (leading) or *back* (trailing) currency symbol and any thousands separators that the user enters.
- For `DECIMAL` fields, the user must enter values without currency symbols.

When the runtime system displays or prints values:

- The DBFORMAT-defined leading or trailing currency symbol is displayed for `MONEY` values.
- If a leading or trailing currency symbol is specified by the `FORMAT` attribute for non-MONEY data types, the symbol is displayed.
- The thousands separator is not displayed unless it is included in a formatting mask of the `FORMAT` attribute or of the USING operator.

When `MONEY` values are converted to character strings by the `LET` statement, both automatic data type conversion and explicit conversion with a `USING` clause insert the DBFORMAT-defined separators and currency symbol into the converted strings.

For example, suppose DBFORMAT is set as follows:

```
*:.:,:SFr
```

The value 1234.56 will print or display as follows:

```
1234,56SFr
```

Here `SFr` stands for the Swiss Franc currency symbol. Values input by the user into a screen form are expected to contain commas, not periods, as their decimal separator because DBFORMAT has `*:.:,:SFr` as its setting in this example.

When using a graphical front-end, the decimal separator of the numeric keypad will produce the character defined by this environment variable.

DBMONEY
Defines currency symbol and decimal separator for input and display of numeric values, when DBFORMAT is not defined.

The DBMONEY environment variable defines the currency symbol and the decimal separator for numeric values.

Important:

- When defined, the DBFORMAT environment variable takes precedence over DBMONEY.
- The DBMONEY environment variable is also used by the IBM® Informix® database client and server to make date to string conversions.

The value of a DBMONEY variable must use the following syntax:

```
front{.|,}back
```

1. *front* is a character string representing a leading currency symbol that precedes the value.
2. *back* is a character string representing a trailing currency symbol that follows the value.

If neither DBMONEY, nor DBFORMAT are defined, the default numeric formatting depends on the type of platform where the runtime system executes:

- On destkop/server platforms, the default numeric format defines the (`,`) comma as thousands separator, the (`.`) dot as decimal separator, and the (`$`) dollar sign as front currency symbol for MONEY values. This corresponds to `DBMONEY="$."`, or `DBFORMAT="$:,:."`.
- On mobile platforms, the numeric format defaults to the regional settings defined on the device. Normally, there is no need to modify these defaults.

DBMONEY can only define the currency symbol and decimal separator characters must be specified in this environment variable. If you want to define the thousands separator, use the DBFORMAT environment variable instead. However, if only DBMONEY is used, an implicit thousands separator is selected.

The currency symbol in DBMONEY can be up to seven characters long and can contain any character except a comma or a period. It can be non-ASCII characters if the current locale supports a code set that defines the non-ASCII characters you use.

The DBMONEY environment variable can be set to define the input and display format for values of the following types:

- `MONEY` (thousands separator, decimal separator and currency symbol)
- `DECIMAL` (thousands separator, decimal separator)
- `SMALLFLOAT` (thousands separator, decimal separator)
- `FLOAT` (thousands separator, decimal separator)
- `SMALLINT` (thousands separator)
- `INTEGER` (thousands separator)

- `BIGINT` (thousands separator)

Numeric values will be displayed in forms and reports according to this environment variable.

DBMONEY will also be used for implicit data conversion between numeric values and character strings.

The position of the currency symbol (relative to the decimal separator) indicates whether the currency symbol appears before or after the `MONEY` value. When the currency symbol is positioned in DBMONEY before the decimal separator, it is displayed before the value ($1234.56). When it is positioned after the decimal separator, it is displayed after the value (1234.56F).

The runtime system recognizes the period (.) and the comma (,) as decimal separators. All other characters are considered to be part of the currency symbol. For example, ", FR" defines a MONEY format with the comma as decimal separator and the string " FR" (including the space) as the currency symbol.

Because only its position within a DBMONEY setting indicates whether a symbol is the *front* or *back* currency symbol, the decimal separator is required. If you use DBMONEY to specify a *back* symbol, for example, you must supply a decimal separator (a comma or period). Similarly, if you use DBMONEY to change the decimal separator from a period to a comma, you must also supply a currency symbol.

To avoid ambiguity in displayed numbers and currency values, do not use the thousands separator of DBFORMAT as the decimal separator of DBMONEY. For example, specifying comma as the DBFORMAT thousands separator dictates using the period as the DBMONEY decimal separator.

When using a graphical front-end, the decimal separator of the numeric keypad will produce the character defined by this environment variable.

DBPATH
Defines the paths to search for Genero program resource files.

For IBM® Informix® 4GL compatibility, DBPATH is used by the runtime system to find resource files such as form definitions.

> **Important:** The DBPATH environment variable is also used by the IBM® Informix® SE engine and SQLite, to define the path list to find database files. Genero has introduced the FGLRESOURCEPATH environment variable to not interfer with the database DBPATH settings. Consider dedicating DBPATH for database configurations, and use the FGLRESOURCEPATH to define program resource path list.

When FGLRESOURCEPATH is not defined, DBPATH environment variable is used to define the search paths for:

1. Form files loaded (.42f),
2. Message files (.iem),
3. Action defaults files (.4ad),
4. Presentation styles files (.4st),
5. Start menu files (.4sm),
6. Toolbar files (.4tb),
7. Topmenu files (.4tm),
8. Compiled localized strings files (.42s).

By the default, the runtime system looks for resource files is the current directory.

DBPATH must contain a list of paths, separated by the operating system specific path separator.

The path separator is platform specific (**":"** on UNIX™ platforms and **";"** on Windows™ platforms).

DBPRINT
Defines the print device to be used by reports.

The DBPRINT environment variable specifies the print device to be used by reports defined `TO PRINTER`.

On UNIX™ systems, the DBPRINT environment variable typically contains the printer queue command (such as **lp**).

To have the runtime system print to the printer on the client running the Genero Desktop Client (GDC), set `DBPRINT=FGLSERVER`.

DBSCREENDUMP
Defines the output file name for text screen shots.

The DBSCREENDUMP environment variable defines the output file name for text screen shots when pressing Ctrl-P.

When using the TUI mode, if the user pressed the Ctrl-P key, the runtime system will dump the current screen into the file defined by this variable.

Unlike DBSCREENOUT, the output of DBSCREENDUMP includes the escape sequences of TTY attributes, which makes it less readable.

DBSCREENOUT
Defines the output file name for text screen shots.

The DBSCREENOUT environment variable defines the output file name for text screen shots when pressing Ctrl-P.

When using the TUI mode, if the user pressed the Ctrl-P key, the runtime system will dump the current screen into the file defined by this variable.

Unlike DBSCREENDUMP, the output of DBSCREENOUT excludes the escape sequences of TTY attributes.

DBTEMP
Defines the directory for temporary files.

The DBTEMP environment variable defines the directory for temporary files created by the runtime system.

> **Important:** The DBTEMP environment variable is also used by the IBM® Informix® database client and server for temporary files.

This environment variable is use to create temporary files for:

1. `TEXT` or `BYTE` data located in a temporary file (`LOCATE IN FILE` without file name specification).
2. Temporary files of emulated scrollable cursors when the database engine does not support this feature.
3. Temporary file name generation with os.Path.makeTempName().
4. Temporary files created by the Web Services API, such as com.HTTPResponse.getFileResponse on page 2073.

On mobile devices, do not set DBTEMP environment variable: The runtime system will automatically use the appriopriate temporary directory withing the app sandbox file system.

FGL_LENGTH_SEMANTICS
Defines the length semantics to be used in programs.

Define the FGL_LENGTH_SEMANTICS environment variable to specify byte or character length semantics, by setting the value to BYTE or CHAR, respectively.

If the variable is not set, byte length semantics will be used by default.

When using a single-byte character set such as ISO-8859-1, use byte length semantics (the default). If the application character set is UTF-8, you should use char length semantics.

FGLAPPDIR
Contains the path to the application directory when executing on a mobile device.

When executing on mobile devices, the FGLAPPDIR environment variable is an automatic environment variable that contains the path to the *appdir* directory, containing application program files (`.42m`, `.42f`, and other resources).

This variable is typically used to define environment variables with `mobile.environment` FGLPROFILE entries, relative to the mobile appdir where application program files and resources are located.

> **Note:** During development (when executing programs on a server), consider defining the FGLAPPDIR in the shell environment, along with the other environment variables that are defined with `mobile.environment` entries, as these are only read when executing on mobile devices.

FGLAPPSERVER
Defines the listening port of the Web service in development context.

The FGLAPPSERVER environment variable defines the port on which the web service server will be started.

During development, define this environment variable before starting the web service server program, to let web service clients connect directly to the runtime system.

In production, Genero Application Server (GAS) is used to deploy web services servers. The GAS will automatically set FGLAPPSERVER. Do not manually set FGLAPPSERVER when GAS is involved.

FGLDBPATH
Defines the path to database schema files for compilers.

The fglcomp and fglform compilers need database schema files to compile source modules and forms. The path to the database schema files can be specified with FGLDBPATH.

If FGLDBPATH is not defined, the current directory is the default path for the database schema files. You can provide a list of paths, separated by the operating system specific path separator. FGLDBPATH is only used in development.

FGLDBPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is **":"** on UNIX™ platforms and **";"** on Windows™ platforms.

FGLDIR
Defines the installation directory of Genero Business Development Language.

The FGLDIR environment variable defines the installation directory of the runtime system and compilers of Genero.

When executing on a mobile device, the FGLDIR environment variable is automatically set by the front-end component, before starting the runtime system component. As result, it is possible to use the $FGLDIR keyword in FGLPROFILE environment variable settings when executing on mobile devices.

FGLGUI
Defines the user interface mode to be used by the program.

The FGLGUI environment variable indicates whether the applications are run in TUI or GUI mode.

When set to 0 (zero), the application executes in TUI mode.

When set to 1 (one, the default), the application executes in GUI mode and needs a front-end to display application windows.

FGLGUIDEBUG
Defines the debug level in GUI mode.

The FGLGUIDEBUG environment variable defines the debug level, when the GUI mode is used by the program.

By setting FGLGUIDEBUG to 1, the runtime system will display AUI protocol exchanges in the stderr output of the console running the program on the server.

The runtime system displays detailed information about user interface events that occur during program execution.

FGLIMAGEPATH
Defines the search paths for VM server image files.

FGLIMAGEPATH basics

The FGLIMAGEPATH environment variable is used by the runtime system, to find image resources on the server where the program executes, when the image name specified in the form element is not an URL that can be directly resolved and fetched by the front-end.

Image resources found through FGLIMAGEPATH will be transmitted to the front-end for display.

FGLIMAGEPATH defines a list of directories and/or image-to-font-glyph mapping files: If a path of FGLIMAGEPATH is a directory, it will be used for image file and font file lookup. If the element is a file name, it will be used as an image-to-font-glyph mapping file.

FGLIMAGEPATH setting on mobile devices

When executing on a mobile device, the environment variables must be defined with `mobile.environment` FGLPROFILE entries. The FGLAPPDIR and FGLDIR environment variables are automatically defined by the front-end component, and can be referenced with the $FGLAPPDIR and $FGLDIR placeholders, when defining FGLIMAGEPATH in FGLPROFILE:

```
mobile.environment.FGLIMAGEPATH = "$FGLAPPDIR/myimages:$FGLAPPDIR/icons/
myimage2font.txt:$FGLDIR/lib/image2font.txt"
```

For more details about environment variable settings for mobile apps, see Setting environment variables in FGLPROFILE (mobile) on page 170.

Default behavior when FGLIMAGEPATH is not defined

If the FGLIMAGEPATH environment variable is not defined, the runtime system will by default:

- Find image resource files in the current working directory where the BDL program executes.

  Note: When executing the app on an iOS device, instead of searching the current working directory, image resources are by default found in the *appdir* directory.
- Use `FGLDIR/lib/image2font.txt` along with `FGLDIR/lib/FontAwesome.ttf`, for image to font glyph mapping (to get default icons).

Order of precedence in FGLIMAGEPATH

It is possible to mix several image file directories with several image-to-font-glyph mapping files in FGLIMAGEPATH:

The list of mapping files and directories defines the order of precedence to resolve conflicts, when several image names can resolve to several image resources.

For example, if a form element defines an image as "smiley", and if FGLIMAGEPATH is defined as:

```
/opt/myapp/images:/opt/myapp/image2font.txt
```

If the `/opt/myapp/images` directory contains an image file `"smiley.png"`, and the `/opt/myapp/image2font.txt` file contains a mapping for `"smiley"`, the `"smiley.png"` file from `/opt/myapp/images` will be selected by the runtime system.

If FGLIMAGEPATH is defined as follows:

```
/opt/myapp/image2font.txt:/opt/myapp/images
```

The mapping for smiley to font glyph would take precedence.

FGLIMAGEPATH syntax

FGLIMAGEPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is **":"** on UNIX™ platforms and **";"** on Windows™ platforms.

For example, on UNIX:

```
$ export FGLIMAGEPATH="/var/myapp/myimages:$FGLDIR/lib/image2font.txt"
```

Image-to-font-glyph mapping

Image names can be mapped to font glyphs when at least one file path is specified in FGLIMAGEPATH. The runtime system distinguishes file paths (as image-to-font-glyph mapping files), from directory paths (as locations to file plain image files and font files).

> **Important:** The directory to the font file must be specified in FGLIMAGEPATH, except if the font file is located in the same directory as the mapping file.

A default mapping file ("`image2font.txt`") and its corresponding font file ("`FontAwesome.ttf`") are provided in `FGLDIR/lib`. If FGLIMAGEPATH is not defined, the runtime system will use these files to make the image to font glyph mapping.

> **Important:** When providing your own customized font file, it must be a valid TTF file. For example, changing the file name is not sufficient to turn it into another different font: In order to produce a valid TTF file, use font management tools such as FontForge (http://fontforge.github.io/en-US/) or Fontello (http://fontello.com). Further, to target Microsoft Internet Explorer (version 11), you will need to patch the generated TTF file to remove embedding limitations from TrueType fonts, by setting the `fsType` field in the OS/2 table to zero. This modification can be done with freeware tools like ttembed

The image-to-font-glyph mapping file must have the following syntax:

```
image-name=font-file:hexa-ordinal[:color-spec]
```

where:

1. *image-name* - is the name of the image to be mapped to a font character.
2. *font-file* - is the file name containing the font definitions.
3. *hexa-ordinal* - is the font glyph position in the font file, in hexadecimal notation.
4. *color-spec* - is the color to be used, in RGB hexadecimal format or as color alias as defined in presentation style colors. This field is optional: If not specified, the glyph will be displayed in a default color according to the front-end platform.

Lines starting with the # sharp character are considered as comment lines and ignored.

For example:

```
# Common icons
camera=FontAwesome.ttf:f030
file=FontAwesome.ttf:f0f6:#8B0000
smiley=FontAwesome.ttf:f118:yellow
# Traffic lights
circle-red=FontAwesome.ttf:f111:red
circle-orange=FontAwesome.ttf:f111:orange
```

```
circle-green=FontAwesome.ttf:f111:green
```

FGLIMAGEPATH and gICAPI web components

For applications executing on a server and displaying on GDC/GMA/GMI front-ends in client/server mode (not through the GAS), you can use the FGLIMAGEPATH environment variable to locate the HTML files of gICAPI web components on the server. Like image resources, the web component files will be automatically transferred to the front-end.

FGLLDPATH

Defines the search paths to load program modules.

The FGLLDPATH environment variable defines the search paths to load C extensions and modules, and by default to find sources with the debugger.

A program can be composed by several p-code modules (.42m) and can use C extensions. When linking and when executing the program, the runtime system must known where to search for these modules. You can use the FGLLDPATH environment variable to define the search paths to load C extensions and p-code modules.

FGLLDPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is **":"** on UNIX™ platforms and **";"** on Windows™ platforms.

The FGLLDPATH variable is used at link time and at run time.

The directories are searched for the modules in the following order:

1. The current directory.
2. The directory where the program (.42r) file resides.
3. A path defined in the FGLLDPATH environment variable.
4. The FGLDIR/lib directory.

If FGLSOURCEPATH is not defined, the debugger will use FGLLDPATH to find program sources.

FGLPROFILE

Defines the configuration files to be used by the runtime system.

Usage

The FGLPROFILE environment variable defines a list of configuration files to be used by the runtime system.

If FGLPROFILE is not set, the runtime system reads entries from the default configuration file located in FGLDIR/etc/fglprofile.

FGLPROFILE can define one unique configuration file, or a list of files to be loaded sequentially.

FGLPROFILE must contain a list of file paths, separated by the operating system specific path separator. The path separator is **":"** on UNIX™ platforms and **";"** on Windows™ platforms.

> **Note:** On mobile devices, it is not possible to define environment variables. To specify a custom fglprofile file for a mobile application, you must deploy a file with the name "`fglprofile`" in the *appdir* directory, along with the other application program files (`.42m`, `.42f`, and so on). Only one custom fglprofile file can be deployed for a given mobile application.

FGLRESOURCEPATH

Defines search path for resource files.

The FGLRESOURPATH environment variable is used to define the search paths for:

1. Form files loaded (.42f),
2. Message files (.iem),

3. Action defaults files (.4ad),
4. Presentation styles files (.4st),
5. Start menu files (.4sm),
6. Toolbar files (.4tb),
7. Topmenu files (.4tm),
8. Compiled localized strings files (.42s).

For compatibility with Informix® 4GL, DBPATH is used by default to search for resource files such as form files and XML files used by the program. However, DBPATH is also used by the Informix® database software to locate databases: Informix® Dynamic Server uses DBPATH to let you specify fallback servers if INFORMIXSERVER is not available, and former Informix® Standard Engine needs DBPATH to find **.dbs** database files. This can be a problem when connecting from a machine where path format is not the same as on the remote database server: It is not possible to mix UNIX™ and DOS path formats in DBPATH. To work around this Informix® limitation, FGLRESOURCEPATH can be used instead of DBPATH to specify the directories of program resource files. You are then free to define DBPATH as Informix® requires.

The path separator is platform specific (**":"** on UNIX™ platforms and **";"** on Windows™ platforms).

By the default, the runtime system looks for resource files is the current directory.

On mobile platforms, localize string files are searched by default in the language sub dirs of the app directory. For more details, see Using localized strings at runtime on page 331.

FGLSERVER
Defines the graphical front-end form the application.

In GUI mode, FGLSERVER defines the hostname and port of the graphical front end the runtime system will connect to in order to display application forms.

The values for the FGLSERVER environment variable must be specified with the following syntax:

```
{hostname|ipaddress}[:servnum]
```

1. *hostname* is the name of a machine on the network.
2. *ipaddress* is the IP V4 address (Ex: 10:0:0:105).
3. *servnum* identifies the front end.

The *servnum* parameter defines the front end server number (first is 0, second is 1, and so on). This defines implicitly the TCP port number the front end is listening to, as an offset for the base port 6400. For example, FGLSERVER=cobra:1 will use the TCP port 6401 (6400 + 1). This parameter is optional, when not specified, it defaults to zero (i.e. port 6400).

FGLSOURCEPATH
Defines the path to program source files.

The debugger needs to access the source files to display program code. By default, the current directory and the directories defined by FGLLDPATH are searched for the source files.

FGLSOURCEPATH must contain a list of paths, separated by the operating system specific path separator. The path separator is **":"** on UNIX™ platforms and **";"** on Windows™ platforms.

FGLSQLDEBUG
Defines the debug level for tracing SQL instructions.

If FGLSQLDEBUG is set to a value greater than zero, you get a debug trace in the standard error channel for every SQL instruction executed by the program.

FGLSQLDEBUG should only be used in development, or on a production site in order to identify a problem related to SQL statements.

FGLWRTUMASK
Defines the umask to be used by the license manager.

The FGLWRTUMASK environment variable is used by the fglWrt license manager to create the `FGLDIR/lock` directory.

This variable defines the umask to create the `FGLDIR/lock` directory.

The default is 000, which creates a directory with **rwxrwxrwx** rights.

FGLWSDEBUG
The `FGLWSDEBUG` environment variable enables web services library debugging.

Set the FGLWSDEBUG environment variable to turn on debug information display in the web services library.

Table 95: FGLWSDEBUG variable values

Value	Definition
0	No data displayed; debug turned off.
1	Display socket errors.
2	Display HTTP bodies of incoming and outgoing requests (the XML content)
3	Display all information about incoming and outgoing requests (HTTP headers + HTTP bodies)

INFORMIXTERM
Defines terminal control library to be used.

The INFORMIXTERM environment variable indicates what terminal capabilities database must be used by the runtime system when running a program in TUI mode on a dumb terminal.

Possible values of INFORMIXTERM are *terminfo* and *termcap*. If the variable is not set, it defaults to *termcap*.

When set to *termcap* (the default), the runtime system reads terminal capabilities from the file defined by the TERMCAP environment variable.

When set to *terminfo*, the runtime system reads terminal capabilities from the terminfo database of the system (ncurses).

Configuring the front-end connection

In order to execute a Genero program with a graphical user interface, you need to specify the front-end (i.e. the graphical server) to the runtime system.

In development mode, the target front-end is defined with the FGLSERVER environment variable. However, there are various technologies to render a Genero application, according to the front-end platform (PC, mobile device, web browser).

Details about front-end configuration for the runtime system can be found the user interface basics chapter of this manual.

Configuring the database server connections

Before running a Genero program using a database, you must configure the connection parameters to access the database server.

There are different solutions to define database connection parameters, consider using and indirect database connection configuration, by using an abstract database name in programs, and define the real database source, driver with FGLPROFILE entries.

The database configuration details can be found the SQL support chapter of this manual.

Language basics

These topics cover the basics for the Genero Business Development Language

- Syntax features on page 188
- Data types on page 191
- Type conversions on page 211
- Literals on page 225
- Expressions on page 229
- Operators on page 234
- Flow control on page 267
- Functions on page 278
- Variables on page 281
- Constants on page 291
- Records on page 294
- Arrays on page 296
- Types on page 303

Syntax features

Genero BDL is an English-like programming language, easy to write and read.

- Lettercase insensitivity on page 188
- Whitespace separators on page 189
- Quotation marks on page 189
- Escape symbol on page 189
- Statement terminator on page 190
- Comments on page 190
- Identifiers on page 191
- Preprocessor directives on page 191

Lettercase insensitivity

Genero Business Development Language (BDL) is case insensitive, making no distinction between uppercase and lowercase letters, except within quoted strings.

Use pairs of double (") or single (') quotation marks in the code to preserve the lettercase of character literals, filenames, and names of database entities.

You can mix uppercase and lowercase letters in the identifiers that you assign to language entities, but any uppercase letters in identifiers are automatically shifted to lowercase during compilation.

It is strongly recommended that you define a naming convention for your projects. For example, you can use underscore notation (`get_user_name`). If you plan to use the Java™ notation (`getUserName`), do not forget that Genero BDL is case insensitive (`getusername` is the same identifier as `getUserName`).

With Genero BDL you can import and use Java™ classes and objects in BDL code. Genero BDL is case-sensitive regarding Java™ elements.

Whitespace separators

Genero Business Development Language (BDL) is free-form, like C or Pascal, and generally ignores TAB characters, LINEFEED characters, comments, and extra blank spaces between statements or statement elements. You can freely use these whitespace characters to enhance the readability of your source code.

Blank (ASCII 32) characters act as delimiters in some contexts. Blank spaces must separate successive keywords or identifiers, but cannot appear within a keyword or identifier. Pairs of double (") or single (') quotation marks must delimit any character string that contains a blank (ASCII 32) or other whitespace character, such as LINEFEED or RETURN.

Quotation marks

In the Genero BDL language, string literals are delimited by single (') or double (") quotation marks.

```
'Valid character string'
"Another valid character string"
```

Do not mix double and single quotation marks as delimiters of the same string. The following is not a valid character string:

```
'Not A valid character string"
```

To include literal quotation marks within a quoted string, precede each literal quotation mark with the backslash (\), or else enclose the string between a pair of the opposite type of quotation marks:

```
MAIN
   DISPLAY "Type 'Y' if you want to reformat your disk."
   DISPLAY 'Type "Y" if you want to reformat your disk.'
   DISPLAY 'Type \'Y\' if you want to reformat your disk.'
END MAIN
```

A string literal can be written on multiple lines. The compiler merges lines by removing the newline character.

In the SQL language, the standard specifications recommend that you use single quotes for string literals and double quotes for database object identifiers like table or column names. When accessing a non-Informix database, double quotation marks might not be recognized as database object name delimiters. As a general rule, use single quoted string literals in SQL statements, and use non-quoted, lowercase database object identifiers.

Escape symbol

The Genero Business Development Language (BDL) compiler treats a backslash (\ ) as the default escape symbol, and treats the immediately following symbol as a literal, except for special characters such as \r or \t.

See the string literals reference for the complete list.

To specify anything that includes a literal backslash, enter double (\\ ) backslashes wherever a single backslash is required. Similarly, use \\\\ to represent a literal double backslash.

```
MAIN
  DISPLAY "\a"   -- displays a
  DISPLAY "\r"   -- displays CR
  DISPLAY "\n"   -- displays NL
  DISPLAY "\ta"  -- displays <tab>a
  DISPLAY "\\"   -- displays \
  DISPLAY "\\\\" -- displays \\
END MAIN
```

Statement terminator

Genero Business Development Language (BDL) requires no statement terminators, but you can use the semicolon (;) as a statement terminator in some cases.

For example, you can add a semicolon statement terminator for PREPARE and PRINT statements.

```
MAIN
  DISPLAY "Hello, World"  DISPLAY "Hello, World"
  DISPLAY "Hello, World"; DISPLAY "Hello, World"
END MAIN
```

Comments

For clarity and to simplify program maintenance, it is recommended that you document your code by including comments in your source files.

A source comment is text in the source code to assist human readers, but which BDL ignores.

You can use comment indicators during development to disable instruction temporarily, without removing them from your source code modules.

A source comment can be specified by any of the following:

- A pair of minus signs (--) indicates a comment that terminates at the end of the current line. This comment indicator conforms to the ANSI standard for SQL.
- The sharp (#) symbol indicates a comment that terminates at the end of the current line.
- A starting left-brace ({) starts a comment. It can be followed by any character (including line breaks). The comment ends when the closing right-brace (}) symbol is found.

```
MAIN
  -- DISPLAY "This line will be ignored."
  # DISPLAY "This line will be ignored."
  {
  DISPLAY "This line will be ignored."
  DISPLAY "This line will be ignored."
  }
  DISPLAY "Hello, World"
END MAIN
```

Within a quoted string, the compiler interprets comment indicators as literal characters, rather than as comment indicators.

You cannot use braces ({ }) to nest comments within comments.

Comments cannot appear in the form section defining a layout grid, such as SCREEN, TABLE, TREE, or GRID.

The # symbol cannot indicate comments in an SQL statement block nor in the text of a prepared statement.

You cannot specify consecutive minus signs (--) in arithmetic expressions, as BDL interprets what follows as a comment. Instead, use a blank space or parentheses to separate consecutive arithmetic minus signs.

Do not follow the -- comment indicator with the sharp (#) symbol, unless you intend to compile the same source file with the Informix® 4GL product. The --# specific comment indicator is used to distinguish Informix® 4GL code from Genero BDL code. This conditional code compilation technique can be inverted by enclosing code blocks between --#{ and --#} comments:

```
MAIN
  --# DISPLAY "Ignored by I4GL, but compiled with BDL."
  --#{
```

```
      DISPLAY "Ignored by BDL, but compiled with I4GL."
  --#}
END MAIN
```

To summarize:

- Code lines starting with `--#` are compiled with Genero BDL, but ignored by Informix® 4GL.
- Code blocks surrounded with `--#{` and `--#}` are compiled with Informix® 4GL, but ignored by Genero BDL.

Identifiers

A Genero Business Development Language (BDL) identifier is a character string that is declared as the name of a program entity.

Identifiers must conform to the following rules:

- It must include at least one character, without any limitation in size.
- Only ASCII letters, digits, and underscore (_) symbols are valid.
- Blanks, hyphens, and other non-alphanumeric characters are not allowed.
- The initial character must be a letter or an underscore.
- Common identifiers are not case sensitive, so `my_Var` and `MY_vaR` both denote the same identifier. However, in some cases, identifiers are case sensitive (like action names in the AUI tree). It is recommended to always write identifiers in lower case to avoid mistakes.

Within non-English locales, BDL identifiers can include non-ASCII characters in identifiers, if those characters are defined in the code set of the current locale. In multibyte East Asian locales that support languages whose written form is not alphabet-based (such as Chinese, Japanese, or Korean), an identifier does not need to begin with a letter. It is however recommended to program in ASCII.

Preprocessor directives

Genero Business Development Language (BDL) supports preprocessing instructions, which allow you to write macros and conditional compilation rules.

```
&include "myheader.4gl"
FUNCTION debug( msg )
  DEFINE msg STRING
&ifdef DEBUG
  DISPLAY msg
&endif
END FUNCTION
```

> **Note:** Use the preprocessor with care, and only when there is no native language solution. Do not overcrowd your source code with preprocessing directive, that would make the code unreadable and unmaintainable.

Data types

Selecting the correct data type assists you in the input, storage, and display of your data.

Table 96: Genero Business Development Language data types

Data type	Description
BIGINT	8 byte signed integer
BOOLEAN	TRUE/FALSE boolean
BYTE	Large binary data (images)

Data type	Description
CHAR[(n)]	Fixed size character strings
DATE	Simple calendar dates
DATETIME q1 TO q2	High precision date and hour data
DECIMAL[(p[,s])]	High precision decimals
FLOAT[(p)]	8 byte floating point decimal
INTEGER	4 byte signed integer
INTERVAL q1 TO q2	High precision time intervals
MONEY[(p[,s])]	High precision decimals with currency formatting
SMALLFLOAT	4 byte floating point decimal
SMALLINT	2 byte signed integer
STRING	Dynamic size character strings
TINYINT	1 byte signed integer
TEXT	Large text data (plain text)
VARCHAR[(n[,r])]	Variable size character strings

BIGINT

The BIGINT data type is used for storing very large whole numbers.

Syntax

```
BIGINT
```

Usage

The storage of BIGINT variables is based on 8 bytes of signed data (= 64 bits).

The value range is from -9,223,372,036,854,775,807 to +9,223,372,036,854,775,807.

BIGINT variables can be initialized with integer literals:

```
MAIN
  DEFINE i BIGINT
  LET i = 9223372036854775600
  DISPLAY i
END MAIN
```

When assigning a whole number that exceeds the BIGINT range, the overflow error -1284 will be raised.

BIGINT variables are initialized to zero in functions, modules and globals.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

BYTE

The BYTE data type stores any type of binary data, such as images or sounds.

Syntax

```
BYTE
```

Usage

A BYTE or TEXT variable is a handle for a large object (LOB), that is stored in a file or in memory. Such data type is a complex type that cannot be used like INTEGER or CHAR basic types: It is designed to handle a large amount of data and has different semantics as simple types. The main difference with simple data types, is the fact that you must specify the storage with the LOCATE instruction, before using BYTE and TEXT variables.

The maximum size of data that can be handled by BYTE and TEXT variable is theoretically 2^31 bytes (~2.14 Gigabytes), but the practical limit depends from the disk or memory resources available to the process.

BYTE and TEXT variable must be initialized with the LOCATE instruction before usage. The LOCATE instruction basically defines where the large data object has to be stored (in a named file, in a temporary file, or in memory). This instruction will actually allow you to fetch a LOB into memory or into a file, or insert a LOB from memory or from a file into the database. When located in a temporary file (IN FILE), the temp directory can be defined by the DBTEMP environment variable.

```
DEFINE t TEXT
LET t = "aaaa" -- invalid, t is not located
LOCATE t IN MEMORY
LET t = "aaaa" -- valid, now t is located in memory
```

With BYTE and TEXT types, you can insert/update/fetch large objects of the database. The native database type to be used depends from the type of database server. After defining the storage (LOCATE) of a large object handler, load / assign its value and use it directly in the SQL statements, or fetch data from LOB columns of the database, like simple data types:

```
DEFINE t1, t2 TEXT
...
CREATE TABLE mytable ( id INT, data TEXT )
...
LOCATE t1 IN MEMORY
CALL t1.readFile("lob.4gl")
INSERT INTO mytable VALUES ( 1, t1 )
LOCATE t2 IN FILE
SELECT data INTO t2 FROM mytable WHERE id=1
...
```

BYTE and TEXT types implement the readFile() and writeFile() methods to read/write the whole large object data from/to files. These methods can be used to easily interface with other software components:

```
DEFINE t TEXT
LOCATE t IN MEMORY
CALL t.readFile("orig.txt")
CALL t.writeFile("copy.txt")
```

When initializing a BYTE or TEXT variable to NULL (INITIALIZE *var* TO NULL), if the variable is located in a file, the file is truncated (file size will be zero). If the variable is located in memory, the data in memory

will be truncated. A subsequent usage of the variable (for example, FETCH INTO or LET assignment) is still possible:

```
DEFINE b BYTE
LOCATE b IN FILE "picture.png"
INITIALIZE b TO NULL
-- The file "picture.png" is now empty.
```

Resources allocated to a BYTE or TEXT variable can be deallocated with the FREE instruction. A FREE will remove the file if the LOB variable is located in a (named or temporary) file. When located in memory, the FREE instruction will de-allocate the memory. After freeing the resources of a LOB variable, it must be re-located with a LOCATE instruction:

```
DEFINE b BYTE
LOCATE b IN FILE
CALL t.readFile("picture.png") -- ok
FREE b
CALL t.readFile("picture.png") -- Invalid, b is not located.
LOCATE b IN MEMORY
CALL t.readFile("picture.png") -- ok
```

Important:

TEXT and BYTE are reference types. This implies that assigning two variables (LET, passing a variable as parameter to a function, returning a result from a function) does not copy the value (Only the handler is copied. As result, modifying the data with a TEXT/BYTE variable assigned from another TEXT/BYTE variable will in fact modify the same LOB data. Further, the storage resource (file or memory) that was used by the assigned variable becomes unreferenced and is lost:

```
DEFINE b1, b2 BYTE -- Could be TEXT: same behavior
LOCATE b1 IN FILE "mydata" -- reference file directly
LOCATE b2 IN MEMORY -- use memory instead of file
CALL b2.readFile("mydata") -- read file content into memory
# FREE b2 -- this should be done to free memory before LET
LET b2 = b1 -- Now b2 points directly to the file (like b1)
INITIALIZE b1 TO NULL -- truncates reference file
DISPLAY IIF( b2 IS NULL, "b2 is null", "b2 is not null")
-- Displays "b2 is null"
```

In the next (invalid) code example, we try to save the value of the `img BYTE` variable in a temporary variable (`tmp`), with the typical programming pattern to save the value before modification. In fact the `LET tmp=img` assignment does not copy the data of the LOB like for simple data types (STRING, VARCHAR, DECIMAL), only the reference (i.e. handle) to the data is copied:

```
-- WARNING: THIS IS AN INVALID CODE EXAMPLE
DEFINE img, tmp BYTE
LOCATE img IN MEMORY
CALL img.readFile("picture1.png")
LOCATE tmp IN MEMORY
LET tmp = img -- Expecting to save the current data, but now
              -- both variables reference the same data...
CALL img.readFile("picture2.png")
LET img = tmp -- Does not restore the old value: Same data.
```

If you need to clone a large object, use the `writeFile()` / `readFile()` methods.

BOOLEAN

The BOOLEAN data type stores a logical value, TRUE or FALSE.

Syntax

```
BOOLEAN
```

Usage

Boolean data types have two possible values: TRUE (integer 1) and FALSE (integer 0).

Variables of this type can be used to store the result of a boolean expression:

```
DEFINE result BOOLEAN
LET result = ( length("abcdef") > 0 )
```

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

Boolean variables are typically used to store the result of a boolean expression:

```
FUNCTION checkOrderStatus( cid )
  DEFINE oid INT, b BOOLEAN
  LET b = ( isValid(oid) AND isStored(oid) )
  IF NOT b THEN
    ERROR "The order is not ready."
  END IF
END FUNCTION
```

Note that the database vendor specific implementation of the boolean SQL type may not correspond exactly to the Genero BOOLEAN type. For example, IBM® Informix® SQL boolean type accepts the 't' and 'f' values, while the BOOLEAN Genero type expects 0/FALSE and 1/TRUE integer values only. You can however use a BOOLEAN variable in SQL statements: IBM® Informix® will handle the conversion, and for other databases, the db drivers handle the conversion. Note also that the TRUE/FALSE constants are Genero language constants: The SQL syntax of the database may not support these keywords, for example in an statement such as `INSERT INTO mytable (key,bcol) VALUES (455,TRUE)`. For more details, see SQL portability on page 412.

CHAR(size)

The CHAR data type is a fixed-length character string data type.

Syntax

```
CHAR[ACTER] [ (size) ]
```

1. *size* defines the maximum length of the character string, in byte or char units (depending on the character length semantics)
2. The maximum size of a CHAR type is 65534.
3. If no *size* is specified, it defaults to 1.

Usage

The CHAR type is typically used to store fixed-length character strings such as short codes (XB124), phone numbers (650-23-2345), vehicle identification numbers.

CHAR and CHARACTER are synonyms.

The *size* can be expressed in bytes or characters, depending on the length semantics used in programs. For more details about character length semantics, see Length semantics settings on page 314.

When *size* is not specified, the default length is 1.

CHAR variables are initialized to NULL in functions, modules and globals.

Text literals can be assigned to character string variables:

```
MAIN
  DEFINE c CHAR(10)
  LET c = "abcdef"
END MAIN
```

When assigning a non-NULL value, CHAR variables are always blank-padded:

```
MAIN
  DEFINE c CHAR(10)
  LET c = "abcdef"
  DISPLAY "[", c ,"]"   -- displays [abcdef    ]
END MAIN
```

Trailing blanks of a CHAR value are not significant in comparisons:

```
MAIN
  DEFINE c CHAR(5)
  LET c = "abc"
  IF c == "abc" THEN    -- evaluates to TRUE
     DISPLAY "equals"
  END IF
END MAIN
```

Numeric and date-time values can be directly assigned the character strings:

```
MAIN
  DEFINE c CHAR(50), da DATE, dec DECIMAL(10,2)
  LET da = TODAY
  LET dec = 345.12
  LET c = da, " : ", dec
END MAIN
```

When you insert character data from CHAR variables into CHAR columns in a database table, the column-value is blank-padded to the size of the column. Likewise, when you fetch CHAR column values into CHAR variables, the program variable is blank-padded to the size of the variable.

```
MAIN
  DEFINE c CHAR(10)
  DATABASE test1
  CREATE TABLE table1 ( k INT, x CHAR(10) )
  LET c = "abc"
  INSERT INTO table1 VALUES ( 1, c )
  SELECT x INTO c FROM table1 WHERE k = 1
  DISPLAY "[", vc ,"]"     -- displays [abc  ]
END MAIN
```

In SQL statements, the behavior of the comparison operators when using CHAR values may vary from one database to the other. However, most database engines ignore trailing blanks when compating CHAR values. For more details, see SQL portability on page 412.

DATE

The DATE data type stores calendar dates with a Year/Month/Day representation.

Syntax

```
DATE
```

Usage

Storage of DATE variables is based on a 4 byte integer representing the number of days since 1899/12/31.

The value range is from 0001-01-1 (-693594) to 9999-12-31 (2958464) .

DATE variables are initialized to zero (=1899/12/31) in functions, modules and globals.

Several built-in functions and operators specific to the DATE type are available, such as MDY( ) and TODAY. For more details, see Date and time operators on page 259.

Data type conversions, input and display of DATE values are ruled by environment settings, such as the DBDATE and DBCENTURY enviroment variables. Dates can be formatted with the USING operator. For more details, see Formatting DATE values on page 220.

> **Note:** As date-to-string conversion is based on an environment settings, it is not recommended that you hard code strings representing dates:
>
> ```
> LET date_var = "24/12/1998"      -- DBDATE dependant code
> LET date_var = MDY(12,24,1998)   -- Portable code
> ```

To add or substract a given number of days to a DATE, simply use a + or - arithmetic operator followed by an integer expression representing a number of days:

```
MAIN
  DEFINE d DATE
  LET d = TODAY
  LET d = d + 10   -- Add 10 days
  LET d = d - 20   -- Substract 20 days
  DISPLAY "d = ", d USING "yyyy-mm-dd"
END MAIN
```

The difference of two dates returns the number of days:

```
MAIN
  DEFINE d1, d2 DATE
  LET d1 = MDY(12,24,1998)
  LET d2 = MDY(5,11,2010)
  DISPLAY "d2 - d1 = ", (d2-d1)
END MAIN
```

DATE values can be converted directly from/to DATETIME values:

```
MAIN
  DEFINE d DATE,
         dt DATETIME YEAR TO FRACTION(3)
  LET d = TODAY
  LET dt = d;   DISPLAY "dt = ", dt
  LET dt = CURRENT
  LET d = dt;   DISPLAY "d = ", d
END MAIN
```

In order to add or substract a number of months to a DATE, use the UNITS operator:

```
MAIN
    DEFINE d0, d date
    LET d0 = MDY(01, 31, 2015)
    LET d = d0 + 1 UNITS MONTH; DISPLAY d
    LET d = d0 - 1 UNITS MONTH; DISPLAY d
    LET d = d0 - 2 UNITS MONTH; DISPLAY d
END MAIN
```

Note: In fact, the UNITS operator will produce an INTERVAL. Then the DATE value is converted to a DATETIME, to add or substract the INTERVAL value. Finally the DATETIME is converted to a DATE, in order to assign the result to the target variable.

DATETIME qual1 TO qual2

The DATETIME data type stores date and time data with time units from the year to fractions of a second.

Syntax

```
  DATETIME YEAR TO FRACTION [ ( scale ) ]
| DATETIME YEAR TO SECOND
| DATETIME YEAR TO MINUTE
| DATETIME YEAR TO HOUR
| DATETIME YEAR TO DAY
| DATETIME YEAR TO MONTH
| DATETIME YEAR TO YEAR
| DATETIME MONTH TO FRACTION [ ( scale ) ]
| DATETIME MONTH TO SECOND
| DATETIME MONTH TO MINUTE
| DATETIME MONTH TO HOUR
| DATETIME MONTH TO DAY
| DATETIME MONTH TO MONTH
| DATETIME DAY TO FRACTION [ ( scale ) ]
| DATETIME DAY TO SECOND
| DATETIME DAY TO MINUTE
| DATETIME DAY TO HOUR
| DATETIME DAY TO DAY
| DATETIME HOUR TO FRACTION [ ( scale ) ]
| DATETIME HOUR TO SECOND
| DATETIME HOUR TO MINUTE
| DATETIME HOUR TO HOUR
| DATETIME MINUTE TO FRACTION [ ( scale ) ]
| DATETIME MINUTE TO SECOND
| DATETIME MINUTE TO MINUTE
| DATETIME SECOND TO FRACTION [ ( scale ) ]
| DATETIME SECOND TO SECOND
| DATETIME FRACTION TO FRACTION [ ( scale ) ]
```

1. *scale* defines the scale of the fractional part, it can be 1, 2, 3, 4 or 5.

Usage

The DATETIME data type stores an instance in time, expressed as a calendar date and time-of-day.

The qualifiers following the DATETIME keyword define the precision of the DATETIME type. While many sort of datetime types can be defined with all possible qualifier combinations, only a limited set of DATETIME types are typical used in applications:

- DATETIME HOUR TO MINUTE, DATETIME HOUR TO SECOND, DATETIME HOUR TO FRACTION(scale): To hold a time value.

- `DATETIME YEAR TO MINUTE, DATETIME YEAR TO SECOND, DATETIME YEAR TO FRACTION(scale)`: To hold a date with time value.

`DATETIME YEAR TO DAY` is equivalent to `DATE`, consider used `DATE` instead.

When the `FRACTION` qualifier is specified without a precision, the precision defaults to 3.

`DATETIME` arithmetic is based on the `INTERVAL` data type, and can be combined with `DATE` values:

Table 97: Datetime Arithmetic operators

Left Operand Type	Operator	Right Operand Type	Result Type
DATETIME	-	DATETIME	INTERVAL
DATETIME	-	DATE	INTERVAL
DATETIME	-	INTERVAL	DATETIME
DATETIME	+	INTERVAL	DATETIME

`DATETIME` variables are initialized to `NULL` in functions, modules and globals.

The `CURRENT` operator provides current system date/time:

```
DEFINE dt DATETIME YEAR TO SECOND
LET dt = CURRENT
```

`DATETIME` variables can be assigned with datetime literals, by using the `DATETIME() q1 TO q2` notation:

```
DEFINE dt DATETIME YEAR TO SECOND
LET dt = DATETIME(2014-02-21 13:45:34) YEAR TO SECOND
```

`DATETIME` variables can be assigned from string literals, by using the format `YYYY-MM-DD hh:mm:ss.fffff`, or the ISO 8601 format sub-set (with the T separator between the date and time part, and with optional `+/-nn` UTC indicator or timezone offset):

```
DEFINE dt DATETIME YEAR TO FRACTION(5)
LET dt = "2012-10-05 11:34:56.99999"
LET dt = "2012-10-05T11:34:56.99999+02:00"
```

When converting a `DATETIME` to a string, the format `YYYY-MM-DD hh:mm:ss.fffff` is used.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

Datetime conversion functions are provided in the `util.Datetime` class, for example to convert local datetime to UTC datetime values:

```
IMPORT util
MAIN
    DEFINE dt DATETIME YEAR TO FRACTION(5)
    LET dt = "2012-10-05 11:34:56.99999"
    DISPLAY util.Datetime.toUTC( dt )
END MAIN
```

DECIMAL(p,s)

The `DECIMAL` data type is provided to handle large numeric values with exact decimal storage.

Syntax

```
DECIMAL [ ( precision[,scale]) ]
```

1. *precision* defines the number of significant digits (limit is 32, default is 16).
2. *scale* defines the number of digits to the right of the decimal point.
3. When no *scale* is specified, the data type defines a floating point number.
4. When no (*precision*, *scale*) is specified, it defaults to `DECIMAL(16)`.

Usage

Use the `DECIMAL` data type when you need to store values that have fixed number of digits on the right and left of the decimal point (`DECIMAL(p,s)`), or to store a floating point decimal with an exact number of significant digits (`DECIMAL(p)`).

`DEC`, `DECIMAL` and `NUMERIC` are synonyms.

`DECIMAL` variables are initialized to `NULL` in functions, modules and globals.

When using `DECIMAL(p,s)` with a precision and scale, you define a decimal for fixed point arithmetic, with p significant digits and s digits on the right of the decimal point. For example, `DECIMAL(8,2)` can hold the value 123456.78 (8 (p) = 6 digits on the left + 2 (s) digits of the right of the decimal point).

When using `DECIMAL(p)` with a precision but no scale, you define a floating-point number with p significant digits. For example, `DECIMAL(8)` can store 12345678, as well as 0.12345678.

> **Note:** In most database implementations, the decimal data type always has a fixed number of decimal digits. Use `DECIMAL` types with precision and scale to implement portable code, and avoid mistakes if default sizes apply when precisions and/or scale are omitted in SQL statements. For example, with Oracle, a `NUMBER(p)` is equivalent to a `DECIMAL(p,0)` in BDL, not `DECIMAL(p)`.

When using `DECIMAL` without a precision and scale, it defaults to `DECIMAL(16)`, a floating-point number with a precision of 16 digits.

```
MAIN
  DEFINE d1 DECIMAL(10,4)
  DEFINE d2 DECIMAL(10,3)
  LET d1 = 1234.4567
  LET d2 = d1 / 3 -- Rounds decimals to 3 digits
  DISPLAY d1, d2
END MAIN
```

`DECIMAL` values can be converted to strings according to the DBFORMAT (or DBMONEY) environment variable (defines the decimal separator).

Value ranges

The largest absolute value that a `DECIMAL(p,s)` can store without errors is $10^{p-s} - 10^{s}$. The stored value can have up to 30 significant decimal digits in its fractional part, or up to 32 digits to the left of the decimal point.

When using `DECIMAL(p,s)` the range of values is defined by the p, the number of significant digits. For example, a variable defined as `DECIMAL(5,3)` can store values in the range -99.999 to 99.999. The smallest positive non zero value is 0.001.

When using `DECIMAL(p)` the magnitude can range from -N*10^{-124} to N*10^{124}, where N can have up to p significant digits and be 0<N<10. For example, a variable defined as `DECIMAL(5)` can store values in the range -9.9999e-124 to 9.9999e+124. The smallest positive non zero value is 9.9999e-130.

Exceptions

When the default exception handler is used, if you try to assign a value larger than the decimal definition (for example, 12345.45 into `DECIMAL(4,2)`), no out of range error occurs, and the variable is assigned with `NULL`. If `WHENEVER ANY ERROR` is used, it raises error -1226. If you do not use `WHENEVER ANY ERROR`, the STATUS variable is not set to -1226.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

Computation and rounding rules

When computing or converting decimal values, the "round half away from zero" rule will apply: If the fraction of the value v is exactly 0.5, then r = v + 0.5 if v is positive, and r = v - 0.5 if v is negative. For example, when the result must be rounded to a whole number, 23.5 gets rounded to 24, and -23.5 gets rounded to -24.

In the next example, the division result of 11 / 3 gives the infinite decimal value 3.666666... (with an infinite decimal part). However, this value cannot be stored in a fixed point decimal type. When stored in a `DECIMAL(10,2)`, the value will be rounded to 3.67, and when multiplying 3.67 by 3, the result will be 11.01, instead of 11:

```
MAIN
    DEFINE v DECIMAL(10,2)
    LET v = 11 / 3
    DISPLAY "1. v = ", v USING "---&.&&&&&&&&"
    LET v = v * 3
    DISPLAY "2. v = ", v USING "---&.&&&&&&&&"
END MAIN
```

Output:

```
1. v =    3.67000000
2. v =   11.01000000
```

High-precision math functions

A couple of precision math functions are available, to be used with DECIMAL values. These functions have a higher precision as the standard C library functions based on C double data type, which is equivalent to `FLOAT`:

- FGL_DECIMAL_TRUNCATE()
- FGL_DECIMAL_SQRT()
- FGL_DECIMAL_EXP()
- FGL_DECIMAL_LOGN()
- FGL_DECIMAL_POWER()

FLOAT

The FLOAT data type stores values as double-precision floating-point binary numbers with up to 16 significant digits.

Syntax

```
FLOAT [(precision)]
```

1. FLOAT and DOUBLE PRECISION are synonyms.
2. The *precision* can be specified but it has no effect in programs.

Usage

The storage of FLOAT variables is based on 8 bytes of signed data (=64 bits), this type is equivalent to the double data type in C.

> **Note:** This data type it is not recommended for exact decimal storage; use the DECIMAL type instead.

FLOAT variables are initialized to zero in functions, modules and globals.

SMALLMONEY values can be converted to strings according to the DBFORMAT (or DBMONEY) environment variable.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

INTEGER

The INTEGER data type is used for storing large whole numbers.

Syntax

```
INTEGER
```

1. INT and INTEGER are synonyms.

Usage

The storage of INTEGER variables is based on 4 bytes of signed data (= 32 bits).

The value range is from -2,147,483,647 to +2,147,483,647.

INTEGER variables can be initialized with integer literals:

```
MAIN
  DEFINE i INTEGER
  LET i = 1234567
  DISPLAY i
END MAIN
```

When assigning a whole number that exceeds the INTEGER range, the overflow error -1215 will be raised.

INTEGER variables are initialized to zero in functions, modules and globals.

The INTEGER type can be used to define variables storing values from SERIAL columns.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

INTERVAL qual1 TO qual2

The INTERVAL data type stores spans of time as Year/Month or Day/Hour/Minute/Second/Fraction units.

Syntax 1: *year-month* class interval

```
  INTERVAL YEAR[(precision)] TO MONTH
| INTERVAL YEAR[(precision)] TO YEAR
| INTERVAL MONTH[(precision)] TO MONTH
```

Syntax 2: *day-time* class interval

```
  INTERVAL DAY[(precision)] TO FRACTION[(scale)]
| INTERVAL DAY[(precision)] TO SECOND
| INTERVAL DAY[(precision)] TO MINUTE
| INTERVAL DAY[(precision)] TO HOUR
| INTERVAL DAY[(precision)] TO DAY

| INTERVAL HOUR[(precision)] TO FRACTION[(scale)]
| INTERVAL HOUR[(precision)] TO SECOND
| INTERVAL HOUR[(precision)] TO MINUTE
| INTERVAL HOUR[(precision)] TO HOUR

| INTERVAL MINUTE[(precision)] TO FRACTION[(scale)]
| INTERVAL MINUTE[(precision)] TO SECOND
| INTERVAL MINUTE[(precision)] TO MINUTE

| INTERVAL SECOND[(precision)] TO FRACTION[(scale)]
| INTERVAL SECOND[(precision)] TO SECOND

| INTERVAL FRACTION TO FRACTION[(scale)]
```

1. *precision* defines the number of significant digits of the first qualifier, it must be an integer from 1 to 9. For YEAR, the default is 4. For all other time units, the default is 2. For example, YEAR(5) indicates that the INTERVAL can store a number of years with up to 5 digits.
2. *scale* defines the scale of the fractional part, it can be 1, 2, 3, 4 or 5.

Usage

The INTERVAL data type stores a span of time, the difference between two points in time. It can also be used to store quantities that are measured in units of time, such as ages or times.

The INTERVAL data type falls in two classes, which are mutually exclusive:

- *Year-time* intervals store a span of years, months or both.
- *Day-time* intervals store a span of days, hours, minutes, seconds and fraction of seconds, or a contiguous subset of those units.

INTERVAL values can be negative.

INTERVAL variables are initialized to NULL in functions, modules and globals.

INTERVAL variables can be assigned from interval literals, by using the INTERVAL() q1 TO q2 notation:

```
DEFINE iv INTERVAL DAY(5) TO SECOND
LET iv = INTERVAL(-7634 14:23:55) DAY(5) TO SECOND
```

INTERVAL variables can be assigned from string literals, by using the format YYYY-MM or DD hh:mm:ss.fffff, according to the interval class:

```
DEFINE iv INTERVAL DAY(5) TO SECOND
```

```
LET iv = "-7634 14:23:55"
```

INTERVAL variables defined with a single time unit can be assigned from integer values, by using the UNITS operator:

```
DEFINE iv INTERVAL SECOND(5) TO SECOND
LET iv = 567 UNITS SECOND
```

Intervals are typically used for DATETIME computation. According to the arithmetic operator, DATETIME or DECIMAL operands are involved:

Table 98: Arithmetic operands for the INTERVAL, DATETIME, and DECIMAL data types

Left Operand Type	Operator	Right Operand Type	Result Type
INTERVAL	*	DECIMAL	INTERVAL
INTERVAL	/	DECIMAL	INTERVAL
INTERVAL	-	INTERVAL	INTERVAL
INTERVAL	+	INTERVAL	INTERVAL
DATETIME	-	INTERVAL	DATETIME
DATETIME	+	INTERVAL	DATETIME
DATETIME	-	DATETIME	INTERVAL

The next example shows how to use INTERVAL with DATETIME variables:

```
MAIN
  DEFINE iym1, iym2 INTERVAL YEAR TO MONTH,
         dt1, dt2 DATETIME YEAR TO MINUTE,
         diff INTERVAL DAY(5) TO MINUTE
  LET iym1 = "2342-4"
  LET iym2 = "-55-11"
  DISPLAY iym1 + iym2
  LET dt1 = CURRENT
  LET dt2 = "2010-12-24 00:00"
  LET diff = dt1 - dt2
  DISPLAY diff
  LET diff = INTERVAL(-7634 14:23) DAY(5) TO MINUTE
  DISPLAY diff
END MAIN
```

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

MONEY(p,s)

The MONEY data type is provided to store currency amounts with exact decimal storage.

Syntax

```
MONEY [ (precision[,scale]) ]
```

1. *precision* defines the number of significant digits (limit is 32, default is 16).
2. *scale* defines the number of digits to the right of the decimal point.
3. When no *scale* is specified, it defaults to 2.
4. When no (*precision*, *scale*) is specified, it defaults to MONEY(16,2).

Usage

The MONEY data type is provided to store currency amounts. Its behavior is similar to the DECIMAL data type, with some important differences:

A MONEY variable is displayed with the currency symbol defined in the DBFORMAT (or DBMONEY) environment variable.

You cannot define floating-point numbers with MONEY: If you do not specific the *scale* in the data type declaration, it defaults to 2. A MONEY without *precision* and *scale* defaults to MONEY(16,2), which is equivaltent to a DECIMAL(16,2).

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

See DECIMAL(p,s) on page 200 to learn other facts about the MONEY(p,s) data type.

SMALLFLOAT

The SMALLFLOAT data type stores values as single-precision floating-point binary numbers with up to 8 significant digits.

Syntax

```
SMALLFLOAT
```

1. SMALLFLOAT and REAL are synonyms.

Usage

The storage of SMALLFLOAT variables is based on 4 bytes of signed data (=32 bits), this type is equivalent to the `float` data type in C.SMALLFLOAT variables are initialized to zero in functions, modules and globals.

> **Note:** This data type it is not recommended for exact decimal storage; use the DECIMAL data type instead.

SMALLMONEY values can be converted to strings according to the DBFORMAT (or DBMONEY) environment variable.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

SMALLINT

The SMALLINT data type is used for storing small whole numbers.

Syntax

```
SMALLINT
```

Usage

The storage of SMALLINT variables is based on 2 bytes of signed data (= 16 bits).

The value range is from -32,767 to +32,767.

SMALLINT variables can be initialized with integer literals:

```
MAIN
  DEFINE i SMALLINT
  LET i = 1234
  DISPLAY i
```

```
END MAIN
```

When assigning a whole number that exceeds the SMALLINT range, the overflow error -1214 will be raised.

SMALLINT variables are initialized to zero in functions, modules and globals.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

STRING

The STRING data type is a variable-length, dynamically allocated character string data type, without limitation.

Syntax

```
STRING
```

Usage

The STRING data type is typically used to implement utility functions manipulating character string with unknown size, and in some special cases, in SQL statements.

STRING variables are initialized to NULL in functions, modules and globals.

The behavior of a STRING variable is similar to the VARCHAR data type, except that there is no theoretical size limit.

STRING variables can be initialized from string literals:

```
MAIN
    DEFINE s STRING
    LET s = "abcdef"
END MAIN
```

Variables declared with the STRING data type can be used to call STRING-type methods such as getLength() or toUpperCase(). For more details, see The STRING data type as class on page 1689:

```
MAIN
    DEFINE s STRING
    LET s = "abc"
    DISPLAY s.toUpperCase()
END MAIN
```

STRING variables have significant trailing blanks (i.e. "abc " is different from "abc"). However, in comparisons, trailing blancs do not matter:

```
MAIN
    DEFINE s STRING
    LET s = "abc  " -- a b c + 2 white spaces
    DISPLAY "1: s.length:", s.getLength()
    DISPLAY "[", s, "]" -- displays "[abc  ]"
    DISPLAY IIF(s=="abc","Equals",NULL)
END MAIN
```

Unlike CHAR and VARCHAR, a STRING can hold a value of zero length without being NULL. For example, if you trim a string variable with the trim() method and if the original value is a set of blank characters, the

result is an empty string. But testing the variable with the `IS NULL` operator will evaluate to `FALSE`. Using a `VARCHAR` with the `CLIPPED` operator would give a `NULL` string in this case:

```
MAIN
    DEFINE s STRING
    LET s = "     " -- 5 spaces
    LET s = s.trim()
    DISPLAY "s = [", s, "] len=", s.getLength()
    DISPLAY IIF(s IS NULL, "NULL", "not NULL")
END MAIN

outputs:

s = [] len=          0
not NULL
```

`STRING` typed variables can be used in some special cases to hold SQL character string data, when the size of the SQL data string is not known (string expressions, large strings like JSON documents). In order to store character string data stored in a database, consider using the `CHAR` or `VARCHAR` types instead of `STRING`.

In `STRING` methods, positions and length parameters (or return values) can be expressed in bytes or characters, depending on the length semantics used in programs. For more details, see Length semantics settings on page 314

TINYINT

The `TINYINT` data type is used for storing very small whole numbers.

Syntax

```
TINYINT
```

Usage

The storage of `TINYINT` variables is based on 1 byte of signed data (= 8 bits).

The value range is from -128 to +127.

`TINYINT` variables can be initialized with integer literals:

```
MAIN
  DEFINE i TINYINT
  LET i = 101
  DISPLAY i
END MAIN
```

When assigning a whole number that exceeds the `TINYINT` range, the overflow error -8097 will be raised.

`TINYINT` variables are initialized to zero in functions, modules and globals.

The `TINYINT` variables cannot be `NULL`.

Data type conversion can be controlled by catching the runtime exceptions. For more details, see Handling type conversion errors on page 216.

TEXT

The TEXT data type stores large text data.

Syntax

```
TEXT
```

Usage

A BYTE or TEXT variable is a handle for a large object (LOB), that is stored in a file or in memory. Such data type is a complex type that cannot be used like INTEGER or CHAR basic types: It is designed to handle a large amount of data and has different semantics as simple types. The main difference with simple data types, is the fact that you must specify the storage with the LOCATE instruction, before using BYTE and TEXT variables.

The maximum size of data that can be handled by BYTE and TEXT variable is theoretically 2^31 bytes (~2.14 Gigabytes), but the practical limit depends from the disk or memory resources available to the process.

BYTE and TEXT variable must be initialized with the LOCATE instruction before usage. The LOCATE instruction basically defines where the large data object has to be stored (in a named file, in a temporary file, or in memory). This instruction will actually allow you to fetch a LOB into memory or into a file, or insert a LOB from memory or from a file into the database. When located in a temporary file (IN FILE), the temp directory can be defined by the DBTEMP environment variable.

```
DEFINE t TEXT
LET t = "aaaa" -- invalid, t is not located
LOCATE t IN MEMORY
LET t = "aaaa" -- valid, now t is located in memory
```

With BYTE and TEXT types, you can insert/update/fetch large objects of the database. The native database type to be used depends from the type of database server. After defining the storage (LOCATE) of a large object handler, load / assign its value and use it directly in the SQL statements, or fetch data from LOB columns of the database, like simple data types:

```
DEFINE t1, t2 TEXT
...
CREATE TABLE mytable ( id INT, data TEXT )
...
LOCATE t1 IN MEMORY
CALL t1.readFile("lob.4gl")
INSERT INTO mytable VALUES ( 1, t1 )
LOCATE t2 IN FILE
SELECT data INTO t2 FROM mytable WHERE id=1
...
```

BYTE and TEXT types implement the readFile() and writeFile() methods to read/write the whole large object data from/to files. These methods can be used to easily interface with other software components:

```
DEFINE t TEXT
LOCATE t IN MEMORY
CALL t.readFile("orig.txt")
CALL t.writeFile("copy.txt")
```

When initializing a BYTE or TEXT variable to NULL (INITIALIZE *var* TO NULL), if the variable is located in a file, the file is truncated (file size will be zero). If the variable is located in memory, the data in memory

will be truncated. A subsequent usage of the variable (for example, FETCH INTO or LET assignment) is still possible:

```
DEFINE b BYTE
LOCATE b IN FILE "picture.png"
INITIALIZE b TO NULL
-- The file "picture.png" is now empty.
```

Resources allocated to a BYTE or TEXT variable can be deallocated with the FREE instruction. A FREE will remove the file if the LOB variable is located in a (named or temporary) file. When located in memory, the FREE instruction will de-allocate the memory. After freeing the resources of a LOB variable, it must be re-located with a LOCATE instruction:

```
DEFINE b BYTE
LOCATE b IN FILE
CALL t.readFile("picture.png") -- ok
FREE b
CALL t.readFile("picture.png") -- Invalid, b is not located.
LOCATE b IN MEMORY
CALL t.readFile("picture.png") -- ok
```

Important:

TEXT and BYTE are reference types. This implies that assigning two variables (LET, passing a variable as parameter to a function, returning a result from a function) does not copy the value (Only the handler is copied. As result, modifying the data with a TEXT/BYTE variable assigned from another TEXT/BYTE variable will in fact modify the same LOB data. Further, the storage resource (file or memory) that was used by the assigned variable becomes unreferenced and is lost:

```
DEFINE b1, b2 BYTE -- Could be TEXT: same behavior
LOCATE b1 IN FILE "mydata" -- reference file directly
LOCATE b2 IN MEMORY -- use memory instead of file
CALL b2.readFile("mydata") -- read file content into memory
# FREE b2 -- this should be done to free memory before LET
LET b2 = b1 -- Now b2 points directly to the file (like b1)
INITIALIZE b1 TO NULL -- truncates reference file
DISPLAY IIF( b2 IS NULL, "b2 is null", "b2 is not null")
-- Displays "b2 is null"
```

In the next (invalid) code example, we try to save the value of the `img BYTE` variable in a temporary variable (`tmp`), with the typical programming pattern to save the value before modification. In fact the `LET tmp=img` assignment does not copy the data of the LOB like for simple data types (STRING, VARCHAR, DECIMAL), only the reference (i.e. handle) to the data is copied:

```
-- WARNING: THIS IS AN INVALID CODE EXAMPLE
DEFINE img, tmp BYTE
LOCATE img IN MEMORY
CALL img.readFile("picture1.png")
LOCATE tmp IN MEMORY
LET tmp = img -- Expecting to save the current data, but now
              -- both variables reference the same data...
CALL img.readFile("picture2.png")
LET img = tmp -- Does not restore the old value: Same data.
```

If you need to clone a large object, use the `writeFile()` / `readFile()` methods.

It is possible to assign TEXT variables to/from VARCHAR, CHAR and STRING variables.

VARCHAR(size)

The VARCHAR data type is a variable-length character string data type, with a maximum size.

Syntax

```
VARCHAR [ ( size [,reserve] ) ]
```

1. *size* defines the maximum length of the character string, in byte or char units (depending on the character length semantics)
2. The maximum size of a VARCHAR type is 65534.
3. When no *size* is specified, it defaults to 1.
4. *reserve* is ignored; Its inclusion in the syntax is permitted for compatibility with the SQL data type.

Usage

The VARCHAR type is typically used to store variable-length character strings such as names, addresses and comments.

The *size* can be expressed in bytes or characters, depending on the length semantics used in programs. For more details about character length semantics, see Length semantics settings on page 314.

When *size* is not specified, the default length is 1.

VARCHAR variables are initialized to NULL in functions, modules and globals.

Text literals can be assigned to character string variables:

```
MAIN
  DEFINE c VARCHAR(10)
  LET c = "abcdef"
END MAIN
```

VARCHAR variables store trailing blanks (trailing blanks are displayed or printed in reports, and stored in database columns):

```
MAIN
  DEFINE vc VARCHAR(10)
  LET vc = "abc  "       -- a b c + 2 white spaces
  DISPLAY "[", vc ,"]"   -- displays [abc  ]
END MAIN
```

Trailing blanks of a VARCHAR value are not significant in comparisons:

```
MAIN
  DEFINE vc VARCHAR(10)
  LET vc = "abc  "        -- a b c + 2 white spaces
  IF vc == "abc " THEN    -- evaluates to TRUE
     DISPLAY "equals"
  END IF
END MAIN
```

Numeric and date-time values can be directly assigned the character strings:

```
MAIN
  DEFINE vc VARCHAR(50), da DATE, dec DECIMAL(10,2)
  LET da = TODAY
  LET dec = 345.12
  LET vc = da, " : ", dec
END MAIN
```

When you insert character data from VARCHAR variables into VARCHAR columns in a database table, the trailing blanks are kept. Likewise, when you fetch VARCHAR column values into VARCHAR variables, trailing blanks are kept.

```
MAIN
  DEFINE vc VARCHAR(10)
  DATABASE test1
  CREATE TABLE table1 ( k INT, x VARCHAR(10) )
  LET vc = "abc  "         -- two trailing blanks
  INSERT INTO table1 VALUES ( 1, vc )
  SELECT x INTO vc FROM table1 WHERE k = 1
  DISPLAY "[", vc ,"]"     -- displays [abc  ]
END MAIN
```

In SQL statements, the behavior of the comparison operators when using VARCHAR values differs from one database to the other. IBM® Informix® is ignoring trailing blanks, but most other databases take trailing blanks of VARCHAR values into account. For more details, see SQL portability on page 412.

Type conversions

Explains data type conversion rules of the language.

- When does type conversion occur? on page 211
- Data type conversion reference on page 212
- Handling type conversion errors on page 216
- Formatting numeric values on page 217
- Formatting DATE values on page 220
- Formatting DATETIME values on page 221
- Formatting INTERVAL values on page 223

When does type conversion occur?

The runtime system performs data conversion implicitly without objection, as long as the data conversion is valid. A date value can be converted to a character string, but a character string can only be converted to a date if the string represents a valid date in the current date format settings (DBDATE).

Implicit data type conversion can for example occur in the following cases:

- In a `LET` assignment,
- In an expression, when operands are not of the same data type,
- In `DISPLAY` instructions, or `PRINT` instructions in reports,
- In dialogs, when values must be converted to strings to be displayed in form fields,
- When passing and returning values to/from a function,
- When serializing numeric values in `UNLOAD`, JSON methods, etc.

In the next code example, implicit data type conversion occurs

1. When assiging the result of the decimal expression to the variable `v`,
2. When passing the decimal value `d` to function `func()`,
3. When returing the varchar value from the function `func()`,
4. In the `DISPLAY` instruction, to convert the decimal to a string.

```
MAIN
  DEFINE d DECIMAL(10,2), v VARCHAR(50)
  LET v = 1234.50 * 2   -- 1.
  LET d = func(d)       -- 2. and 3.
  DISPLAY d             -- 4.
END MAIN
```

```
FUNCTION func(v)
  DEFINE v VARCHAR(50)
  DISPLAY v
  RETURN v   -- 3.
END FUNCTION
```

Data type conversion reference

Boolean type conversions

A BOOLEAN value is an integer value 1 or 0 and thus can be converted to/from any other numeric type of the language.

When converting a numeric value to a BOOLEAN, any value different from 0 becomes TRUE, otherwise (zero) is FALSE.

```
DEFINE hasContent BOOLEAN, s STRING
LET s = "abc"
LET hasContent = s.getLength()
```

When converting a string (CHAR, VARCHAR or STRING) to BOOLEAN, the string will be converted to a number first, then the number-to-boolean conversion applies. If the string value cannot convert to a numeric value (for example, "abc"), the boolean value becomes NULL.

When converting a BOOLEAN to a string, the result will be "1" or "0" string values, according to the boolean value.

Large object type conversions

A TEXT value can be converted to/from CHAR, VARCHAR or STRING.

The BYTE type cannot be converted to/from any other type.

Integers to decimal types

TINYINT, SMALLINT, INTEGER and BIGINT values can be converted to SMALLFLOAT, FLOAT, DECIMAL or MONEY as long as the decimal type is defined with sufficient digits to hold the whole number.

If the integer value exceeds the range of the receiving data type, an overflow error occurs.

Decimal to integer types

When converting a SMALLFLOAT, FLOAT DECIMAL or MONEY to a TINYINT, SMALLINT, INTEGER or BIGINT, the fractional part of the decimal value is truncated.

```
MAIN
  DEFINE d DECIMAL(10,2),
         i INTEGER
  LET d = 123.45
  LET i = d
  DISPLAY i   -- displays 123
END MAIN
```

If the decimal value exceeds the range of the receiving integer data type, an overflow error occurs.

Decimal to decimal types

Converting between SMALLFLOAT, FLOAT DECIMAL or MONEY types is allowed as long as the receiving type is defined with sufficient digits to hold the whole part of the original value.

If the original value contains more fractional digits than the receiving data type supports, low-order digits are discarded.

```
MAIN
  DEFINE d1 DECIMAL(10,2),
         d2 DECIMAL(5,1)
  LET d1 = 123.45
  LET d2 = d1
  DISPLAY d2    -- displays 123.5
END MAIN
```

Decimal to character types

Converting `SMALLFLOAT`, `FLOAT` `DECIMAL` or `MONEY` values to `CHAR`, `VARCHAR` and `STRING` implies numeric formatting.

Numeric formatting is controlled by the DBMONEY and DBFORMAT environment variables.

According to the conversion context, the resulting string is left-aligned (for lossless conversions) or right-aligned (for visual conversions), and the decimal part is kept, according to the numeric type.

```
MAIN
  DEFINE m MONEY(8,2),
         s VARCHAR(10)
  LET m = 123.45
  LET s = m   -- Lossless conversion "$123.45"
  DISPLAY m   -- Visual conversion   "    $123.45"
END MAIN
```

Fixed point decimals (`DECIMAL(p,s)`) are converted to strings that can fit in a `CHAR(p+2)`: The string is build with up to p significant digits + 1 character for the sign + 1 character for the decimal point. The result of a `DECIMAL(p,s)` to string conversion is never longer than p + 2 characters. For example, a `DECIMAL(5,2)` can produce "-999.99" (5 + 2 = 7c).

Floating point decimals (`DECIMAL(p)`) are converted to strings that can fit in a `CHAR(p+7)`: The string is build with up to p significant digits + 1 character for the sign + 1 character for the decimal point + the length of the exponent of needed ("e+123"). The result of a `DECIMAL(p)` to string conversion is never longer than p + 7. For example, a `DECIMAL(5)` can produce "-1.2345e-123" (5 + 7 = 12c).

`DECIMAL` to string conversion depends on the context in which the conversion occurs:

1. *Visual conversion*: The result of this conversion will typically be presented to the end user. This conversion happens in `DISPLAY`, `MESSAGE`, `ERROR`, `PRINT`. The result of a visual conversion is right aligned (padded with leading blanks). This padding results in the same length for any value for a given decimal precision. The length of the result is the maximum possible length as described previously (p+2 for `DECIMAL(p,s)`, p+7 for `DECIMAL(p)`).

   Visual conversion examples for `DECIMAL(5,2)`:

   ```
      Values     1234567
   ----------------------
      0       |  "   0.00"
      -999.99 | "-999.99"
      12.3    |  "  12.30"
      12.34   |  "  12.34"
   ```

   Visual conversion examples for `DECIMAL(5)`:

   ```
      Values     123456789012
   ---------------------------
      0       |  "         0.0"
      -99999  |  "    -99999.0"
   ```

```
   12.3    | "        12.3"
   12.34   | "       12.34"
   12.345  | "      12.345"
   1.23e7  | "  12300000.0"
   1e100   | "       1e100"
```

2. *Form field conversion*: This conversion concerns decimal numbers presented in form-fields. The result of this conversion is in best case the same as (1). The result of the conversion depends on the width of the form-field. If the width of the form-field is smaller than the perfect length, automatic rounding and exponential notation might be used.
3. *Lossless conversion*: Such conversion happens when assigning numbers to string variables (`LET`), passing numbers as parameters to functions expecting strings, returning numbers from functions to strings, serializing numbers (`UNLOAD`, XML or JSON APIs). These conversions must avoid the loss of significant digits. When using floating point decimals, this leads to a variable length of the resulting string. A conversion must be reversible: decimal to string to decimal must give the original value. If the target variable is shorter then the maximum possible length, then automatic rounding will occur.

   Lossless conversion examples of `DECIMAL(5,2)`:

```
   Values     1234567
----------------------
   0       | "0.00"
   -999.99 | "-999.99"
   12.3    | "12.30"
   12.34   | "12.34"
```

   Lossless conversion examples of `DECIMAL(5)`:

```
   Values     123456789012
----------------------------
   0       | "0.0"
   -99999  | "-99999.0"
   12.3    | "12.3"
   12.34   | "12.34"
   12.345  | "12.345"
   1.23e7  | "12300000.0"
   1e100   | "1e100"
```

Automatic rounding occurs if the target string variable is shorter than the maximum possible length of the `DECIMAL` type. Such conversion might loose significant digits: The runtime system tries to round the value, to fit into the target variable.

```
  Values     Different target sizes
              123456   12345   1234
-----------------------------------
  0.98765  | "0.9877" "0.988" "0.99"
  123.45   | "123.45" "123.5" "123"
```

Automatic switch to the exponential notation will occur if the integer part of the decimal value does not fit into the target string variable. For example, if the source variable is a `DECIMAL(12)` and the target variable is a `CHAR(9)`:

```
  Values         123456789
----------------------------
  1234567     | "1234567.0"
  12345678    | "12345678"
  123456789   | "123456789"
  1234567890  | "1.2346e10"
  12345678901 | "1.2346e11"
```

The exponential notation will also be used if the absolute value of a floating point decimal is less than 1e-8 (0.00000001).

Default formatting of floating point decimals has been revised with Genero 2.50. If DECIMAL(P)-to-string conversion must round to 2 digits, use the fglrun.decToCharScale2 FGLPROFILE entry:

```
fglrun.decToCharScale2 = true
```

> **Note:** Do not use the decToCharScale2 configuration parameter, unless you have migration issues.

Formatting a FLOAT is the same as DECIMAL(16). Any FLOAT value with up to 15 digits is exact. There is no precision loss when converting an exact FLOAT back and forth to/form a string. Some FLOAT values require 16, in some rare cases 17 digits for an exact string representation. 16 and 17 digits are not always exact: "8.000000000000001" and "8.000000000000002" represent the same float value.

Formatting a SMALLFLOAT is the same as DECIMAL(7). Any SMALLFLOAT value with up to 6 digits is exact. There is no precision loss when converting an exact SMALLFLOAT back and forth to/form a string. Some SMALLFLOAT values require 7, in some rare cases 8 digits for an exact string representation. 7 and 8 digits SMALLFLOAT are not always exact: "0.0009999901" and "0.0009999902" represent the same SMALLFLOAT value.

Character to decimal types

A CHAR, VARCHAR and STRING value can be converted to a TINYINT, SMALLINT, INTEGER, BIGINT, SMALLFLOAT, FLOAT DECIMAL or MONEY value as long as the character string value represents a valid number.

If the original value contains more significant digits or more fractional digits than the receiving data type supports, low-order digits are discarded.

```
MAIN
  DEFINE d DECIMAL(10,2)
  LET d = "-123.45"
  DISPLAY d   -- displays -123.45
  LET d = "1234567890123.45"
  DISPLAY d   -- displays null
  LET d = "12345678.999"
  DISPLAY d   -- displays 12345679.00
END MAIN
```

Date time to character types

Converting DATE, DATETIME and INTERVAL values to CHAR, VARCHAR and STRING implies date time formatting.

DATE formatting is controlled by the DBDATE environment variable.

When converting a DATETIME to a string, the YYYY-MM-DD hh:mm:ss.fffff standard format is used.

When converting an INTERVAL to a string, either YYYY-MM or DD hh:mm:ss.fffff standard formats are used, according to the interval class.

If the resulting is longer than the receiving variable, the resulting character string is null.

```
MAIN
  DEFINE d DATE,
         s VARCHAR(20),
         v VARCHAR(5)
  LET d = MDY(12,24,2012)
  LET s = d
  DISPLAY s   -- displays 12/24/2012
```

```
  LET v = d
  DISPLAY v   -- displays null
END MAIN
```

Character to date time types

Converting a CHAR, VARCHAR or STRING value to a DATE, DATETIME or INTERVAL is possible as long as the character string defines a well formatted date time or interval value.

When converting a character string to a DATE, the string must follow the date format defined by the DBDATE environment variable.

When converting a string to a DATETIME, the format must be YYYY-MM-DD hh:mm:ss.fffff or follow the ISO 8601 format sub-set (with the T separator between the date and time part, and with optional UTC indicator or timezone offset)

```
MAIN
  DEFINE dt DATETIME YEAR TO SECOND
  LET dt = "2012-12-24 11:33:45"
  DISPLAY dt   -- displays 2012-12-24 11:33:45
  LET dt = "2012-12-24T11:33:45+01:00"
  DISPLAY dt   -- displays 2012-12-24 11:33:45 (if TZ=UTC+1h)
  LET dt = "2012-12-24T10:33:45Z"
  DISPLAY dt   -- displays 2012-12-24 11:33:45 (if TZ=UTC+1h)
END MAIN
```

Converting DATE to/from DATETIME types

When converting a DATETIME to another DATETIME with a different precision, truncation from the left or right can occur. When then target type has more fields as the source type, the year, month and day fields are filled with the current date.

When converting a DATE to a DATETIME, the datetime fields are filled with year, month and day from the date value and time fields are set to zero.

When converting a DATETIME to a DATE, an implicit EXTEND(*datetime-value,* YEAR TO DAY) is performed.

```
MAIN
  DEFINE da DATE,
         dt1 DATETIME YEAR TO SECOND,
         dt2 DATETIME HOUR TO MINUTE
  LET da = MDY(12,24,2012)
  LET dt1 = da
  DISPLAY dt1   -- displays 2012-12-24 00:00:00
  LET dt2 = "23:45"
  LET dt1 = dt2
  DISPLAY dt1   -- displays <current date> 00:00:00
END MAIN
```

Unsupported type conversions

Other data type conversions not mentioned in this topic are not allowed and will result in a runtime error.

Handling type conversion errors

By default, in case of type conversion error or overflow errors, the program continues, the target variable is set to NULL and the global STATUS variable is not set.

In order to detect data conversion and overflow errors, use the WHENEVER ANY ERROR statement.

The next code example:

```
MAIN  -- DBDATE set to Y4MD-
  DEFINE v VARCHAR(50), d DATE
  LET v = "2012-99-99"      -- invalid date string
  LET d = v
  DISPLAY status, "/", NVL(d,"NULL")  -- displays 0/NULL
  WHENEVER ANY ERROR CONTINUE
  LET d = v
  DISPLAY status, "/", NVL(d,"NULL")  -- displays -1205/NULL
  WHENEVER ANY ERROR STOP
  LET d = "2012-11-23"  -- valid date, ok
  DISPLAY status, "/", NVL(d,"NULL")  -- displays 0/2012-11-23
  LET d = v   -- program execution stopped with error -1205
END MAIN
```

Above code will produce the following output:

```
          0/NULL
      -1218/NULL
Program stopped at 'x.4gl', line number 10.
FORMS statement error number -1218.
String to date conversion error.
```

Conversion and overflow errors are implicitly trapped in TRY/CATCH blocks.

In the next example, the `INTERVAL` variable is not large enough to hold the result of `d2 - d1`:

```
MAIN
  DEFINE d1, d2 DATETIME YEAR TO FRACTION(5)
  DEFINE i INTERVAL SECOND(2) TO SECOND
  LET d1 = "2015-11-06 17:40:21.436"
  LET d2 = "2015-11-06 10:40:21.436"
  TRY
      LET i = d2 - d1
  CATCH
      DISPLAY STATUS, " / ", err_get(STATUS)
  END TRY
END MAIN
```

Above code will produce the following output:

```
      -1265 / Overflow occurred on a datetime or interval operation.
```

Formatting numeric values

When does numeric formatting take place?

Numeric formatting occurs when converting a number to a string with the `USING` operator, for example in a `LET`, `DISPLAY` or `PRINT` instruction, and when displaying numeric values in form fields defined with the `FORMAT` attribute.

Numeric values can be of type such as `INTEGER`, `FLOAT`, `DECIMAL`, `MONEY`, etc.

This example formats a `DECIMAL(10,2)` value with the `USING` operator:

```
MAIN
  DEFINE d DECIMAL(10,2)
  LET d = -123456.78
  DISPLAY d USING "-,---,--&.&& @"
END MAIN
```

Front currency symbol, thousands separator, decimal separator and back currency symbol are defined with the DBFORMAT (or DBMONEY) environment variable. For example, if DBFORMAT is defined as `":.:,:E"`, the previous code example will produce the following output:

```
-123,456.78 E
```

Default formatting occurs when `USING` or `FORMAT` are not used, and a numeric value has to be converted to a character string, for example when passing a `DECIMAL(p,s)` to a function expecting a `VARCHAR(n)`. For more details about default formatting, see Data type conversion reference on page 212.

This topic describes the syntax of the *format-string* in the `USING "format-string"` operator and `FORMAT = "format-string"` form field attribute.

Formatting symbols for numbers

When formatting numeric values, the *format-string* of the `USING` operator or `FORMAT` attribute consists of a set of place holders that represent digits, currency symbols, thousands and decimal separators. For example, `"###.##@"` defines three places to the left of the decimal point and exactly two to the right, plus a "back" currency symbol at the end of the string.

> **Note:** The `USING` operator or `FORMAT` attribute are required to display the thousands separator defined in DBFORMAT.

The *format-string* must use normalized placeholders described in Table 99: Format-string symbols for Numeric data types on page 218. The placeholders will be replaced by digits, blanks or by the elements defined in the DBFORMAT (or DBMONEY) environment variables. Any other character will be interpreted as a literal, and can be used at any place in the format string.

If the numeric value is too large to fit in the number of characters defined by the format, the result string is filled with a set of star characters (`********`), indicating an overflow.

The minus sign (-), plus sign (+), parentheses (( )), and dollar sign ($) float. This means that when you specify multiple leading occurrences of one of these characters, the result string gets only a single character immediately to the left of the first digit.

Table 99: Format-string symbols for Numeric data types

Placeholder	Description
*	The star placeholder fills with asterisks any position that would otherwise be blank.
&	The ampersand placeholder is used to define the position of a digit, and is replaced by a zero if that position would otherwise be blank.
#	The sharp placeholder is used to define the position of a digit, it is used to specify a maximum width for the resulting string. The # is replaced by a blank, if no digit is to be displayed at that position.
<	Consecutive "less than" characters cause left alignment and define digit positions.
-	Displays a minus sign if the value is negative, or a blank if the value is positive. When you group several minus signs in the format string, a single minus sign floats immediately to the left of the first digit.
+	Displays a minus sign if the value is negative, or a plus sign if the value is positive. When you group several plus signs in the format string, a single plus sign floats immediately to the left of the first digit.
(	Displayed as left parenthesis for negative numbers. It is used to display "accounting parentheses" instead of a minus sign for negative numbers. Consecutive left parentheses display a single left parenthesis to the left of the first digit.

Placeholder	Description
)	Displayed as right parenthesis for negative numbers. This wildcard character is used in conjunction with a open brace to display "accounting parentheses" for negative numbers.
, (comma)	The comma placeholder is used to define the position for the thousand separator defined in DBFORMAT. The thousand separator will only be displayed if there is a number on the left of it.
. (period)	The period placeholder is used to define the position for the decimal separator defined in DBFORMAT. You can only have one decimal separator in a number format string.
$	The dollar sign is the placeholder for the front currency symbol defined in DBFORMAT. When you group several consecutive dollar signs, a single front currency symbol floats immediately to the left of the first digit. The front currency symbol can be defined in DBFORMAT with more than one character (EUR, USD).
@	The "at" sign is the placeholder for the back currency symbol defined in DBFORMAT. Put several consecutive @ signs at the end of the format string to display a currency symbol defined in DBFORMAT with more than one character.

Table 100: Numeric formatting examples

Format String	Numeric value	DBFORMAT	Result string
[######.##]	0	:.:,:	[,]
[######.##]	-1234.56	:.:,:	[1234,56] (no sign!)
[######.##]	-1234567.89	:.:,:	[*********] (overflow)
[######.##]	+1234.56	:.:,:	[1234,56]
[#####&.&&]	0	:.:,:	[0,00]
[******.**]	0	:.:,:	[******,00]
[******.**]	-12.34	:.:,:	[****12,34] (no sign!)
[******.**]	+12.34	:.:,:	[****12,34]
[<<<<<<.<<]	-12.34	:.:,:	[12,34] (no sign!)
[<<<<<<.<<]	+12.34	:.:,:	[12,34]
[---,--&.&&]	-1234.56	:.:,:	[-1.234,56]
[+++,++&.&&]	-1234.56	:.:,:	[-1.234,56]
[+++,++&.&&]	+1234.56	:.:,:	[+1.234,56]
[$---,--&.&&]	-1234.56	E:.:,:	[E -1.234,56]
[$---,--&.&&]	+1234.56	E:.:,:	[E 1.234,56]
[$$$---,--&.&&]	+1234.56	E:.:,:	[E 1.234,56]
[$$$---,--&.&&]	+1234.56	EUR:.:,:	[EUR 1.234,56]
[-,---,-$&.&&]	-12.34	E:.:,:	[-E12,34]
[-,---,-$&.&&]	-1234.56	E:.:,:	[-E1.234,56]
[-,-$$,$$&.&&]	-12.34	E:.:,:	[- E12,34]

Format String	Numeric value	DBFORMAT	Result string
[-,-$$,$$&.&&]	-1234.56	E:.:,:	[-E1.234,56]
[---,--&.&&@]	-1234.56	:.:,:E	[-1.234,56E]
[---,--&.&&@]	+1234.56	:.:,:E	[1.234,56E]
[---,--&.&&@@@]	+1234.56	:.:,:EUR	[1.234,56EUR]
[($---,--&.&&)]	-1234.56	E:.:,:	[(E -1.234,56)]
[($###,##&.&&)]	-1234.56	E:.:,:	[(E 1.234,56)]
[(((((,(($.&&)]	0	E:.:,:	[E,00]
[(((((,(($.&&)]	-12.34	E:.:,:	[(E12,34)] (no sign!)
[(((((,(($.&&)]	+12.34	E:.:,:	[E12,34]
[(((((,(($.&&)]	-1234.56	E:.:,:	[(E1.234,56)] (no sign!)
[(((((,(($.&&)]	+1234.56	E:.:,:	[E1.234,56]

Formatting DATE values

When does DATE formatting take place?

Date formatting occurs when converting a DATE to a string with the USING operator, for example in a LET, DISPLAY or PRINT instruction, and when displaying date values in form fields defined with the FORMAT attribute.

This example formats a DATE value with the USING operator:

```
MAIN
  DEFINE d DATE
  LET d = MDY(12,24,2014)
  DISPLAY d USING "mmm ddd yyyy"
END MAIN
```

This code example produces the following output:

```
Dec Wed 2014
```

Default formatting occurs when USING or FORMAT are not used, and a date value has to be converted to a character string, for example when passing a DATE to a function expecting a VARCHAR(n). Default date formatting is based on the date format defined with the DBDATE environment variable. For more details about default formatting, see Data type conversion reference on page 212.

This topic describes the syntax of the *format-string* in the USING "*format-string*" operator and FORMAT = "*format-string*" form field attribute.

Formatting symbols for DATE values

When formatting DATE values, the *format-string* of the USING operator or FORMAT attribute consists of a set of place holders that represent the day of the month as digits or as abbreviated name of the month, the month of the year as digits or as abbreviated name of the month, and the year as 2, 3 or 4 digits.

Table 101: Format-string symbols for DATE values on page 221 shows the formatting symbols for DATE expressions. Any character different from the placeholders described in this table is interpreted as a literal and will appear as-is in the resulting string.

The calendar used for date formatting is the Gregorian calendar. The `c1` placeholder is a formatting symbol used to adapt the date to the Ming Guo calendar.

Table 101: Format-string symbols for DATE values

Placeholder	Description
`dd`	Day of the month as a 2-digit integer.
`ddd`	Three-letter English-language abbreviation of the day of the week. For example: Mon, Tue.
`mm`	Month as a 2-digit integer.
`mmm`	Three-letter English-language abbreviation of the month. For example: Jan, Feb.
`yy`	Year, as a 2-digits integer representing the 2 trailing digits.
`yyy`	Year as a 3-digit number (Ming Guo format only)
`yyyy`	Year as a 4-digit number.
`c1`	Ming Guo format modifier, see Using the Ming Guo date format on page 323.

Table 102: Date formatting examples

Format String	Date value	Result string
`dd/mm/yyyy`	`2011-10-24`	`24/10/2011`
`[dd/mm/yy]`	`2011-10-24`	`[24/10/11]`
`[ddd-mmm-yyyy]`	`0141-10-24`	`[Tue-Oct-0141]`
`(ddd.) mmm. dd, yyyy`	`1999-09-23`	`(Thu.) Sep. 23, 1999`

Formatting DATETIME values

When does DATETIME formatting take place?

Datetime formatting occurs when converting a `DATETIME` to a string, for example in a `LET`, `DISPLAY` or `PRINT` instruction, and when displaying datetime values in form fields.

By default, DATETIME values are formatted in the ISO format:

```
yyyy-mm-dd hh:mm:ss.fffff
```

The next example formats a `DATETIME` value by using the `util.Datetime.format()` method:

```
IMPORT util
MAIN
  DEFINE dt DATETIME YEAR TO SECOND
  LET dt = CURRENT
  DISPLAY util.Datetime.format(dt, "%Y-%m-%d %H:%M:%S")
END MAIN
```

This code example produces the following output:

```
2015-12-23 11:45:33
```

A datetime value can be formatted with the `util.Datetime.format()` method.

Converting strings to DATETIME values

When a string represents a datetime value is ISO format, it can be directly converted to a `DATETIME`:

```
DEFINE dt DATETIME YEAR TO FRACTION(5)
LET dt = "2015-12-24 11:34:56.82373"
```

If you need to convert a string that does not follow the ISO format, use the `util.Datetime.parse()` method, by specifying a format string:

```
DEFINE dt DATETIME YEAR TO MINUTE
LET dt = util.Datetime.parse( "2014-12-24 23:45", "%Y-%m-%d %H:%M" )
```

Formatting symbols for DATETIME values

When formatting `DATETIME` values, the *format-string* of the `util.Datetime.parse()` and `util.Datetime.format()` methods consists of a set of place holders that represent the different parts of a datetime value (year, month, day, hour, minute, second and fraction).

s shows the formatting symbols for `DATETIME` expressions. Any character different from the placeholders described in this table is interpreted as a literal and will appear as-is in the resulting string.

The calendar used for date formatting is the Gregorian calendar.

Table 103: Format-string symbols for DATETIME values

Placeholder	Description
%a	The abbreviated name of the day of the week. **Note:** When parsing a datetime string, %a and %A are equivalent to detect the name of the day of the week in abbreviated form or full day name.
%A	The full name of the day of the week.
%b or %h	The abbreviated month name. **Note:** When parsing a datetime string, %b/%h and %B are equivalent to detect the month name in abbreviated form or full month name.
%B	The full month name.
%c	The date and time representation.
%C	The century number (0-99)
%D	Equivalent to %m/%d/%y
%d	The day of month with 2 digits (01-31)
%e	The day of month with one or 2 digits (1-31)
%F	The fractional part of a second
%H	The hour with 2 digits (00-23).
%I	The hour on a 12-hour clock (1-12)

Placeholder	Description
%y	The year on 2 digits (91)
%Y	The year on 4 digits (1991)
%m	The month as 2 digits (01-12)
%M	The minutes (00-59)
%n	A newline character
%p	The locale's equivalent of AM or PM
%r	The 12-hour clock time. In the POSIX locale equivalent to %I:%M:%S %p
%R	Equivalent to %H:%M
%S	The seconds (00-59)
%t	A tab character
%T	Equivalent to %H:%M:%S
%x	The date, using the locale's date format.
%X	The time, using the locale's time format.
%w	The ordinal number of the day of the week (0-6), with Sunday = 0.
%y	The year within century (0-99)
%Y	The year, including the century (for example, 1991)

Table 104: Datetime formatting examples

Format String	Datetime value	Result string
%d/%m/%Y %H:%M	2011-10-24 11:23:45	24/10/2011 11:23
[%d %b %Y %H:%M:%S]	2011-10-24 11:23:45	[24 Oct 2011 11:23:45]
(%a.) %b. %d, %Y	1999-09-23	(Thu.) Sep. 23, 1999

Formatting INTERVAL values

When does INTERVAL formatting take place?

Interval formatting occurs when converting a INTERVAL to a string, for example in a LET, DISPLAY or PRINT instruction, and when displaying interval values in form fields.

By default, INTERVAL values are formatted in the ISO format:

```
[+|-]yyyy-mm
```

or:

```
[+|-]dddd hh:mm:ss.fffff
```

The next example formats a INTERVAL value by using the util.Interval.format() method:

```
IMPORT util
```

```
MAIN
  DEFINE iv INTERVAL DAY(6) TO MINUTE
  LET iv = "-157 11:23"
  DISPLAY util.Interval.format(iv, "%d %H:%M")
END MAIN
```

This code example produces the following output:

```
-157 11:23
```

And interval value can be formatted with the `util.Interval.format()` method.

Converting strings to INTERVAL values

When a string represents a interval value is ISO format, it can be directly converted to a `INTERVAL`:

```
DEFINE iv INTERVAL HOUR(6) TO FRACTION(5)
LET iv = "20234:34:56.82373"
```

If you need to convert a string that does not follow the ISO format, use the `util.Interval.parse()` method, by specifying a format string:

```
DEFINE iv INTERVAL DAY(6) TO FRACTION(5)
LET iv = util.Interval.parse( "-7467 + 23:45:34.12345", "%d + %H:%M:%S%F5" )
```

Formatting symbols for INTERVAL values

When formatting `INTERVAL` values, the *format-string* of the `util.Interval.parse()` and `util.Interval.format()` methods consists of a set of place holders that represent the different parts of a interval value (year, month, day, hour, minute, second and fraction).

Table 105: Format-string symbols for INTERVAL values on page 224 shows the formatting symbols for `INTERVAL` expressions. Any character different from the placeholders described in this table is interpreted as a literal and will appear as-is in the resulting string.

Table 105: Format-string symbols for INTERVAL values

Placeholder	Description
`%Y`	Years (0-999999999)
`%m`	Month (00-11) or (0-999999999)
`%d`	Days (0-999999999)
`%H`	Hours (00-23) or (0-999999999)
`%M`	Minutes (00-59) or (0-999999999)
`%S`	Secondes (00-59) or (0-999999999)
`%F[n]`	The fractional part of a second, where n specifies the number of digits in the fractional part (1 to 5)
`%t`	A tab character
`%n`	A newline character

Table 106: Interval formatting examples

Format String	Interval value	Result string
`%d days %H:%M`	`54561 11:23`	`54561 days 11:23`
`%d days %H:%M:%S%F5`	`54561 11:23:45.12345`	`54561 days 11:23:45.12345`
`[%Y years and %m months]`	`1023-03`	`[1023 years and 03 months]`

Literals

Describes the syntax of literals (constant values) to be used in sources.

- Integer literals on page 225
- Numeric literals on page 225
- Text literals on page 226
- Datetime literals on page 227
- Interval literals on page 228

Integer literals

Integer literals define a whole number in an expression.

Syntax

```
[+|-] digit[...]
```

1. *digit* is a digit character from '0' to '9'.

Usage

Integer literals are in base-10 notation, without blank spaces and commas and without a decimal point.

Integer literals can be used to specify values for `DECIMAL(P,0)`, `BIGINT`, `INTEGER`, `SMALLINT` and `TINYINT` data types.

Example

```
MAIN
  DEFINE n INTEGER
  LET n = 1234567
END MAIN
```

Numeric literals

Numeric literals define values with a decimal part in an expression.

Syntax

```
[+|-] digit[...] . digit[...] [ {e|E} [+|-] digit[...] ]
```

1. *digit* is a digit character from '0' to '9'.
2. Note that the decimal separator is always a dot, independently from DBMONEY.
3. The E notation can be used to specify the exponent.

Usage

Numeric/decimal literals in base-10 notation, without blank spaces and commas, with a decimal part after a dot.

Numeric literals can be used to specify values for `DECIMAL(P,S)`, `MONEY(P,S)`, `FLOAT` and `SMALLFLOAT` data types.

Example

```
MAIN
  DEFINE n DECIMAL(10,2)
  LET n = 12345.67
  LET n = -1.23456e-10
END MAIN
```

Text literals

Text literals define a character string in an expression.

Syntax 1 (using double quotes)

```
" char [...] "
```

Syntax 2 (using single quotes)

```
' char [...] '
```

1. *char* is any character supported in the current locale, or a `\` backslash escape character as described below:
   - `\\` : the backslash character.
   - `\"` : double-quote character.
   - `\'` : single-quote character.
   - `\n` : newline character.
   - `\r` : carriage-return character.
   - `\0` : null character.
   - `\f` : form-feed character.
   - `\t` : tab character.
   - `\x`*NN* : ASCII character defined by the hexadecimal code *NN*.

Usage

A text literal (or character string literal) defines a character string constant containing valid characters in the current application character set. The application character set is defined by the current locale.

A text literal can be written on multiple lines, the compiler merges lines by removing the newline character.

An empty string ("") is equivalent to `NULL`.

The escape character is the backslash character (\).

When using single quotes as delimiters, double quotes can be used as is inside the string, while single quotes must be doubled or escaped with a backslash:

```
DISPLAY '  2 double quotes: " "   2 single quotes: '' \' '
displays as:
  2 double quotes: " "   2 single quotes: ' '
```

When using double quotes as delimiters, single quotes can be used as is inside the string, while double quotes must be doubled or escaped with a backslash:

```
DISPLAY "  2 double quotes: "" \"   2 single quotes: ' ' "
displays as:
  2 double quotes: " "   2 single quotes: ' '
```

Special characters can be specified with backslash escape symbols. Use for example \n to insert a new-line character in a string literal:

```
DISPLAY "First line\nSecond line"
```

The \x*NN* hexadecimal notation allows you to specify control characters in a string literal. Only ASCII codes (<=0x7F) are allowed.

Example

```
MAIN
  DISPLAY "Some text in double quotes"
  DISPLAY 'Some text in single quotes'
  DISPLAY "Include double quotes:  \"  ""  "
  DISPLAY 'Include single quotes:  \'  ''  '
  DISPLAY 'Insert a newline character here: \n and continue with
 text.'
  DISPLAY "This is a text
     on multiple
     lines.\
     You can insert a newline with back-slash at the end of the
 line."
  IF "" IS NULL THEN
     DISPLAY 'Empty string is NULL'
  END IF
END MAIN
```

Datetime literals

Datetime literals define date/time value in an expression.

Syntax

```
DATETIME ( dtrep ) qual1 TO qual2[(scale)]
```

where *qual1* can be one of:

```
  YEAR
  MONTH
  DAY
  HOUR
  MINUTE
  SECOND
  FRACTION
```

and *qual2* can be one of:

```
  YEAR
  MONTH
  DAY
  HOUR
  MINUTE
```

```
  SECOND
  FRACTION
  FRACTION(1)
  FRACTION(2)
  FRACTION(3)
  FRACTION(4)
  FRACTION(5)
```

1. *dtrep* is the datetime value representation in normalized format (YYYY-MM-DD hh:mm:ss.fffff).
2. *scale* defines the number of significant digits of the fractions of a second.
3. *qual1* and *qual2* qualifiers define the precision of the `DATETIME` literal.

Usage

A datetime literal is specified with the `DATETIME( )` notation, and is typically used in interval or datetime expressions, or to assign a DATETIME variable. In order to get the current date and time, use the `CURRENT` operator.

Example

```
MAIN
  DEFINE d1 DATETIME YEAR TO SECOND
  DEFINE d2 DATETIME HOUR TO FRACTION(5)
  LET d1 = DATETIME( 2002-12-24 23:55:56 ) YEAR TO SECOND
  LET d2 = DATETIME( 23:44:55.34532 ) HOUR TO FRACTION(5)
END MAIN
```

Interval literals

Interval literals define an interval value in an expression.

Syntax 1: *year-month* class interval

```
 INTERVAL ( inrep ) YEAR[(precision)] TO MONTH
|INTERVAL ( inrep ) YEAR[(precision)] TO YEAR
|INTERVAL ( inrep ) MONTH[(precision)] TO MONTH
```

Syntax 2: *day-time* class interval

```
 INTERVAL ( inrep ) DAY[(precision)] TO FRACTION[(scale)]
|INTERVAL ( inrep ) DAY[(precision)] TO SECOND
|INTERVAL ( inrep ) DAY[(precision)] TO MINUTE
|INTERVAL ( inrep ) DAY[(precision)] TO HOUR
|INTERVAL ( inrep ) DAY[(precision)] TO DAY

|INTERVAL ( inrep ) HOUR[(precision)] TO FRACTION[(scale)]
|INTERVAL ( inrep ) HOUR[(precision)] TO SECOND
|INTERVAL ( inrep ) HOUR[(precision)] TO MINUTE
|INTERVAL ( inrep ) HOUR[(precision)] TO HOUR

|INTERVAL ( inrep ) MINUTE[(precision)] TO FRACTION[(scale)]
|INTERVAL ( inrep ) MINUTE[(precision)] TO SECOND
|INTERVAL ( inrep ) MINUTE[(precision)] TO MINUTE

|INTERVAL ( inrep ) SECOND[(precision)] TO FRACTION[(scale)]
|INTERVAL ( inrep ) SECOND[(precision)] TO SECOND

|INTERVAL ( inrep ) FRACTION TO FRACTION[(scale)]
```

1. *inrep* is the interval value representation in normalized format (YYYY-MM or DD hh:mm:ss.fffff, according to the interval class).

Usage

An interval literal is specified with the `INTERVAL( )` notation, and is typically used to assign in interval or datetime expressions, or to assign an interval variable.

Example

```
MAIN
  DEFINE i1 INTERVAL YEAR TO MONTH
  DEFINE i2 INTERVAL HOUR(5) TO SECOND
  LET i1 = INTERVAL( 345-5 ) YEAR TO MONTH
  LET i2 = INTERVAL( 34562:22:33 ) HOUR(5) TO SECOND
END MAIN
```

Expressions

Shows the possible expressions supported in the language.

- Understanding expressions on page 229
- Boolean expressions on page 230
- Integer expressions on page 231
- Numeric expressions on page 231
- String expressions on page 232
- Date expressions on page 232
- Datetime expressions on page 233
- Interval expressions on page 233

Understanding expressions

What is an expression ?

An expression is a sequence of operands, operators, and parentheses that the runtime system can evaluate as a single value. Operands are program variables, constants, functions returning a single value and literal values. Operators are used for arithmetic or string manipulation, and the parentheses are used to overwrite precedence of operators.

Language and SQL expressions

Expressions in SQL statements are evaluated by the database server, not by the runtime system. The set of operators that can appear in SQL expressions resembles the set of language operators, but they are not identical. A program can include SQL operators, but these are restricted to SQL statements. Similarly, most SQL operands are not valid in program expressions. The SQL identifiers of databases, tables, or columns can appear in a `LIKE` clause or field name in program instructions, provided that these SQL identifiers comply with the naming rules of language. Here are some examples of SQL operands and operators that cannot appear in other language expressions:

- SQL identifiers, such as column names
- The SQL keywords `USER` and `ROWID`
- Built-in or aggregate SQL functions that are not part of the language
- The `BETWEEN` and `IN` operators
- The `EXISTS`, `ALL`, `ANY`, or `SOME` keywords of SQL expressions

Conversely, you *cannot* include language-specific operators in SQL expressions. For example:

- Arithmetic operators for exponentiation (`**`) and modulus (`MOD`)
- String operators `ASCII`, `COLUMN`, `SPACE`, `SPACES`, and `WORDWRAP`
- Field operators `FIELD_TOUCHED()`, `GET_FLDBUF()`, and `INFIELD()`
- The report operators `LINENO` and `PAGENO`

Parentheses in expressions

Parentheses are used as in algebra, to override the default order of precedence of operators. In mathematics, this use of parentheses represents the "associative" operator. It is, however, a convention in computer languages to regard this use of parentheses as delimiters rather than as operators. (Do not confuse this use of parentheses to specify *operator precedence* with the use of parentheses to enclose arguments in function calls or to delimit other lists.)

In this example, the variable **y** is assigned the value of 2.

```
LET y = 15 MOD 3 + 2
```

In this example, **y** is assigned the value of 0 because the parentheses change the sequence of operations.

```
LET y = 15 MOD (3 + 2)
```

Boolean expressions

A boolean expressions evaluates to an `INTEGER` value that can be `TRUE`, `FALSE` and in some cases, `NULL`.

```
MAIN
  DEFINE r, c INTEGER
  LET c = 4
  LET r = ( c!=5 ) AND ( c==2 OR c==4 )
  IF ( r AND canReadFile("config.txt") ) THEN
     DISPLAY "OK"
  END IF
END MAIN
```

Boolean expressions are a combination of logical operators and boolean comparison operators such as `==`, `>=` or `!=`. The result type of a boolean expression is an `INTEGER`. Any integer value different from zero is defined as true, while zero is defined as false. You can use an `INTEGER` or a `BOOLEAN` variable to store the result of a boolean expression.

```
MAIN
  DEFINE b BOOLEAN
  LET b = ( "a" == "b" )  -- result is FALSE (0)
END MAIN
```

If an expression that returns `NULL` is the operand of the `IS NULL` operator, the value of the boolean expression is `TRUE`.

```
MAIN
  DEFINE r INTEGER
  LET r = NULL
  IF r IS NULL THEN
     DISPLAY "TRUE"
  END IF
END MAIN
```

If you include a boolean expression in a context where the runtime system expects a number, the expression is evaluated, and is then converted to an integer by the rules `TRUE=1` and `FALSE=0`.

```
MAIN
  DEFINE r INTEGER
  LET c = 4
  LET r = 4 + (1==0)    -- result is 4.
END MAIN
```

The boolean expression evaluates to `TRUE` if the value is a non-zero real number or any of the following items:

- Character string representing a non-zero number
- Non-zero `INTERVAL`
- Any `DATE` or `DATETIME` value
- A `TRUE` value returned by a boolean function like `INFIELD()`
- The built-in integer constant `TRUE`

If a boolean expression includes an operand whose value is not an integer data type, the runtime system attempts to convert the value to an integer according to the data conversion rules.

A boolean expression evaluates to `NULL` if the value is `NULL` and the expression does not appear in any of the following contexts:

- The `IS [NOT] NULL` test.
- Boolean Comparisons.
- Any conditional statement (`IF`, `CASE`, `WHILE`).

The syntax of boolean expressions in programs is not the same as *Boolean conditions* in SQL statements.

Boolean expressions in `CASE`, `IF`, or `WHILE` statements return `FALSE` if any element of the comparison is `NULL`, except for operands of the `IS NULL` and the `IS NOT NULL` operator.

Integer expressions

An integer expression evaluates to a whole number.

```
MAIN
  DEFINE r, c INTEGER
  LET c = 4
  LET r = c * ( 2 + c MOD 4 ) / getRowCount("customers")
END MAIN
```

The operands of an integer expression can be:

- An integer literal.
- A variable or constant of type `TINYINT`, `SMALLINT`, `INTEGER` or `BIGINT`.
- A function returning a single integer value.
- A boolean expression.
- The result of a DATE subtraction, as a number of days.

If an integer expression includes an operand whose value is not an integer data type, the runtime system attempts to convert the value to an integer according to the data conversion rules.

If an element of an integer expression is `NULL`, the expression is evaluated to `NULL`.

Numeric expressions

A numeric expression evaluates to a decimal value.

```
MAIN
```

```
  DEFINE r, c DECIMAL(10,2)
  LET c = 456.22
  LET r = c * 2 + ( c / 4.55 )
END MAIN
```

The operands of a numeric expression can be one of:

- An integer literal.
- A decimal literal.
- A variable or constant of numeric data type.
- A function returning a single numeric value.
- A boolean expression.
- The result of a DATE subtraction, as a number of days.

If a number expression includes an operand whose value is not a numeric data type, the runtime system attempts to convert the value to a number according to the data conversion rules.

If an element of a number expression is `NULL`, the expression is evaluated to `NULL`.

String expressions

A string expression includes at least one character string value and evaluates to a string data type value.

```
MAIN
  DEFINE r, c VARCHAR(100)
  LET c = "abcdef"
  LET r = c[1,3] || ": " || TODAY USING "YYYY-MM-DD" || " " || length(c)
END MAIN
```

The data type of string expression result is `STRING`.

At least one of the operands in a string expression must be one of:

- A string literal.
- A variable or constant of `CHAR`, `VARCHAR` or `STRING` data type.
- A function returning a single character value.

Other operands whose values are not character string data types are converted to strings according to the data conversion rules.

If an element of a string expression is `NULL`, the expression is evaluated to NULL.

An empty string ("") is equivalent to `NULL`.

Date expressions

A date expression evaluates to a `DATE` data type value.

```
MAIN
  DEFINE r, c DATE
  LET c = TODAY + 4
  LET r = ( c - 2 )
END MAIN
```

The operands of a date expression can be one of:

- A string literal that can be evaluated to a date according to DBDATE environment variable.
- A variable or constant of type `DATE`.
- A function returning a single Date value.
- A unary + or - associated to an integer expression representing a number of days.
- The `TODAY` constant.

- A CURRENT expression with YEAR TO DAY qualifiers.
- An EXTEND expression with YEAR TO DAY qualifiers.

If a date expression includes an operand whose value is not a date data type, the runtime system attempts to convert the value to a date value according to the data conversion rules.

If an element of an date expression is NULL, the expression is evaluated to NULL.

Datetime expressions

A datetime expression evaluates to a DATETIME data type.

```
MAIN
  DEFINE r, c DATETIME YEAR TO SECOND
  LET c = CURRENT YEAR TO SECOND
  LET r = c + INTERVAL( 234-02 ) YEAR TO MONTH
END MAIN
```

The operands of a datetime expression can be one of:

- A datetime literal.
- A string literal representing a datetime with the format YYYY-MM-DD hh:mm:ss.fffff.
- A variable or constant of DATETIME type.
- A function returning a single Datetime value.
- A unary + or - associated to an interval expression.
- A CURRENT expression.
- An EXTEND expression.

If a datetime expression includes an operand whose value is not a datetime data type, the runtime system attempts to convert the value to a datetime value according to the data conversion rules.

If an element of an integer expression is NULL, the expression is evaluated to NULL.

Interval expressions

An interval evaluates to an INTERVAL data type.

```
MAIN
  DEFINE r, c INTERVAL HOUR TO MINUTE
  LET c = "12:45"
  LET r = c + ( DATETIME(14:02) HOUR TO MINUTE - DATETIME(10:43) HOUR TO
 MINUTE )
END MAIN
```

The operands of an interval expression must be one of:

- An interval literal.
- A string literal representing an Interval with the format YYYY-MM-DD hh:mm:ss.fffff.
- An integer expression using the UNITS operator.
- A variable or constant of INTERVAL type.
- A function returning a single interval value.
- The result of a DATETIME subtraction.

If an interval expression includes an operand whose value is not an interval data type, the runtime system attempts to convert the value to an interval value according to the data conversion rules.

If an element of an integer expression is NULL, the expression is evaluated to NULL.

Operators

This section describes basic syntax elements that can appear in expressions.

There are different sort of basic syntax elements such as operators for arithmetics, string and comparison, predefined variables and registers like `SQLSTATE`, and utility operators like `SFMT()` or `TODAY`.

Elements of an expressions are evaluated according to their precedence, from highest to lowest, as described in the order of precedence list. Use `()` parentheses to instruct the runtime system to evaluate the expression in a different way than the default order of precedence.

- Order of precedence on page 234
- General warnings regarding expressions on page 236
- List of expression elements on page 236

Order of precedence

The following list describes the precedence order of expression elements. The order of precedence defines in which order the elements of an expression are evaluated.

For example, the `MOD` operator has a higher precedence as the `*` operator. When computing an expression like `( 33 MOD 2 * 5 )`, the runtime system first evaluates `(33 MOD 2) = 1` and then evaluates `(1 * 5) = 5`. The order of evaluation can be changed this by using parentheses: `( 33 MOD ( 2 * 5 ) ) = 3`.

Table 107: Order of precedence list

P	Syntax Element	A	Description	Example
14	`CAST(v AS type)`	N	Type casting	`CAST(var AS fgl.FglRecord)`
14	`INSTANCEOF`	L	Type checking	`var INSTANCEOF`

P	Syntax Element	A	Description	Example
				`java.lang.Boolean`
13	`UNITS`	L	Single-qualifier interval	`(integer) UNITS DAY`
12	`+`	R	Unary plus	`+ number`
12	`-`	R	Unary minus	`- number`
11	`**`	L	Exponentiation	`x ** 5`
11	`MOD`	L	Modulus	`x MOD 2`
10	`*`	L	Multiplication	`x * y`
10	`/`	L	Division	`x / y`
9	`+`	L	Addition	`x + y`
9	`-`	L	Subtraction	`x - y`
8	`||`	L	Concatenation	`"Amount:" || amount`
7	`LIKE`	R	String comparison	`mystring LIKE "A%"`
7	`MATCHES`	R	String comparison	`mystring MATCHES "A*"`
6	`<`	L	Less than	`var < 100`
6	`<=`	L	Less then or equal to	`var <= 100`
6	`>`	L	Greater than	`var > 100`
6	`>=`	L	Greater than or equal to	`var >= 100`
6	`==`	L	Equals	`var == 100`
6	`<> or !=`	L	Not equal to	`var <> 100`
5	`IS NULL`	L	Test for NULL	`var IS NULL`
5	`IS NOT NULL`	L	Test for NOT NULL	`var IS NOT NULL`

In this table, the **P** column defines the precedence, from highest (14) to lowest (1). Note that some operators have the same precedence (i.e. are equivalent in evaluation order). The **A** column defines the direction of association (L=Left, R=Right, N=None).

General warnings regarding expressions

Pure SQL Syntax Elements

The following are related to SQL syntax and not part of the language:

- `BETWEEN` *`expr`* `AND` *`expr`*
- `IN ( ` *`expr`* `[ , ..' ] )`

Report Routine Syntax Elements

The following are only available in the FORMAT section of report routines:

- `PAGENO`
- `WORDWRAP`

See Report Definition for more details.

List of expression elements

Comparison operators

Comparison operators allow you to compare two values, to include the greater than, less than and equal to functions.

Table 108: Comparison operators

Operator	Description
IS NULL on page 237	The `IS NULL` operator checks for `NULL` values.
LIKE on page 237	The `LIKE` operator returns `TRUE` if a string matches a given mask.
MATCHES on page 238	The `MATCHES` operator returns `TRUE` if a string matches a given mask.
Equal to (==) on page 239	The == operator checks for equality of two expressions or for two record variables.
Different from (!=) on page 240	The `!=` operator checks for non-equality of two expressions or for two record variables.
Lower (<) on page 241	The < operator is provided to test whether a value or expression is lower than another.
Lower or equal (<=) on page 241	The <= operator is provided to test whether a value or expression is lower than or equal to another.
Greater (>) on page 242	The > operator is provided to test whether a value or expression is greater than another.
Greater or equal (>=) on page 242	The >= operator is provided to test whether a value or expression is greater than or equal to another.
NVL() on page 242	The `NVL( )` operator returns the second parameter if the first argument evaluates to `NULL`.

Operator	Description
IIF() on page 243	The `IIF()` operator returns the second or third parameter according to the boolean expression given as first argument.

IS NULL
The `IS NULL` operator checks for `NULL` values.

Syntax

```
expr IS NULL
```

1. *expr* can be any expression supported by the language.

Usage

The `IS NULL` operator can be used to test whether the left-hand expression is `NULL`.

This operator applies to most data types, except complex types like `BYTE` and `TEXT`.

Example

```
MAIN
  DEFINE n INTEGER
  LET n = NULL
  IF n IS NULL THEN
     DISPLAY "The variable is NULL."
  END IF
END MAIN
```

LIKE
The `LIKE` operator returns `TRUE` if a string matches a given mask.

Syntax

```
expr [NOT] LIKE mask [ ESCAPE "char" ]
```

1. *expr* is any character string expression.
2. *mask* is a character string expression defining the filter.
3. *char* is a single char specifying the escape symbol.

Usage

The *mask* can be any combination of characters, including the % and _ wildcards:

- The % percent character matches any string of zero or more characters.
- The _ underscore character matches any single character.

The ESCAPE clause can be used to define an escape character different from the default backslash. It must be enclosed in single or double quotes.

A backslash (or the escape character specified by the ESCAPE clause) makes the operator treat the next character as a literal character, even if it is one of the special symbols in the *mask* list. This allows you to search for %,_ or \ characters.

Do not confuse with the `LIKE` clause of the `DEFINE` instruction .`LIKE` operators used in SQL statements are evaluated by the database server. This may have a different behavior than the `LIKE` operator of the language.

If you need to escape a wildcard character, keep in mind that a string constant must also escape the backslash character. As a result, if you want to pass a backslash to the `LIKE` operator (by using backslash as default escape character), you need to write four backslashes in the original string constant.

The next table shows some examples of string constants used in the source code and their equivalent `LIKE` pattern:

Table 109: Examples of string constants used in the source code and their equivalent `LIKE` pattern

Original String Constant	Equivalent MATCHES pattern	Description
`"%"`	`%`	Matches any character in a non-empty string.
`"_"`	`_`	Matches a single character.
`"abc%"`	`abc%`	Starts with abc.
`"*abc"`	`%abc`	Ends with abc.
`"%abc%"`	`%abc%`	Contains abc.
`"abc__"`	`abc__`	Strings equals abc followed by two additional characters.
`"\\%"`	`\%`	Contains a single star character (the % wildcard is escaped)
`"%abc\\\\def%"`	`%abc\\def%`	Contains abc followed by a backslash followed by def (the backslash is escaped)

Example

```
MAIN
  IF "abcdef" LIKE "a%e_" THEN
     DISPLAY "The value matches."
  END IF
END MAIN
```

MATCHES

The `MATCHES` operator returns `TRUE` if a string matches a given mask.

Syntax

```
expr [NOT] MATCHES mask [ ESCAPE "char" ]
```

1. *expr* is any character string expression.
2. *mask* is a character string expression defining the filter.
3. *char* is a single char specifying the escape symbol.

Usage

The *mask* can be any combination of characters, including the `*`, `?`, `[`, `]`, `-` and `^` wildcards:

- The `*` star character matches any string of zero or more characters.
- The `?` question mark matches any single character.
- The `[ ]` brackets match any enclosed character.
- Inside `[ ]`, the `-` (hyphen) between characters means a range of characters.

- Inside `[ ]`, the ^ An initial caret matches any character that is not listed.

The ESCAPE clause can be used to define an escape character different from the default backslash. It must be enclosed in single or double quotes.

A backslash (or the escape character specified by the ESCAPE clause) makes the operator treat the next character as a literal character, even if it is one of the special symbols in the *mask* list. This allows you to search for wildcard charachers such as `*`, `?`, `[`, `]` or `\`.

If you need to escape a wildcard character, keep in mind that a string constant must also escape the backslash character. As a result, if you want to pass a backslash to the MATCHES operator (by using backslash as default escape character), you need to write four backslashes in the original string constant.

The next table shows some examples of string constants used in the source code and their equivalent MATCHES pattern:

Table 110: String constants used in the source code and their equivalent `MATCHES` pattern

Original String Constant	Equivalent MATCHES pattern	Description
`"*"`	`*`	Matches any character in a non-empty string.
`"?"`	`?`	Matches a single character.
`"abc*"`	`abc*`	Starts with abc.
`"*abc"`	`*abc`	Ends with abc.
`"*abc*"`	`*abc*`	Contains abc.
`"abc??"`	`abc??`	Starts with abc, followed by two additional characters.
`"[a-z]*"`	`[a-z]*`	Starts with a letter in the range a to z.
`"[^0-9]*"`	`[^0-9]*`	Must not start with a digit.
`"\\*"`	`\*`	Contains a single star character (the * wildcard is escaped)
`"*abc\\\\def*"`	`*abc\\def*`	Contains abc followed by a backslash followed by def (the backslash is escaped)

Example

```
MAIN
  IF "55f-plot" MATCHES "55[a-z]-*" THEN
     DISPLAY "Item reference format is correct."
  END IF
END MAIN
```

Equal to (==)

The == operator checks for equality of two expressions or for two record variables.

Syntax 1: Expression comparison

```
expr == expr
```

Syntax 2: Record comparison

```
record1.* == record2.*
```

1. *expr* can be any expression supported by the language.
2. *record1* and *record2* are records with the same structure.

Usage

The == operator evaluates whether two expressions or two records are identical.

A single equal sign (=) can be used as an alias for the == operator.

When comparing expressions using the first syntax, the result of the operator is FALSE when one of the operands is NULL. This first syntax applies to most data types, except complex types like BYTE and TEXT.

When comparing two records using the second syntax, the runtime system compares all corresponding members of the records. If a pair of members are different, the result of the operator is FALSE. When two corresponding members are NULL, they are considered as equal. This second syntax allows you to compare all members of records, but records must have the same structure.

Example

```
MAIN
  DEFINE n INTEGER
  LET n=512
  IF n==512 THEN
     DISPLAY "The variable equals 512."
  END IF
END MAIN
```

Different from (!=)

The != operator checks for non-equality of two expressions or for two record variables.

Syntax 1: Expression comparison

```
expr != expr
```

Syntax 2: Record comparison

```
record1.* != record2.*
```

1. <> is a synonym for!=
2. *expr* can be any expression supported by the language.
3. *record1* and *record2* are records with the same structure.

Usage

The != operator evaluates whether two expressions or two records are different.

A less-than sign followed by a greater-than sign (<>) can be used as an alias for the != operator.

When comparing expressions with the first syntax, the result of the operator is FALSE when one of the operands is NULL. This syntax applies to most data types except complex types like BYTE and TEXT.

When comparing two records with the second syntax, the runtime system compares all corresponding members of the records. If one pair of members are different, the result of the operator is TRUE. When two corresponding members are NULL, they are considered as equal. This second syntax allows you to compare all members of records, but records must have the same structure.

Example

```
MAIN
  DEFINE n INTEGER
  LET n==512
  IF n!=32 THEN
     DISPLAY "The variable is not equal to 32."
  END IF
END MAIN
```

Lower (<)

The < operator is provided to test whether a value or expression is lower than another.

Syntax

```
expr < expr
```

Usage

Applies to most data types, except complex types such as BYTE and TEXT.

If one of the operands is NULL, the comparison expression evaluates to FALSE.

Example

```
MAIN
  DEFINE n INT
  LET n = 45
  IF n < 100 THEN
     DISPLAY "The variable is lower than 100."
  END IF
END MAIN
```

Lower or equal (<=)

The <= operator is provided to test whether a value or expression is lower than or equal to another.

Syntax

```
expr <= expr
```

Usage

Applies to most data types, except complex types such as BYTE and TEXT.

If one of the operands is NULL, the comparison expression evaluates to FALSE.

Example

```
MAIN
  DEFINE n INT
  LET n = 100
  IF n <= 100 THEN
     DISPLAY "The variable is lower than or equal to 100."
  END IF
END MAIN
```

Greater (>)
The > operator is provided to test whether a value or expression is greater than another.

Syntax

```
expr > expr
```

Usage

Applies to most data types, except complex types such as BYTE and TEXT.

If one of the operands is NULL, the comparison expression evaluates to FALSE.

Example

```
MAIN
  DEFINE n INT
  LET n = 200
  IF n > 100 THEN
     DISPLAY "The variable is greater than 100."
  END IF
END MAIN
```

Greater or equal (>=)
The >= operator is provided to test whether a value or expression is greater than or equal to another.

Syntax

```
expr >= expr
```

Usage

Applies to most data types, except complex types such as BYTE and TEXT.

If one of the operands is NULL, the comparison expression evaluates to FALSE.

Example

```
MAIN
  DEFINE n INT
  LET n = 100
  IF n >= 100 THEN
     DISPLAY "The variable is greater than or equal to 100."
  END IF
END MAIN
```

NVL()
The NVL( ) operator returns the second parameter if the first argument evaluates to NULL.

Syntax

```
NVL( main-expr, subst-expr )
```

1. *main-expr* and *subst-expr* are any expression supported by the language.

Usage

The `NVL()` operator evaluates the first argument, and returns the result if the value is not null, otherwise it returns the second argument. This allows you to write the equivalent of the following `IF` statement, in a simple scalar expression:

```
IF main-expr IS NOT NULL THEN
  RETURN main-expr
ELSE
  RETURN subst-expr
END IF
```

Example

```
MAIN
  DEFINE var VARCHAR(100)
  LET var = arg_val(1)
  DISPLAY "The argument value is: ", NVL(var, "NULL")
END MAIN
```

IIF()

The `IIF()` operator returns the second or third parameter according to the boolean expression given as first argument.

Syntax

```
IIF( bool-expr, true-expr, false-expr)
```

1. *bool-expr* is a boolean expression.
2. *true-expr* and *false-expr* are language expressions.

Usage:

The `IIF()` operator evaluates the first argument, the returns the second argument if the first argument is true, otherwise it returns the third argument. This allows you to write the equivalent of the following `IF` statement, in a simple scalar expression:

```
IF bool-expr THEN
   RETURN true-expr
ELSE
   RETURN false-expr
END IF
```

Example

```
MAIN
  DEFINE var VARCHAR(10)
  LET var = arg_val(1)
  DISPLAY IIF(var == "A", "Accepted", "Rejected")
END MAIN
```

Logical operators

Logical operators include NOT, AND and OR.

Table 111: Logical operators

Operator	Description
NOT on page 244	The NOT operator performs a logical negation to invert a boolean expression.
AND on page 244	The AND operator is the logical intersection operator.
OR on page 245	The OR operator is the logical union operator.

NOT

The NOT operator performs a logical negation to invert a boolean expression.

Syntax

```
NOT bool-expr
```

1. *bool-expr* is a boolean expression.

Usage

The NOT operator is typically used to invert the value of a boolean expression.

If the operand is NULL, the negation expression evaluates to NULL.

Example

```
MAIN
  IF NOT ( 256 == 257 ) THEN
     DISPLAY "This line should display"
  END IF
END MAIN
```

AND

The AND operator is the logical intersection operator.

Syntax

```
bool-expr AND bool-expr
```

1. *bool-expr* is a boolean expression.

Usage

If one of the operands is NULL, the logical expression evaluates to FALSE.

By default, the runtime system evaluates both operands on the left and right side of the AND keyword. This is the traditional behavior of the Genero language, but in fact the right operand does not need to be evaluated if the first operand evaluates to FALSE. This method is called *short-circuit evaluation*, and can be enabled by adding the OPTIONS SHORT CIRCUIT clause at the beginning of the module.

Example

```
MAIN
```

```
  IF 256!=257 AND 257==257 THEN
     DISPLAY "This line should display"
  END IF
END MAIN
```

OR

The OR operator is the logical union operator.

Syntax

```
bool-expr OR bool-expr
```

1. *bool-expr* is a boolean expression.

Usage

If one of the operands is NULL, the logical expression evaluates to FALSE.

By default, the runtime system evaluates both operands on the left and right side of the OR keyword. This is the traditional behavior of the Genero language, but in fact the right operand does not need to be evaluated if the first operand evaluates to TRUE. This method is called *short-circuit evaluation*, and can be enabled by adding the OPTIONS SHORT CIRCUIT clause at the beginning of the module.

Example

```
MAIN
  IF TRUE OR FALSE THEN
     DISPLAY "This line should display"
  END IF
END MAIN
```

Arithmetic operators

Arithmetic operators allow you to complete numeric operations, such as addition and subtraction.

Table 112: Arithmetic operators

Operator	Description
Addition (+) on page 245	The + operator adds a number to another.
Subtraction (-) on page 246	The - operator subtracts a number from another.
Multiplication (*) on page 246	The * operator multiplies a number with another.
Division (/) on page 247	The / operator divides a number by another.
Exponentiation (**) on page 247	The ** operator calculates an exponentiation.
MOD on page 247	The MOD operator calculates the modulus.

Addition (+)

The + operator adds a number to another.

Syntax

```
num-expr + num-expr
```

1. *num-expr* is a numeric expression.

Usage

Use the + operator to add two numeric values.

If one of the operands is `NULL`, the arithmetic expression evaluates to `NULL`.

Example

```
MAIN
  DISPLAY 100 + 200
END MAIN
```

Subtraction (-)

The - operator subtracts a number from another.

Syntax

```
num-expr - num-expr
```

1. *num-expr* is a numeric expression.

Usage

Use the - operator to subtract a numeric value from another numeric value.

If one of the operands is `NULL`, the arithmetic expression evaluates to `NULL`.

Example

```
MAIN
  DISPLAY 100 - 200
END MAIN
```

Multiplication (*)

The * operator multiplies a number with another.

Syntax

```
num-expr * num-expr
```

1. *num-expr* is a numeric expression.

Usage

Use the * operator to multiply a numeric value to another numeric value.

If one of the operands is `NULL`, the arithmetic expression evaluates to `NULL`.

Example

```
MAIN
  DISPLAY 100 * 200
END MAIN
```

Division (/)
The / operator divides a number by another.

Syntax

```
num-expr / num-expr
```

1. *num-expr* is a numeric expression.

Usage

Use the / operator to divide a numeric value by another numeric value.

If one of the operands is NULL, the arithmetic expression evaluates to NULL.

Example

```
MAIN
  DISPLAY 100 / 200
END MAIN
```

Exponentiation (**)
The ** operator calculates an exponentiation.

Syntax

```
num-expr ** int-expr
```

1. *num-expr* is a numeric expression.

Usage

The ** operator returns a value calculated by raising the left-hand operand to a power corresponding to the integer part of the right-hand operand.

If the right operand is a number with a decimal part, it is rounded to a whole integer before computing the exponentiation.

Example

```
MAIN
  DISPLAY 2 ** 8
  DISPLAY 10 ** 4
END MAIN
```

MOD
The MOD operator calculates the modulus.

Syntax

```
int-expr MOD int-expr
```

1. *int-expr* is an integer expression.

Usage

The MOD operator returns the remainder, as an integer, from the division of the integer part of two numbers.

If the right operand is a number with a decimal part, it is rounded to a whole integer before computing the modulus.

Example

```
MAIN
  DISPLAY 256 MOD 16
  DISPLAY 26 MOD 2
  DISPLAY 27 MOD 2
END MAIN
```

Character string operators

Character string operators allow you to work with and manipulate character strings.

Table 113: Character string operators

Operator	Description
ASCII() on page 248	The `ASCII()` operator produces an ASCII character.
COLUMN on page 249	The `COLUMN` operator generates blanks.
Concatenate (||) on page 250	The `||` operator makes a string concatenation.
Append (,) on page 250	The `,` (comma) appends an expression to a string.
Substring ([s,e]) on page 250	The `[]` (square braces) extract a substring.
USING on page 251	The `USING` operator converts date and numeric values to a string, according to a formatting mask.
CLIPPED on page 252	The `CLIPPED` operator removes trailing blanks of a string expression.
ORD() on page 252	The `ORD()` operator returns the code point of a character in the current locale.
SPACES on page 252	The `SPACES` operator returns a character string with blanks.
LSTR() on page 253	The `LSTR()` operator returns a localized string.
SFMT() on page 253	The `SFMT()` operator replaces place holders in a string with values.

ASCII()

The `ASCII()` operator produces an ASCII character.

Syntax

```
ASCII ( int-expr )
```

1. *int-expr* is an integer expression, in the range 0-255 or 0-127, according to the current locale.

Usage

The `ASCII()` operator returns the character corresponding to the ASCII code passed as a parameter.

`ASCII()` is typically used to generate a non-printable character such as newline or escape. You should avoid to use this function for other characters.

The possible values of the integer parameter passed to ASCII() depends on the locale settings:

- For single byte encodings (like ISO8859-1), the argument must be in the range of 0 to 255.
- For UTF-8, using char length semantics, the argument must be any valid 16bit code point.
- For any other locale setting (any multibyte character set, or UTF-8 with byte length semantics), the argument must be in the range 0 to 127.

When the argument is zero, ASCII() has a different behavior, according to the context:

- ASCII(0) only displays the NULL character within the PRINT statement.
- If you specify ASCII(0) in other contexts, it returns a blank space.

Example

```
MAIN
  DISPLAY ASCII(65), ASCII(66), ASCII(7)
END MAIN
```

COLUMN

The COLUMN operator generates blanks.

Syntax

```
COLUMN position
```

1. *position* is the column position (starts at 1).

Usage

The COLUMN operator is typically used in report routines to align data in PRINT statements and move the character position forward within the current line. This operator makes sense when used in an expression with the comma append operator: Spaces will be generated according to the number of characters that have been used in the expression, before the COLUMN operator.

The COLUMN operator can be used outside report routines, in order to align data to be displayed with a proportional font, typically in a TUI context. For example, the next lines will always display the content of the *lastname* variable starting from column 30 of the terminal, no matters the number of characters contained in the *firstname* variable. The example defines VARCHAR variables, since CHAR variables are blank-padded, we would need to use the CLIPPED operator:

```
DEFINE firstname, lastname VARCHAR(50)
DISPLAY firstname, COLUMN(30), lasttname
```

The *pos* operand must be a non-negative integer that specifies a character position offset (from the left margin) no greater than the line width (that is, no greater than the difference (right margin - left margin). This designation moves the character position to a left-offset, where 1 is the first position after the left margin. If current position is greater than the operand, the COLUMN specification is ignored.

Example

```
PAGE HEADER
  PRINT "Number", COLUMN 12,"Name", COLUMN 35,"Location"
ON EVERY ROW
  PRINT customer_num, COLUMN 12, fname, COLUMN 35,city
```

Concatenate (||)
The || operator makes a string concatenation.

Syntax

```
expr || expr
```

1. *expr* can be a character, numeric or date time expression.

Usage

The || operator is the concatenation operator that produces a string expression from the expression elements on both sides of the operator.

This operator has a high precedence; it can be used in parameters for function calls. The precedence of this operator is higher than LIKE and MATCHES, but less than arithmetic operators. For example, a || b + c is equivalent to (a||(b+c)).

If any of the members of a concatenation expression is NULL, the result string will be NULL.

Example

```
MAIN
  DISPLAY "Length: " || length( "ab" || "cdef" )
END MAIN
```

Append (,)
The , (comma) appends an expression to a string.

Syntax

```
char-expr, expr
```

Usage

The comma operator formats and concatenates expressions together.

This operator can only be used in some instructions such as LET, PRINT, MESSAGE, ERROR and DISPLAY instructions.

As an alternative, use the || concatenation operator.

Use the comma concatenation operator when data needs to be formatted for printing and display.

Example

```
MAIN
  DISPLAY "Today:", TODAY, " and a number: ", 12345.67
END MAIN
```

Substring ([s,e])
The [] (square braces) extract a substring.

Syntax

```
char-variable [ start  [, end  ] ]
```

1. *char-variable* must be a character data type variable.

2. *start* defines the position of the first character of the substring to be extracted.
3. *end* defines the position of the last character of the substring to be extracted.
4. If *end* is not specified, only one character is extracted.

Usage

The `[]` (square braces) notation following a `CHAR` or `VARCHAR` variable extracts a substring from that character variable.

The *start* and *end* arguments can be expressed in bytes or characters, depending on the length semantics used in your programs.

> **Important:** Substring expressions in SQL statements are evaluated by the database server. This may have a different behavior than the substring operator of the language.

Example

```
MAIN
  DEFINE s CHAR(10)
  LET s = "abcdef"
  DISPLAY s[3,4]
END MAIN
```

USING

The `USING` operator converts date and numeric values to a string, according to a formatting mask.

Syntax

```
expr USING format
```

1. *expr* is a language expression.
2. *format* is a string expression that defines the formatting mask to be used.

Usage

The `USING` operator applies a formatting string to the left operand.

The left operand must be a valid date, integer or decimal number. Note that `DATETIME` and `INTERVAL` expressions cannot be formatted with the USING operator.

The format string can be any valid string expression using formatting characters as described in Formatting numeric values on page 217 and Formatting DATE values on page 220.

The `USING` operator has a low order of precedence: if you use operators with a higher precedence, the resulting string might not be what you are expecting.

For example, the `||` concatenation operator is evaluated before `USING`. As a result:

```
LET x = a || b USING "format"
```

will first concatenate a and b, then apply the `USING` format.

To solve this issue, use braces around the `USING` expression:

```
LET x = a || (b USING "format")
```

Example

```
MAIN
  DEFINE d DECIMAL(12,2)
```

```
  LET d = -12345678.91
  DISPLAY d USING "$-##,###,##&.&&"
  DISPLAY TODAY USING "yyyy-mm-dd"
END MAIN
```

CLIPPED
The `CLIPPED` operator removes trailing blanks of a string expression.

Syntax

```
expr CLIPPED
```

1. *expr* is a language expression.

Usage

This operator removes all trailing spaces of a string expression.

The `CLIPPED` operator is typically used to remove the trailing blanks of a `CHAR` value, which would be printed otherwise.

Example

```
MAIN
  DISPLAY "Some text   " CLIPPED
END MAIN
```

ORD()
The `ORD()` operator returns the code point of a character in the current locale.

Syntax

```
ORD( source STRING )
```

1. *source* is a string expression.

Usage

The value returned by `ORD()` is the code point in the current locale of the character passed as argument.

Only the first character of the argument is evaluated.

When using UTF-8 with character length semantics, the `ORD()` operator returns the UNICODE code point of the character.

`ORD` returns `NULL` if the argument passed is not valid.

SPACES
The `SPACES` operator returns a character string with blanks.

Syntax

```
int-expr SPACES
```

1. *int-expr* is an integer expression.
2. `SPACE` (without S) is an alias for this operator.

Usage

The SPACE operator is typically used in reports to print spaces to align data in the report output.

Example

```
MAIN
  DISPLAY 20 SPACES || "xxx"
END MAIN
```

LSTR()
The LSTR( ) operator returns a localized string.

Syntax

```
LSTR(str-expr)
```

1. *str-expr* is a string expression.

Usage

The LSTR( ) operator returns a localized string corresponding to the identifier passed as parameter.

Normally localized strings are automatically replaced when using the %"ident" notation in the source code. When the localized string identifier is not known at compile time, use the LSTR( ) function.

Example

```
MAIN
  DISPLAY LSTR ("str"||123)  -- loads string 'str123'
END MAIN
```

SFMT()
The SFMT( ) operator replaces place holders in a string with values.

Syntax

```
SFMT( str-expr , param [ , param [...] ])
```

1. *str-expr* is a string expression.
2. *param* is any valid expression used to replace parameter place holders (%*n*).
3. At least one parameter is required.

Usage

The SFMT( ) operator can be used with parameters that will be automatically set in the string at the position defined by parameter placeholders. The parameters used with the SFMT( ) operator can be any valid expressions. Numeric and date/time expressions are evaluated to strings according to the current format settings (DBDATE, DBMONEY).

A placeholder a is special marker in the string, that is defined by the percent character followed by the parameter number. For example, %4 represents the parameter #4. You are allowed to use the same parameter placeholder several times in the string. If you want to use the percent sign in the string, you must escape it with %%.

Predefined placeholders can be used to insert information about last runtime system error that occurred. Note that these are only available in the context of a runtime error trapped with a WHENEVER ERROR GOTO / CALL handler:

Table 114: Predefined placeholders for runtime system error information

Predefined parameter	Description
%(ERRORFILE)	Name of the module where last runtime error occurred.
%(ERRORLINE)	Line number in the module where last runtime error occurred.
%(ERRNO)	Last operating system error number.
%(STRERROR)	Last operating system error text.

Example

```
MAIN
  DEFINE n INTEGER
  LET n = 234
  DISPLAY SFMT("Order #%1 has been %2.",n,"deleted")
END MAIN
```

In this example, %1 is replaced by the value of the variable *n*, while %2 is replaced by the string "deleted", resulting in: `Order #234 has been deleted.`

Associative syntax operators

Associative syntax operators allow you to group together objects.

Table 115: Associative syntax operators

Operator	Description
Parentheses (()) on page 254	Parentheses (( )) force the evaluation of an expression before other operators.
Membership (object.member) on page 255	Separator for object members.
Variable parameter list ([ ]) on page 255	Variable parameter list delimiters.

Parentheses (())

Parentheses (( )) force the evaluation of an expression before other operators.

Syntax

```
( expr  [...] )
```

1. *expr* is a language expression.

Usage

Parentheses can be used to change the order in which expression elements are evaluated, to bypass the precedence of operators.

Parentheses can also be used to ease the readability of the code in a complex expression.

Example

```
MAIN
  DEFINE n INTEGER
```

```
  LET n = ( ( 3 + 2 ) * 2 )
  IF n=10 AND ( n<=0 OR n>=20 ) THEN
    DISPLAY "OK"
  END IF
END MAIN
```

Membership (object.member)
Separator for object members.

Syntax

```
setname.element
```

Usage

The period expression element specifies that its right-hand operand is a member of the set whose name is its left-hand operand.

This notation is used to reference `RECORD` members, object and class methods, as well as module elements.

Example

```
IMPORT FGL customer_module
...
MAIN
  DEFINE rec RECORD
     n INTEGER,
     c CHAR(10)
  END RECORD
  DEFINE form ui.Form
  LET rec.n = 12345
  LET rec.c = "abcdef"
  ...
  CALL form.setElementHidden("page1")
  ...
  CALL customer_module.check(345)
  ...
END MAIN
```

Variable parameter list ([ ])
Variable parameter list delimiters.

Syntax

```
[ variable [,...] ]
```

Usage

The square brace notation in function parameters defines a variable list of arguments for a built-in function or a built-in class method.

The elements of a variable parameter list are program variables which are passed by reference. As result, the called function can modify the content of the passed variables, to return values in output parameters.

It is not possible to define user functions with variable parameter lists.

For real usage examples, see the read and write methods of the `base.Channel` class.

Example

```
MAIN
  DEFINE id INTEGER, name STRING,
           count INTEGER, stat INTEGER
  LET id = 12345
  LET name = "Forman"
  -- Warning: This is a fake call, the function does not exist!
  -- Here, id and name are passed as input values, while count
  -- and stat are used as output parameters...
  CALL built_in_function( [id,name], [count, stat] )
END MAIN
```

SQL related operators

SQL related operators allow you to retrieve the SQL state and the SQL error message.

Table 116: SQL related operators

Operator	Description
SQLSTATE on page 256	The SQLSTATE variable returns the code corresponding to the last SQL error.
SQLERRMESSAGE on page 257	The SQLERRMESSAGE variable holds the error message corresponding to the last SQL error.

SQLSTATE

The SQLSTATE variable returns the code corresponding to the last SQL error.

Syntax

```
SQLSTATE
```

Usage

The SQLSTATE predefined variable returns the ANSI/ISO SQLSTATE code when an SQL error occurred.

The SQLSTATE error code is a standard ANSI specification, but not all database engines support this feature. Check the database server documentation for more details.

The variable is NULL if the last SQL statement was successfull.

Example

```
MAIN
  DATABASE stores
  WHENEVER ERROR CONTINUE
  SELECT foo FROM bar
  DISPLAY SQLSTATE
END MAIN
```

SQLERRMESSAGE

The SQLERRMESSAGE variable holds the error message corresponding to the last SQL error.

Syntax

```
SQLERRMESSAGE
```

Usage

The SQLERRMESSAGE predefined variable returns the error message if an SQL error occurred.

The variable is NULL if the last SQL statement was successfull.

Example

```
MAIN
  DATABASE stores
  WHENEVER ERROR CONTINUE
  SELECT foo FROM bar
  DISPLAY SQLERRMESSAGE
END MAIN
```

Data type operators

Data type operators allow you cast a data type or create an instance of a data type.

Table 117: Data type operators

Operator	Description
CAST on page 257	The CAST operator converts a Java™ object to the user-defined type or Java class specified.
INSTANCEOF on page 258	The INSTANCEOF checks the class of an object.

CAST

The CAST operator converts a Java™ object to the user-defined type or Java class specified.

Syntax

```
CAST( obj AS type )
```

1. *obj* is a Java object.
2. *type* is a user-defined type or a Java class.

Usage

The CAST( ) operator is required when you want to assign an object reference to variable defined with a user-defined type or Java class which requires narrowing reference conversion.

Example

In this example, when assigning a `java.lang.StringBuffer` reference to a `java.lang.Object` variable, widening reference conversion occurs and no CAST( ) operator is needed, but when assigning an `java.lang.Object` reference to a `java.lang.StringBuffer` variable, you must cast the object reference to a `java.lang.StringBuffer`:

```
IMPORT JAVA java.lang.Object
```

```
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE sb1, sb2 java.lang.StringBuffer
  DEFINE o java.lang.Object
  LET sb1 = StringBuffer.create()
  LET o = sb1 -- Widening Reference Conversion does not need CAST()
  LET sb2 = CAST( o AS java.lang.StringBuffer ) -- Narrowing
                  -- Reference Conversion needs CAST()
END MAIN
```

In order to cast an `fgl.FglRecord` object to a regular `RECORD`, you need to specify a user-defined type (`TYPE` definition):

```
IMPORT JAVA com.fourjs.fgl.lang.FglRecord
TYPE mytype RECORD f1, f2 INTEGER END RECORD
MAIN
  DEFINE r mytype
  DEFINE jr fgl.FglRecord
  LET jr = r
  LET r = CAST(jr AS mytype)
  -- This is denied:
  --   CAST(jr AS RECORD f1, f2 INTEGER END RECORD)
END MAIN
```

INSTANCEOF

The `INSTANCEOF` checks the class of an object.

Syntax

```
expr INSTANCEOF type
```

1. *expr* can be any expression supported by the language.
2. *type* is a structured user defined type or a Java™ class.

Usage

The `INSTANCEOF` operator evaluates to `TRUE` if the object reference is of the type or class specified.

The `INSTANCEOF` operator is used to check if an expression (usually, an object reference) is one of the type or class specified by *type*.

Example

```
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
IMPORT JAVA java.lang.Number
MAIN
  DEFINE o java.lang.Object
  DEFINE sb java.lang.StringBuffer
  LET sb = StringBuffer.create()
  LET o = sb
  DISPLAY sb INSTANCEOF java.lang.StringBuffer  -- shows 1
  DISPLAY o INSTANCEOF java.lang.StringBuffer   -- shows 1
  DISPLAY o INSTANCEOF java.lang.Number         -- shows 0
END MAIN
```

Assignment operators

An assignment operator allows you to assign a variable with an expression.

Table 118: Assignment operators

Operator	Description
Assignment (:=) on page 259	The := operator assigns a variable with an expression and returns the result.

Assignment (:=)

The := operator assigns a variable with an expression and returns the result.

Syntax

```
variable := expr
```

Usage

The := assignment operator puts a value in the left-hand variable and the resulting value can again be used in an expression.

Do not confuse with the LET instruction.

The := assignment operator has the lowest precedence, it can be used at many places and can simplify coding.

Example

In the next example, the := operator is used to increment the array index before usage:

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF STRING,
         idx INTEGER
  LET idx = 0
  LET arr[idx:=idx+1] = "One"
  LET arr[idx:=idx+1] = "Two"
  LET arr[idx:=idx+1] = "Three"
END MAIN
```

Date and time operators

Date and time operators allow you to work with date and time values.

Table 119: Date and time operators

Operator	Description
CURRENT on page 260	The CURRENT operator returns the current system date and time.
EXTEND() on page 260	The EXTEND( ) operator adjusts a date time value according to the qualifier.
DATE() on page 261	The DATE( ) operator converts an expression to a DATE value.
TIME() on page 261	The TIME( ) operator returns a time part of the date time expression.

Operator	Description
TODAY on page 262	The TODAY operator returns the current calendar date.
YEAR() on page 262	The YEAR( ) operator extracts the year of a date time expression.
MONTH() on page 263	The MONTH( ) operator extracts the month of a date time expression.
DAY() on page 263	The DAY( ) operator extracts the day of the month of a date time expression.
WEEKDAY() on page 263	The WEEKDAY( ) operator extracts the day of the week of a date time expression.
MDY() on page 264	The MDY( ) operator creates a date from month, day and year units.
UNITS on page 264	The UNITS operator converts an integer to an interval.

CURRENT
The CURRENT operator returns the current system date and time.

Syntax

```
CURRENT [ qual1 TO qual2 [(scale)] ]
```

1. *qual1*, *qual2* and *scale* define the date time qualifier.

Usage

The CURRENT operator returns the system date/time in the current local timezone.

This operator can be used to assign the current system date and time to a DATETIME variable.

Use optional datetime qualifiers to specify the precision of the returned value. The possible qualifiers are the same as in a DATETIME data type definition.

If the datetime qualifiers are not specified after the CURRENT keyword, the precision defaults to YEAR TO FRACTION(3) precision.

Example

```
MAIN
  DISPLAY CURRENT YEAR TO FRACTION(4)
  DISPLAY CURRENT HOUR TO SECOND
  DISPLAY CURRENT
END MAIN
```

EXTEND()
The EXTEND( ) operator adjusts a date time value according to the qualifier.

Syntax

```
EXTEND ( dt-expr, qual1 TO qual2 [(scale)])
```

1. *dt-expr* is a date / time expression.
2. *qual1*, *qual2* and *scale* define the date time qualifier.

Usage

The `EXTEND()` operator is used to convert a date time expression to a `DATETIME` value with a different precision.

The default qualifier is `YEAR TO DAY`.

The possible qualifiers are the same as in a `DATETIME` data type definition.

The expressions passed as first parameter must be a valid datetime value. If it is a character string, it must consist of valid and unambiguous time-unit values and separators, but with these restrictions:

- It cannot be a character string in date format, such as "12/12/99".
- It cannot be an ambiguous numeric datetime value, such as "05:06" or "05".
- It cannot be a time expression that returns an `INTERVAL` value.

Example

```
MAIN
  DISPLAY EXTEND ( TODAY, YEAR TO FRACTION(4) )
END MAIN
```

DATE()

The `DATE()` operator converts an expression to a `DATE` value.

Syntax

```
DATE [(expr)]
```

1. *expr* is the expression to be converted to a date.

Usage

`DATE()` converts a character string, an integer or datetime expression to a `DATE` value.

When *expr* is a character string expression, it must properly formatted according to datetime format settings like DBDATE.

If *expr* is an integer expression, it is used as the number of days since December 31, 1899.

If you supply no operand, it returns a character representation of the current date in the format "weekday month day year".

Example

```
MAIN
  DISPLAY DATE ( 34000 )
  DISPLAY DATE ( "12/04/1978" )
  DISPLAY DATE ( CURRENT )
END MAIN
```

TIME()

The `TIME()` operator returns a time part of the date time expression.

Syntax

```
TIME [(datetime-expr)]
```

1. *datetime-expr* is a datetime expression.

Usage

`TIME()` converts the time-of-day portion of its datetime operand to a character string.

This operator converts a date time expression to a character string representing the time-of-day part of its operand.

The format of the returned string is always "hh:mm:ss".

If you supply no operand, it returns a character representation of the current time. You can use the `CURRENT` operator to get a datetime result of the current system time.

Example

```
MAIN
  DISPLAY TIME ( CURRENT )
END MAIN
```

TODAY

The `TODAY` operator returns the current calendar date.

Syntax

```
TODAY
```

Usage

`TODAY` returns the current system date as a `DATE` value, in the current local timezone.

This operator can be used to assign the current system date to a `DATE` variable.

The `TODAY` operator is the `DATE` equivalent for the `CURRENT` operator used for `DATETIME`.

Example

```
MAIN
  DISPLAY TODAY
END MAIN
```

YEAR()

The `YEAR()` operator extracts the year of a date time expression.

Syntax

```
YEAR ( expr )
```

1. *expr* is a date / time expression.

Usage

Returns an integer corresponding to the year portion of its operand.

Example

```
MAIN
  DISPLAY YEAR ( TODAY )
  DISPLAY YEAR ( CURRENT )
```

```
END MAIN
```

MONTH()
The `MONTH()` operator extracts the month of a date time expression.

Syntax

```
MONTH ( expr )
```

1. *expr* is a date / time expression.

Usage

Returns a positive whole number between 1 and 12 corresponding to the month of its operand.

Example

```
MAIN
  DISPLAY MONTH ( TODAY )
  DISPLAY MONTH ( CURRENT )
END MAIN
```

DAY()
The `DAY()` operator extracts the day of the month of a date time expression.

Syntax

```
DAY ( expr )
```

1. *expr* is a date / time expression.

Usage

Returns a positive whole number between 1 and 31 corresponding to the day of the month of its operand.

Example

```
MAIN
  DISPLAY DAY ( TODAY )
  DISPLAY DAY ( CURRENT )
END MAIN
```

WEEKDAY()
The `WEEKDAY()` operator extracts the day of the week of a date time expression.

Syntax

```
WEEKDAY ( expr )
```

1. *expr* is a date / time expression.

Usage

Returns a positive whole number between 0 and 6 corresponding to the day of the week implied by its operand.

The integer 0 (Zero) represents Sunday.

Example

```
MAIN
  DISPLAY WEEKDAY( TODAY )
  DISPLAY WEEKDAY( CURRENT )
END MAIN
```

MDY()

The `MDY()` operator creates a date from month, day and year units.

Syntax

```
MDY ( expr1, expr2, expr3 )
```

1. *expr1* is an integer representing the month (from 1 to 12).
2. *expr2* is an integer representing the day (from 1 to 28, 29, 30 or 31 depending on the month).
3. *expr3* is an integer representing the year (four digits).

Usage

The `MDY()` operator builds a date value with 3 integers representing the month, day and year.

The result is a `DATE` value.

This function is sensitive to the `C1` modifier of the DBDATE environment variable, defining a Ming Guo date format.

Example

```
MAIN
  DISPLAY MDY ( 12, 3+2, 1998 )
END MAIN
```

UNITS

The `UNITS` operator converts an integer to an interval.

Syntax

```
expr UNITS qual[(scale)]
```

where *qual* can be one of:

```
  YEAR
  MONTH
  DAY
  HOUR
  MINUTE
  SECOND
  FRACTION(1-6)
```

1. *expr* is an integer expression.

Usage

The `UNITS` operator converts an integer expression to an `INTERVAL` value expressed in a single unit of time that you specify after the `UNITS` keyword.

For the qualifiers YEAR, MONTH, DAY, HOUR and SECOND, if the left-hand expression evaluates to a decimal number, any fractional part is discarded before the UNITS operator is applied. However, when using UNITS FRACTION, the expression can be a decimal number where the integer part is interpreted as a number of seconds, and the decimal part as the fraction of a second.

UNITS has a higher precedence than any arithmetic or boolean operator. As a result, a left-hand arithmetic expression that uses a UNITS operator must be enclosed in parentheses. For example, `10 + 20 UNITS MINUTES` will be evaluated as `10 + (20 UNITS MINUTES)` and give a conversion error. It must be written `(10 + 20) UNITS MINUTES` to get the expected result.

Because the difference between two DATE values is an integer count of days rather than an INTERVAL data type, you might want to use the UNITS operator to convert such differences explicitly to INTERVAL values.

Example

```
MAIN
  DEFINE d DATE
  LET d = TODAY + 200
  DISPLAY (d - TODAY) UNITS DAY
END MAIN
```

Dialog handling operators

Dialog handling operators allow you to handle variables in a DIALOG statement.

Table 120: Dialog handling operators

Operator	Description
GET_FLDBUF() on page 265	The GET_FLDBUF( ) operator returns as character strings the current values of the specified fields.
INFIELD() on page 266	The INFIELD( ) operator checks for the current screen field.
FIELD_TOUCHED() on page 266	The FIELD_TOUCHED( ) operator checks if fields were modified during the dialog execution.

GET_FLDBUF()

The GET_FLDBUF( ) operator returns as character strings the current values of the specified fields.

Syntax

```
GET_FLDBUF ( [group.]field  [,...] )
```

1. *group* can be a table name, a screen record, a screen array or FORMONLY.
2. *field* is the name of the screen field.

Usage

The GET_FLDBUF( ) operator is used to get the value of a screen field before the input buffer is copied into the associated variable.

> **Note:** This operator should only be used in dialogs allowing field input (INPUT, INPUT ARRAY, CONSTRUCT). The behavior is undefined when used in DISPLAY ARRAY.

The GET_FLDBUF( ) operator takes the field names as identifiers, not as string expressions:

```
LET v = GET_FLDBUF( customer.custname )
```

If multiple fields are specified between parentheses, use the RETURNING clause:

```
CALL GET_FLDBUF( customer.* ) RETURNING rec_customer.*
```

When used in a INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

The values returned by this operator are context dependent; it must be used carefully. If possible, use the variable associated to the input field instead.

When using the UNBUFFERED mode, program variables are automatically assigned, and the GET_FLDBUF operator is not required in most cases.

Example

```
INPUT BY NAME ...
   ...
   ON KEY(CONTROL-Z)
      LET v = GET_FLDBUF( customer.custname )
      IF check_synonyms(v) THEN
         ...
```

INFIELD()

The INFIELD( ) operator checks for the current screen field.

Syntax

```
INFIELD ( [group.]field )
```

1. *group* can be a table name, a screen record, a screen array or FORMONLY.
2. *field* is the name of the field in the form.

Usage

INFIELD checks for the current field in a CONSTRUCT, INPUT or INPUT ARRAY dialog.

When used in an INPUT ARRAY instruction, the runtime system assumes that you are referring to the current row.

For a generic coding equivalent, use the DIALOG.getCurrentItem( ) method.

Example

```
INPUT ...
  IF INFIELD( customer.custname ) THEN
     MESSAGE "The current field is customer's name."
  ...
```

FIELD_TOUCHED()

The FIELD_TOUCHED( ) operator checks if fields were modified during the dialog execution.

Syntax

```
FIELD_TOUCHED (
```

```
{ [group.]field.*
| group.*
| *
} [,...] )
```

1. *group* can be a table name, a screen record, a screen array or `FORMONLY`.
2. *field* is the name of the field in the form.

Usage

`FIELD_TOUCHED` returns `TRUE` if the value of a screen field (or multiple fields) has changed since the beginning of the interactive instruction.

The operator accepts a list of explicit field names, and supports the [*group*.]* notation in order to check multiple fields in a single evaluation. When passing a simple asterisk (*) to the operator, the runtime system will check all fields used by the current dialog.

When used in an `INPUT ARRAY` instruction, the runtime system assumes that you are referring to the current row.

The `FIELD_TOUCHED` operator can only be used inside an `INPUT`, `INPUT ARRAY` and `CONSTRUCT` interaction block.

For more details about the `FIELD_TOUCHED` operator usage and the understand the "touched flag" concept, refer to the definition of the `DIALOG` instruction.

Do not confuse the `FIELD_TOUCHED` operator with`FGL_BUFFERTOUCHED` built-in function; which checks a different field modification flag, that is reset when entering the field. The global touched flag controlled by `FIELD_TOUCHED` is reset when the dialog starts or when `DIALOG.setFieldTouched()` is used.

Example

```
INPUT ...
  ...
  AFTER FIELD custname
    IF FIELD_TOUCHED( customer.custname ) THEN
       MESSAGE "Customer name was changed."
    END IF
  ...
  AFTER INPUT
    IF FIELD_TOUCHED( customer.* ) THEN
       MESSAGE "Customer record was changed."
    END IF
  ...
```

Flow control

Definition of language elements and instructions that control the flow of a program.

- CALL on page 268
- RETURN on page 270
- CASE on page 270
- CONTINUE block-name on page 272
- EXIT block-name on page 273
- FOR on page 274
- GOTO on page 275
- IF on page 276
- LABEL on page 276

- SLEEP on page 277
- WHILE on page 277

CALL

The CALL instruction invokes a specified function or method.

Syntax

```
CALL [ prefix. ] function ( [ parameter  [,...] ] )
      [ RETURNING variable  [,...] ]
```

1. *prefix* can be an imported module, an imported C-Extension module, a built-in class, a variable referencing an object of a built-in class, a Java™ class, a variable referencing a Java™ object.
2. *function* can a function defined in one of the modules of the program, a function defined in one of the modules of the program, a C function defined in a C extension module, a built-in function of the language, a built-in class or object method of the language or a Java™ class or object method of an imported Java™ class.
3. *parameter* can be any valid expression, including object references of built-in classes or Java™ classes.
4. *variable* is a variable receiving a value returned by the function.

Usage

The CALL instruction invokes the function or class/object method specified and passes the program flow control to that function/method. After the called function was executed, the flow control goes back to the caller, the runtime system executing the next statement that appears after the CALL instruction.

Function arguments can be any expression supported by the language. Use a double-pipe operator || to pass the concatenation of character string expressions as a parameter.

```
CALL my_function( TODAY, 20*0.5435, 'abc'||'def'||var1 )
```

The RETURNING clause assigns values returned by the function to variables in the calling routine. The RETURNING clause is only needed when the function returns parameters.

```
MAIN
  DEFINE var1 CHAR(15)
  DEFINE var2 CHAR(15)
  CALL foo() RETURNING var1, var2
  DISPLAY var1, var2
END MAIN

FUNCTION foo()
  DEFINE r1 CHAR(15)
  DEFINE r2 CHAR(15)
  LET r1 = "return value 1"
  LET r2 = "return value 2"
  RETURN r1, r2
END FUNCTION
```

If the function returns a unique parameter, the function can be used in an expression and can be directly assigned to a variable with `LET var = function(...)` statement.

```
MAIN
  DEFINE var1 CHAR(10)
  DEFINE var2 CHAR(2)
  LET var1 = foo()
  DISPLAY "var1 = " || var1
  CALL foo() RETURNING var2
```

```
  DISPLAY "var2 = " || var2
END MAIN

FUNCTION foo()
  RETURN "Hello"
END FUNCTION
```

The value of a receiving variable may be different from the value returned by the function, following the data conversion rules.

```
MAIN
  DEFINE s STRING
  LET s = div(10,2)
END MAIN

FUNCTION div(x,y)
  DEFINE x,y INTEGER
  RETURN x / y
END FUNCTION
```

Records can be passed to and returned from functions, but the record structure must be flat and each member is passed or returned individually by value. Records are not passed by reference.

```
MAIN
  DEFINE r RECORD
             x INT,
             y INT,
             z INT
           END RECORD
  CALL foo(r.*) RETURNING r.*
  DISPLAY r.*
END MAIN

FUNCTION foo(x,y,z)
  DEFINE x,y,z INT
  RETURN z,y,x
END FUNCTION
```

If the `IMPORT FGL` instruction was used to import a module, *function* can be prefixed with the name of the module followed by a dot (i.e. *module.function)*. The module prefix is required to fully-qualify the function in case of conflicts (i.e. when functions with the same name are defined in several modules).

```
-- main.4gl
IMPORT FGL module1
IMPORT FGL module2
MAIN
  CALL module1.show("aaa")
  CALL module2.show("aaa")
END MAIN
```

```
-- module1.4gl
FUNCTION show(s)
  DEFINE s STRING
  DISPLAY s
END FUNCTION
```

```
-- module2.4gl
FUNCTION show(s)
  DEFINE s STRING
  DISPLAY s
```

```
END FUNCTION
```

RETURN

The RETURN instruction returns flow control to the function caller.

Syntax

```
RETURN [ value [,...]  ]
```

1. *value* can be any valid expression, an object reference or dynamic array reference.

Usage

The RETURN instruction transfers the control back from a function with optional return values.

Record members can be returned with the .* or THRU notation. Each member is returned as an independent variable.

A function may have several RETURN points (not recommended in structured programming) but they must all return the same number of values.

The number of returned values must correspond to the number of variables listed in the RETURNING clause of the CALL statement invoking this function.

A function cannot return a static array, but can return the reference of a dynamic array.

Example

```
MAIN
  DEFINE fname, lname VARCHAR(30)
  CALL foo(NULL) RETURNING fname, lname
  DISPLAY fname CLIPPED, " ", upshift(lname) CLIPPED
  CALL foo(1) RETURNING forname, surname
  DISPLAY fname CLIPPED, " ", upshift(lname) CLIPPED
END MAIN

FUNCTION foo(code)
  DEFINE code INTEGER
  DEFINE person RECORD
                  fname VARCHAR(30),
                  lname VARCHAR(30)
                END RECORD
  IF code IS NULL THEN
     RETURN NULL, NULL
  ELSE
     LET person.fname = "John"
     LET person.lname = "Smith"
     RETURN person.*
  END IF
END FUNCTION
```

CASE

The CASE instruction specifies statement blocks that must be executed conditionally.

Syntax 1

```
CASE expression-1
  WHEN expression-2
```

```
    { statement | EXIT CASE }
    [...]
  [ OTHERWISE
    { statement | EXIT CASE }
    [...]
  ]
END CASE
```

Syntax 2

```
CASE
  WHEN boolean-expression
    { statement | EXIT CASE }
    [...]
  [ OTHERWISE
    { statement | EXIT CASE }
    [...]
  ]
END CASE
```

1. *expression-1* is any expression supported by the language.
2. *expression-2* is an expression that is tested against *expression-1*.
3. *expression-1* and *expression-2* should have the same data type.
4. *boolean-expression* is any boolean expression supported by the language.
5. *statement* is any instruction supported by the language.

Usage

In a `CASE` flow control block, the first matching `WHEN` block is executed. If there is no matching `WHEN` block, then the `OTHERWISE` block is executed. If there is no matching `WHEN` block and no `OTHERWISE` block, the program execution continues with the next statement following the `END CASE` keyword.

The `EXIT CASE` statement transfers the program control to the statement following the `END CASE` keyword. There is an implicit `EXIT CASE` statement at the end of each `WHEN` block and at the end of the `OTHERWISE` block. The `OTHERWISE` block must be the last block of the `CASE` instruction.

A null expression is considered as false: When doing a `CASE expr ... WHEN [NOT] NULL` using the syntax 1, it always evaluates to `FALSE`. Use syntax 2 as `CASE ... WHEN expr IS NULL` to test if an expression is null.

Make sure that *expression-2* is not a boolean expression when using the first syntax. The compiler will not raise an error in this case, but you might get unexpected results at runtime.

If there is more than one *expression-2* matching *expression-1* (syntax 1), or if two boolean expressions (syntax 2) are true, only the first matching `WHEN` block will be executed.

Example

```
MAIN
   DEFINE v CHAR(10)
   LET v = "C1"
   -- CASE Syntax 1
   CASE v
     WHEN "C1"
       DISPLAY "Value is C1"
     WHEN "C2"
       DISPLAY "Value is C2"
     WHEN "C3"
       DISPLAY "Value is C3"
     OTHERWISE
```

```
        DISPLAY "Unexpected value"
    END CASE
    -- CASE Syntax 2
    CASE
      WHEN ( v="C1" OR v="C2" )
        DISPLAY "Value is either C1 or C2"
      WHEN ( v="C3" OR v="C4" )
        DISPLAY "Value is either C3 or C4"
      OTHERWISE
        DISPLAY "Unexpected value"
    END CASE
END MAIN
```

CONTINUE *block-name*

The `CONTINUE` *`block-name`* instruction resumes execution of a loop or dialog statement.

Syntax

```
CONTINUE
 { FOR
 | FOREACH
 | WHILE
 | MENU
 | CONSTRUCT
 | INPUT
 | DIALOG
 }
```

Usage

The `CONTINUE` *`block-name`* instruction transfers the program execution from a statement block to another location in the compound statement that is currently being executed.

`CONTINUE` *`block-name`* can only be used within the statement block specified by *block-name*. For example, `CONTINUE FOR` can only be used within a `FOR ... END FOR` statement block.

The `CONTINUE FOR`, `CONTINUE FOREACH`, or `CONTINUE WHILE` keywords cause the current `FOR`, `FOREACH`, or `WHILE` loop (respectively) to begin a new cycle immediately. If conditions do not permit a new cycle, however, the looping statement terminates.

The `CONTINUE CONSTRUCT`, `CONTINUE INPUT` and `CONTINUE DIALOG` statements cause the program to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field.

The `CONTINUE MENU` statement causes the program to ignore the remaining statements in the current `MENU` control block and re-display the menu. The user can then choose another menu option.

`CONTINUE INPUT` is valid in `INPUT` and `INPUT ARRAY` statements.

Example

```
MAIN
  DEFINE i INTEGER
  LET i = 0
  WHILE i < 5
    LET i = i + 1
    DISPLAY "i=" || i
    CONTINUE WHILE
    DISPLAY "This will never be displayed!"
  END WHILE
```

```
END MAIN
```

EXIT *block-name*

The EXIT *block* instruction transfers control out of the current program block.

Syntax

```
EXIT
 { CASE
 | FOR
 | FOREACH
 | WHILE
 | MENU
 | CONSTRUCT
 | REPORT
 | DISPLAY
 | INPUT
 | DIALOG }
```

Usage

The EXIT *block-name* instruction transfers control out of a control structure (a block, a loop, a CASE statement, or an interface instruction).

The EXIT *block-name* instruction must be used inside the control structure specified by *block-name*. For example, EXIT FOR can only appear inside a FOR ... END FOR iteration block.

EXIT DISPLAY exits the DISPLAY ARRAY instruction and EXIT INPUT exits an INPUT or an INPUT ARRAY block.

EXIT CONSTRUCT exits current CONSTRUCT block.

EXIT DIALOG exits current DIALOG block.

To exit a function, use the RETURN instruction. To terminate a program, use the EXIT PROGRAM instruction.

Example

```
MAIN
  DEFINE i INTEGER
  LET i = 0
  WHILE TRUE
    DISPLAY "This is an infinite loop. How would you get out of
 here?"
    LET i = i + 1
    IF i = 100 THEN
       EXIT WHILE
    END IF
  END WHILE
  DISPLAY "Done."
END MAIN
```

FOR

The FOR instruction executes a statement block a specified number of times.

Syntax

```
FOR counter = start TO finish [ STEP value ]
   { statement
   | EXIT FOR
   | CONTINUE FOR }
   [...]
END FOR
```

1. *counter* is the loop counter and must be an integer variable.
2. *start* is an integer expression used to set an initial counter value.
3. *finish* is any valid integer expression used to specify an upper limit for *counter*.
4. *value* is any valid integer expression whose value is added to *counter* after each iteration of the statement block.
5. When the STEP keyword is not given, *counter* increments by 1.
6. *statement* is any instruction supported by the language.
7. If *value* is less than 0, *counter* is decreased. In this case, *start* should be higher than *finish*.

Usage

The FOR instruction block executes the statements up to the END FOR keyword a specified number of times, or until EXIT FOR terminates the FOR statement. The CONTINUE FOR instruction skips the next statements and continues with the next iteration.

On the first iteration through the loop, the counter is set to the initial expression at the left of the TO keyword. For all further iterations, the value of the increment expression in the STEP clause specification (1 by default) is added to the counter in each pass through the block of statements. When the sign of the difference between the values of counter and the finish expression at the right of the TO keyword changes, the runtime system exits from the FOR loop.

The FOR loop terminates after the iteration for which the left- and right-hand expressions are equal. Execution resumes at the statement following the END FOR keywords. If either expression returns NULL, the loop cannot terminate, because the boolean expression "left = right" cannot become TRUE.

A *value* that equals 0 causes an unending loop unless there is an adequate EXIT FOR statement.

Using NULL for *start*, *finish* or *value* is treated as 0. There is no way to catch this as an error.

If *statement* modifies the value of *counter*, you might get unexpected results at runtime. In this case, it is recommended that you use a WHILE loop instead.

It is highly recommended that you ensure that *statement* does not modify the values of *start*, *finish* or *value*.

Example

```
MAIN
  DEFINE i, i_min, i_max INTEGER
  LET i_min = 1
  LET i_max = 10
  DISPLAY "Count from " || i_min || " to " || i_max
  DISPLAY "Counting forwards..."
  FOR i = i_min TO i_max
      DISPLAY i
  END FOR
  DISPLAY "... and backwards."
  FOR i = i_max TO i_min STEP -1
```

```
      DISPLAY i
  END FOR
END MAIN
```

GOTO

The `GOTO` instruction transfers program control to a labeled line within the same program block.

Syntax

```
GOTO [:] label-id
```

1. *label-id* is the name of the `LABEL` statement to jump to.

Usage

A `GOTO` statement continues program execution in the line following the `LABEL` instruction using the *label-id* identifier specified in the `GOTO` instruction.

The `LABEL` jump point can be defined before or after the `GOTO` statement.

The `LABEL` and `GOTO` statements must use the *label-id* within a single `MAIN`, `FUNCTION`, or `REPORT` program block.

The `:` colon after the `GOTO` keyword is optional.

`GOTO` statements can reduce the readability of your program source and result in infinite loops. It is recommended that you use `FOR`, `WHILE`and `CASE` statements instead.

The `GOTO` statement can be used in a `WHENEVER` statement to handle exceptions.

Example

```
MAIN
  DEFINE exit_code INTEGER
  DEFINE l_status INTEGER

  WHENEVER ANY ERROR GOTO _error
  DISPLAY 1/0
  GOTO _noerror

LABEL _error:
  LET l_status = STATUS
  DISPLAY "The error number ", l_status, " has occurred."
  DISPLAY "Description: ", err_get(l_status)
  LET exit_code = -1
  GOTO _exit

LABEL _noerror:
  LET exit_code = 0
  GOTO _exit

LABEL _exit:
  EXIT PROGRAM exit_code

END MAIN
```

IF

The IF instruction executes a group of statements conditionally.

Syntax

```
IF condition THEN
   statement
   [...]
[ ELSE
   statement
   [...]
]
END IF
```

1. *condition* is a boolean expression.
2. *statement* is any instruction supported by the language.

Usage

If *condition* is TRUE, the runtime system executes the block of statements following the THEN keyword, until it reaches either the ELSE keyword or the END IF keywords and resumes execution after the END IF keywords.

If *condition* is FALSE, the runtime system executes the block of statements between the ELSE keyword and the END IF keywords. If ELSE is absent, it resumes execution after the END IF keywords.

By default, the runtime system evaluates all part of the condition. The semantics of boolean expressions can be controlled by the OPTIONS SHORT CIRCUIT compiler directive, to reduce expression evaluation when using AND / OR operators.

A NULL expression is considered as FALSE. Use the IS NULL keyword to test if an expression is null.

Example

```
MAIN
  DEFINE name CHAR(20)
  LET name = "John Smith"
  IF name MATCHES "John*" THEN
    DISPLAY "The name starts with [John]!"
  ELSE
    DISPLAY "The name is " || name || "."
  END IF
END MAIN
```

LABEL

The LABEL instruction declares a jump point that can be reached by a GOTO.

Syntax

```
LABEL label-id:
```

1. *label-id* is a unique identifier in a MAIN, REPORT, or FUNCTION program block.
2. The *label-id* must be followed by a colon (:).

Usage

The LABEL instruction declares a statement label, making the next statement one to which a GOTO statement can transfer program control.

Example

```
MAIN
  DISPLAY "Line 2"
  GOTO line5
  DISPLAY "Line 4"
  LABEL line5:
  DISPLAY "Line 6"
END MAIN
```

SLEEP

The SLEEP instruction causes the program to pause for the specified number of seconds.

Syntax

```
SLEEP seconds
```

1. *seconds* must be an integer expression.

Usage

The SLEEP instruction is typically invoked to let the end user read a message displayed on a character terminal.

With graphical applications, the SLEEP command is seldom used.

When *seconds* is lower than zero or is null, the program continues immediately with the next statement.

Example

```
MAIN
  DISPLAY "Please wait 5 seconds..."
  SLEEP 5
  DISPLAY "Thank you."
END MAIN
```

WHILE

The WHILE statement executes a block of statements until the specified condition becomes false.

Syntax

```
WHILE condition
    { statement | EXIT WHILE | CONTINUE WHILE }
   [...]
END WHILE
```

1. *condition* must be a boolean expression.
2. *statement* is any instruction supported by the language.

Usage

As long as the *condition* specified after a `WHILE` keyword is `TRUE`, all statements inside the `WHILE ... END WHILE` block are executed. After executing the last statement of the block, the runtime system again evaluates the condition, and if it is still `TRUE`, continues with the first statement in the block.

The loop stops when the condition becomes `FALSE` or when an `EXIT WHILE` is reached.

Use the `CONTINUE WHILE` instruction to skip the next statements and continue with the loop.

To avoid unending loops, make sure that the condition will become `FALSE` at some point, or that an `EXIT WHILE` statement will be executed.

Example

```
MAIN
  DEFINE cnt INTEGER
  LET cnt = 1
  WHILE cnt <= 100
    DISPLAY "Iter: " || cnt
    LET cnt = cnt + 1
    IF int_flag THEN
      EXIT WHILE
    END IF
  END WHILE
END MAIN
```

Functions

Describes the basics of user defined functions in the language.

- Understanding functions on page 278
- FUNCTION blocks on page 278
- Using functions in programs on page 279
- Examples on page 280

Understanding functions

Functions are named program blocks containing a set of statements to be executed when the function is invoked with a `CALL` statement, or when the function is used in an expression, or when the function is registered in a callback mechanism like `WHENEVER ERROR CALL`.

A functions is defined in a program module, and is by default visible to all modules (i.e. a function is global by default), but it can also be declared as private to the module where it is defined.

FUNCTION blocks

A `FUNCTION` block defines a named procedure with a set of statements.

Syntax

```
[PUBLIC|PRIVATE] FUNCTION function-name ( [ argument [,...]] )
    [ declaration [...] ]
    [ statement [...] ]
    [ return-clause ]
END FUNCTION
```

where *return-clause* is:

```
RETURN expression [,...]
```

1. *function-name* is the function identifier.
2. *argument* is the name of a formal argument of the function.
3. *declaration* is a `DEFINE`, `CONSTANT` or `TYPE` instruction.
4. *statement* is any instruction supported by the language.

Using functions in programs

The `FUNCTION` block defines the body and the signature (i.e. declaration) of a function. The function declaration specifies the name of the function and the identifiers of its formal arguments (if any).

Function names, like other identifiers are case-insensitive. If the function name is also the name of a built-in function, an error occurs at link time, even if the program does not reference the built-in function.

A `FUNCTION` block cannot appear within the `MAIN` block, in a `REPORT` block, or within another `FUNCTION` block.

A function can be invoked with the `CALL` statement, it can be used in an expression when returning a unique value, or it can be invoked automatically when registered by a callback mechanism like `WHENEVER ERROR CALL`.

If no argument is needed in a function call, an empty argument list must still be supplied, enclosed between the parentheses.

By default, functions are public; They can be called by any other module of the program. If a function is only used by the current module, you may want to hide that function to other modules, to make sure that it will not be called by mistake. To keep a function local to the module, add the `PRIVATE` keyword before the function header. Private functions are only hidden to external modules, all function of the current module can still call local private functions.

```
PRIVATE FUNCTION check_number(n)
  ...
END FUNCTION
```

The data type of each formal argument of the function must be specified by a `DEFINE` statement that immediately follows the argument list. The actual argument in a call to the function need not be of the declared data type of the formal argument. If data type conversion is not possible, a runtime error occurs.

```
FUNCTION check_address(zipcode, street, city)
  DEFINE zipcode CHAR(5),
         street VARCHAR(100),
         city VARCHAR(50)
  ...
END FUNCTION
```

Function arguments are passed by value (i.e. value is copied on the stack) for basic data types and records, while dynamic arrays and objects are passed by reference (i.e. a handle to the original data is copied on the stack and thus allows modification of the original data inside the function).

```
-- The following code is useless:
-- Variable x will not be modified by the function
MAIN
  DEFINE x INTEGER
  LET x = 123
  CALL increment(x)
  DISPLAY x   -- displays 123
END MAIN
```

```
FUNCTION increment(x)
  DEFINE x INTEGER
  LET x = x + 1
END FUNCTION
```

Local variables are not visible in other program blocks. The identifiers of local variables must be unique among the variables that are declared in the same `FUNCTION` definition. Any global or module variable that has the same identifier as a local variable, however, is not visible within the scope of the local variable.

```
DEFINE x INTEGER   -- Declares a module variable

FUNCTION func_a()
  DEFINE x INTEGER -- Declares a local variable
  LET x = 123      -- Assigns local variable
END FUNCTION

FUNCTION func_b()
  LET x = 123     -- Changes the module variable
END FUNCTION
```

A function that returns one or more values to the calling routine must include the *return-statement* . Values specified in `RETURN` must correspond in number and position, and must be of the same or of compatible data types , to the variables in the `RETURNING` clause of the `CALL` statement. If the function returns a single value, it can be invoked as an operand within a expression. Otherwise, you must invoke it with the `CALL` statement with a `RETURNING` clause. An error results if the list of returned values in the `RETURN` statement conflicts in number or in data type with the `RETURNING` clause of the `CALL` statement that invokes the function.

```
MAIN
  DEFINE zipcode CHAR(5),
         street VARCHAR(100),
         city VARCHAR(50)
  CALL get_address() RETURNING zipcode, street, city
END MAIN

FUNCTION get_address()
   ...
   RETURN "23500", "461 Ocean blvd", "Kreistone"
END FUNCTION
```

A function can invoke itself recursively with a `CALL` statement. This will result in a recursive call.

Examples

Example 1: Function fetching customer number

```
FUNCTION findCustomerNumber(name)
  DEFINE name VARCHAR(50)
  DEFINE num INTEGER
  CONSTANT sqltxt = "SELECT cust_num FROM customer WHERE cust_name = ?"
  PREPARE stmt FROM sqltxt
  EXECUTE stmt INTO num USING name
  IF SQLCA.SQLCODE = 100 THEN
     LET num = -1
  END IF
  RETURN num
END FUNCTION
```

Example 2: Private function definition

This function will not be visible to other modules:

```
PRIVATE FUNCTION checkIdentifier(name)
  DEFINE name VARCHAR(50)
  IF length(name) == 0 THEN
    RETURN FALSE
  ELSE
    RETURN TRUE
  END IF
END FUNCTION
```

Variables

Explains how to define program variables.

- Understanding variables on page 281
- DEFINE on page 281
- Declaration context on page 282
- Structured types on page 283
- Database column types on page 283
- User defined types on page 284
- Variable initialization values on page 284
- INITIALIZE on page 285
- LOCATE (for TEXT/BYTE) on page 286
- FREE (for TEXT/BYTE) on page 287
- LET on page 288
- VALIDATE on page 288
- THRU operator on page 289
- Examples on page 290

Understanding variables

A variable is a program element that can hold volatile data. The following list summarizes variables usage:

- Variables are declared in programs with the `DEFINE` instruction.
- After definition, variables get default values according their type.
- The scope of a variable can be global, local to a module, or local to a function.
- When defined at the module level, a variable can be declare it as `PRIVATE` or `PUBLIC`.
- You can define structured variables with records, and with arrays.
- Default values (or NULL) can be assigned with the `INITIALIZE` instruction.
- Direct value assignment is done with the `LET` instruction.
- Database validation rules can be applied with the `VALIDATE` instruction.
- Variables can be used as SQL parameters or fetch buffers in SQL statements.
- Interactive instructions use program variables as model to hold the data.

DEFINE

A variable contains volatile information of a specific data type.

Syntax

```
[PUBLIC|PRIVATE] DEFINE variable-definition [,...]
```

where *variable-definition* is:

```
identifier [,...]
   {
     datatype
   |
     LIKE [dbname:]tabname.colname
   }
   [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
```

1. *identifier* is the name of the variable to be defined.
2. *datatype* can be a data type, a record definition, an array definition, a user defined type, a built-in class, an imported package class, or a Java™ class.
3. *dbname* identifies a specific database schema file.
4. *tabname.colname* can be any column reference defined in the database schema file.
5. *attribute* is an attribute to extend the variable definition with properties.
6. *value* is the value for the variable attribute, it is optional for boolean attributes.

Usage

A *variable* is a named location in memory that can store a single value, or an ordered set of values. Variables can be global to the program, module-specific, or local to a function.

You cannot reference any program variable before it has been declared by the `DEFINE` statement.

By default, module-specific variables are private; They cannot be used by an other module of the program. In order to improve code re-usability by data encapsulation, we recommend you to keep module variables private, except if you want to share large data (like arrays) between modules. To make a module variable public, add the `PUBLIC` keyword before `DEFINE`. When a module variable is declared as public, it can be referenced by another module by using the `IMPORT` instruction.

When defining variables with the `LIKE` clause, the data types are taken from the database schema file at compile time. Make sure that the schema file of the database schema during development corresponds to the database schema of the production database; otherwise the variables defined in the compiled version of your modules will not match the table structures of the production database.

To write well-structured programs, avoid global variables. If you need persistent data storage during a program's execution, use variables local to the module and give access to them with functions, or make the module variables `PUBLIC` to other modules.

Variables can be defined with the `ATTRIBUTES()` clause, to specify meta-data information for the variable. This feature is especially used when defining variables for XML-based Web Services. For more details about XML attributes, see Attributes to customize XML serialization on page 2517.

Declaration context

The `DEFINE` statement declares the identifier of one or more variables, that will be visible to other program blocks according to the declaration context of the variables. The scope of reference of a variable defines where it can be referenced in the program. According to the location of the variable definition, memory will be allocated when the program starts, or during the program execution.

The context of a variable declaration in the source module determines where a variable can be referenced by other language statements, and when storage is allocated for the variable in memory. The `DEFINE` statement can appear in three contexts:

1. Within a `FUNCTION`, `MAIN`, or `REPORT` program block, `DEFINE` declares local variables, and causes memory to be allocated on the runtime stack when the function is called. These `DEFINE` declarations of local variables must precede any procedural statements within the same program block. The scope of reference of a local variable is restricted to the same program block. The variable is not visible elsewhere. Functions can be called recursively, and each recursive entry creates its own set of local

variables. The variable is unique to that invocation of its program block. Each time the block is entered, a new copy of the variable is created.

2. Outside any `FUNCTION`, `REPORT`, or `MAIN` program block, the `DEFINE` statement declares module variables. Module variables have a persistent state during program execution. Memory for module variables is allocated when the module is loaded. Module variable declarations (`DEFINE`) must appear before any program blocks. By default, the scope of reference is the whole module (module variables are private to the module), but it can be extended to the whole program when the variable is declared with the `PUBLIC` qualifier.
3. Inside a `GLOBALS` block, the `DEFINE` statement declares global variables that are visible to the whole program. Global variables have a persistent state during program execution. Memory for global variables is allocated when the program starts. Multiple `GLOBALS` blocks can be defined for a given module. Use one module to declare all global variables and reference that module within other modules by using the `GLOBALS "`*`filename`*`.4gl"` statement as the first statement in the module, outside any program block.

A compile-time error occurs if you declare the same name for two variables that have the same scope. You can, however, declare the same name for variables that differ in their scope. For example, you can use the same identifier to reference different local variables in different program blocks.

You can also declare the same name for two or more variables whose scopes of reference are different but overlapping. Within their intersection, the compiler interprets the identifier as referencing the variable whose scope is smaller, and therefore the variable whose scope is a superset of the other is not visible.

If a local variable has the same identifier as a global variable, then the local variable takes precedence inside the program block in which it is declared. Elsewhere in the program, the identifier references the global variable.

A module variable can have the same name as a global variable that is declared in a different module. Within the module where the module variable is declared, the module variable takes precedence over the global variable. Statements in that module cannot reference the global variable.

A module variable cannot have the same name as a global variable that is declared in the same module.

If a local variable has the same identifier as a module variable, then the local identifier takes precedence inside the program block in which it is declared. Elsewhere in the same source-code module, the name references the module variable.

If a variable needs to be persistent during program execution, instead of using global variables, consider defining that variable in the module it belongs to, by specifying the `PUBLIC` or `PRIVATE` modifiers, depending on the scope you want to give to your variable, for other modules.

Structured types

Variables can be defined as `RECORD` or `ARRAY` keywords to declare a structured object.

For example:

```
MAIN
  DEFINE myarr ARRAY[100] OF RECORD
     id INTEGER,
     name VARCHAR(100)
  END RECORD
  LET myarr[2].id = 52
END MAIN
```

Database column types

Variable defined with the `LIKE` keyword get the same data type of the column specified column in a database schema.

For example:

```
SCHEMA stores
DEFINE cname LIKE customer.cust_name
MAIN
  DEFINE cr RECORD LIKE customer.*
  ...
END MAIN
```

A SCHEMA statement must define the database name identifying the database schema files to be used.

The column data types are read from the schema file during compilation. Make sure that your schema files correspond exactly to the production database.

The database schema files must exist and must be located in one of the diretories specified in the FGLDBPATH environment variable.

When using database views, the column cannot be based on an aggregate function like SUM().

If LIKE references a SERIAL column, the variable will be defined with the INTEGER data type. If LIKE references an INT8, SERIAL8 or BIGSERIAL column, the variable will be defined with the BIGINT data type.

The table qualifier must specify *owner* if *table.column* is not a unique column identifier within its database, or if the database is ANSI-compliant and any user of your application is not the owner of *table*.

Database schema files must be generated with the fgldbsch tool before compiling the source module using a DEFINE LIKE instruction.

User defined types

Variables can be defined with a user defined type:

```
TYPE custlist DYNAMIC ARRAY OF RECORD LIKE customer.*
MAIN
  DEFINE cl custlist
  ...
END MAIN
```

The scope of a type can be global, local to a module or local to a function. Variables can be defined with a type defined in the same scope, or in a higher level of scope.

Variable initialization values

When a variable is defined, it is automatically initialized by the runtime system to a default value. The default value the variable is assigned with depends on the data type.

Table 121: data type specific default values for variables

data type	Default Value
CHAR	NULL
VARCHAR	NULL
STRING	NULL
INTEGER	Zero
SMALLINT	Zero
FLOAT	Zero
SMALLFLOAT	Zero

data type	Default Value
DECIMAL	NULL
MONEY	NULL
DATE	1899-12-31 (= Zero in number of days)
DATETIME	NULL
INTERVAL	NULL
TEXT	NULL, must use LOCATE
BYTE	NULL, must use LOCATE

INITIALIZE

The INITIALIZE instruction intializes program variables with NULL or default values.

Syntax

```
INITIALIZE target [,...]
  {
     TO NULL
  |
     LIKE {table.*|table.column}
  }
```

1. *target* is the name of the variable to be initialized.
2. *table.column* can be any column reference defined in the database schema files.

Usage

The INITIALIZE instruction assigns NULL or default values to variables.

The argument of the INITIALIZE instruction can be a simple variable, a record (with .* notation), a record member, a range of record members specified with the THRU keyword, an array or an array element.

The TO NULL clause initializes the variable to null.

When initializing a static array TO NULL, all elements will be initialized to null. When initializing a dynamic array TO NULL, all elements will be removed (i.e. the dynamic array is cleared).

The LIKE clause initializes the variable to the default value defined in the database schema validation file. This clause works only by specifying the *table.column* schema entry corresponding to the variable.

To initialize a complete record, you can use the star to reference all members:

```
INITIALIZE record.* LIKE table.*
```

You cannot initialize variables defined with a complex data type (like TEXT or BYTE) to a non-NULL value.

Example

```
SCHEMA stores
MAIN
  DEFINE cr RECORD LIKE customer.*
  DEFINE a1 ARRAY[100] OF INTEGER
  INITIALIZE cr.cust_name TO NULL
  INITIALIZE cr.cust_name THRU cr.cust_address TO NULL
  INITIALIZE cr.* LIKE customer.*
```

```
  INITIALIZE a1 TO NULL
  INITIALIZE a1[10] TO NULL
END MAIN
```

LOCATE (for TEXT/BYTE)

The LOCATE statement specifies where to store data of TEXT and BYTE variables.

Syntax 1: Locate in memory

```
LOCATE target [,...] IN MEMORY
```

Syntax 2: Locate in a specific file

```
LOCATE target [,...] IN FILE filename
```

Syntax 3: Locate in a temporary file

```
LOCATE target [,...] IN FILE
```

1. *target* is the name of a TEXT or BYTE variable to be located.
2. *filename* is a string expression defining the name of a file.

Usage

Before using TEXT and BYTE large objects, the data storage location must be specified with the LOCATE instruction. After defining the data storage, the variable can be used as input parameter or as a fetch buffer in SQL statements, as well as in interaction statements and reports.

The first syntax using the IN MEMORY clause specifies that the large object data must be located in memory.

The second syntax using the IN FILE *filename* clause specifies that the large object data must be located in a specific file.

The third syntax using the IN FILE clause specifies that the large object data must be located in a temporary file. The location of the temporary file can be defined with the DBTEMP environment variable. If DBTEMP is not defined, the default temporary directory dependents from the platform used.

The FREE instruction can be used to free the resources allocated to the large object variable.

Example

The following code example defines two TEXT variables. The first located in memory and the second located in a named file. The variables are then used in SQL statements:

```
MAIN
  DEFINE ctext1, ctext2 TEXT
  DATABASE stock
  LOCATE ctext1 IN MEMORY
  LOCATE ctext2 IN FILE "/tmp/data1.txt"
  CREATE TABLE lobtab ( key INTEGER, col1 TEXT, col2 TEXT )
  INSERT INTO lobtab VALUES ( 123, ctext1, ctext2 )
END MAIN
```

The next code example illustrates the storage semantics of BYTE and TEXT, by fetching large objects from the database into an array. Each member of the array needs to get an individual storage location, before the data is actually fetched into the LOB handler of the

array element. By using `LOCATE IN FILE`, a temporary file will be created for each large object:

```
MAIN
    DEFINE arr DYNAMIC ARRAY OF RECORD
                  id INTEGER,
                  cmt TEXT
               END RECORD,
           t TEXT, i INTEGER

    DATABASE test1

    LOCATE t IN MEMORY
    CREATE TEMP TABLE tt1 ( id INTEGER, cmt TEXT )
    LET t = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    INSERT INTO tt1 VALUES ( 1, t )
    LET t = "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"
    INSERT INTO tt1 VALUES ( 2, t )

    DECLARE c1 CURSOR FOR SELECT * FROM tt1
    LET i=1
    LOCATE arr[i].cmt IN FILE
    FOREACH c1 INTO arr[i].*
        LOCATE arr[i:=i+1].cmt IN FILE
    END FOREACH
    CALL arr.deleteElement(i)

    FOR i=1 TO arr.getLength()
        DISPLAY arr[i].*
    END FOR

END MAIN
```

FREE (for TEXT/BYTE)

The `FREE` statement releases resources allocated to the specified variable.

Syntax

```
FREE target
```

1. *target* is the name of a `TEXT` or `BYTE` variable to be freed.

Usage

When followed by a variable name, the `FREE` statement releases resources allocated to store the data of `TEXT` and `BYTE` variables.

If the `TEXT`/`BYTE` variable was located in memory, the runtime system releases the memory. If the variable was located in a file, the runtime system deletes the file.

For variables declared in a local scope of reference, the resources are automatically freed by the runtime system when returning from the function or `MAIN` block.

After freeing a `TEXT` or `BYTE` variable, it must be re-configured with a new `LOCATE` call.

Temporary files of large object are automatically deleted when the program ends.

Example

```
MAIN
```

```
  DEFINE ctext TEXT
  DATABASE stock
  LOCATE ctext IN FILE "/tmp/data1.txt"
  SELECT col1 INTO ctext FROM lobtab WHERE key=123
  FREE ctext
END MAIN
```

LET

The LET statement assigns values to variables.

Syntax

```
LET target = expr [,...]
```

1. *target* is the name of the variable to be assigned.
2. *expr* is any valid expression supported by the language.

Usage

The LET statement assigns a value to a variable, or a set of values to all members of a RECORD by using the .* notation.

The runtime system applies data type conversion rules if the data type of *expression* does not correspond to the data type of *target*.

When assigning a numeric of date/time value to a character string variable, the values are formatted for display (for example, the numeric data is right-aligned).

When specifying a comma-separated list of expressions for the right operand, the LET statement concatenates all expressions together. Unlike the || operator, if an expression in the comma-separated list evaluates to NULL, the concatenation result will not be null, except if all expressions to the right of the equal sign are null.

The target variable can be record followed by dot + star (record.*), to reference all record members of the record. In this case, the right operand must also be a record using this notation, and all members will be assigned individually.

Variables defined with a complex data type (like TEXT or BYTE) can only be assigned to NULL.

Example

```
SCHEMA stores
MAIN
  DEFINE c1, c2 RECORD LIKE customer.*
  -- Single variable assignment
  LET c1.customer_num = 123
  -- Complete RECORD assignment
  LET c1.* = c2.*
END MAIN
```

VALIDATE

The VALIDATE instructions checks a variable value according to schema validation rules.

Syntax

```
VALIDATE target [,...] LIKE
```

```
{
   table.*
|
   table.column
}
```

1. *target* is the name of the variable to be validated.
2. If *target* is a record, you can use the star notation to validate all members in the record.
3. *table.column* can be any column reference defined in the database schema.

Usage

The VALIDATE statement tests whether the value of the specified variable is within the range of values for a corresponding column in .val database schema file referenced by a SCHEMA clause. If the value does not match any value defined in the INCLUDE attribute of the corresponding column, the runtime system raises error -1321.

The argument of the VALIDATE instruction can be a simple variable, a record, or an array element. If the target is a record, you can use the dot + star notation to reference all record members in the validation, or specify a range of record members with the THRU clause.

Example

```
SCHEMA stores
MAIN
  DEFINE cname LIKE customer.cust_name
  LET cname = "aaa"
  VALIDATE cname LIKE customer.cust_name
END MAIN
```

THRU operator

The THRU keyword can be used to specify a range of members of a record.

Syntax

```
record.first-member [ THRU | THROUGH ] record.last-member
```

1. *record* defines the record to be used.
2. *first-member* defines the member of the record starting the group of variables.
3. *last-member* defines the member of the record ending the group of variables.
4. THROUGH is a synonym for THRU.

Usage

The THRU keyword can be used in several instructions such as INITIALIZE, VALIDATE, LOCATE, to specify a list of record members.

Example

```
SCHEMA stores
MAIN
  DEFINE cust LIKE customer.*
  INITIALIZE cust.cust_name THRU customer.cust_address TO NULL
END MAIN
```

Examples

Example 1: Local function variables

```
FUNCTION myfunc()
  DEFINE i INTEGER
  FOR i=1 TO 10
      DISPLAY i
  END FOR
END FUNCTION
```

Example 2: PRIVATE module variables

```
PRIVATE DEFINE s VARCHAR(100)

FUNCTION myfunc()
  DEFINE i INTEGER
  FOR i=1 TO 10
      LET s = "item #" || i
  END FOR
END FUNCTION
```

Example 3: PUBLIC module variables

This example declares public and private module variables. Public variables can be shared with other modules.

File "mydebug.4gl":

```
PUBLIC DEFINE level INTEGER,
              logfile STRING
PRIVATE DEFINE count INTEGER

FUNCTION message(m)
  DEFINE m STRING
  IF level THEN
     -- Write message to debug_logfile
     DISPLAY m
  END IF
  LET count = count + 1
END FUNCTION
```

File "mymain.4gl":

```
IMPORT FGL mydebug

MAIN
  LET mydebug.level = 4
  LET mydebug.logfile = "myfile.log"
  CALL mydebug.message("Some debug info...")
END MAIN
```

Example 4: Global variables

File "myglobs.4gl":

```
GLOBALS
  DEFINE userid CHAR(20)
  DEFINE extime DATETIME YEAR TO SECOND
END GLOBALS
```

File "mylib.4gl":

```
GLOBALS "myglobs.4gl"

DEFINE s VARCHAR(100)

FUNCTION myfunc()
  DEFINE i INTEGER
  DISPLAY "User Id = " || userid
  FOR i=1 TO 10
      LET s = "item #" || i
  END FOR
END FUNCTION
```

File "mymain.4gl":

```
GLOBALS "myglobs.4gl"

MAIN
  LET userid = fgl_getenv("LOGNAME")
  LET extime = CURRENT YEAR TO SECOND
  CALL myfunc()
END MAIN
```

Constants

The definition of constants allows to centralize common static values.

- Understanding constants on page 291
- CONSTANT on page 291
- Examples on page 293

Understanding constants

A constant defines a read-only value identified by a name. A constant is similar to a variable, except that its value cannot be modified by program code.

Constants as typically used to define common invariable values that will be used at several place in a program:

```
CONSTANT PI DECIMAL(12,10) = 3.1415926,
       MAX_SIZE INT = 10000,
       ERRMSG = "PROGRAM ERROR: %1" -- type defaults to STRING
```

A good practice is to define constants that belong to the same domain in a single .4gl module, define the constant as `PUBLIC`, and import the module where the constants are needed.

CONSTANT

The `CONSTANT` instruction defines a program constant.

Syntax

```
[PRIVATE|PUBLIC] CONSTANT constant-definition [,...]
```

where *constant-definition* is:

```
identifier [ datatype] = literal
```

1. *identifier* is the name of the constant to be defined.
2. *datatype* can be any data type except complex types like `TEXT` or `BYTE`.
3. *literal* must be an integer, decimal, string, or date/time literal, or an `MDY( )` expression.
4. *literal* cannot be `NULL`.

Usage:

Constants define final static values that can be used in other instructions.

Constants can be defined with global, module, or function scope.

By default, module constants are private; They cannot be used by an other module of the program. To make a module constant public, add the `PUBLIC` keyword before `CONSTANT`. When a module constant is declared as public, it can be referenced by another module by using the `IMPORT` instruction.

When declaring a constant, the data type specification can be omitted. The literal value automatically defines the data type:

```
CONSTANT c1 = "Drink" -- Declares a STRING constant
CONSTANT c2 = 4711    -- Declares an INTEGER constant
```

However, in some cases, you may need to specify the data type:

```
CONSTANT c1 SMALLINT = 12000 -- Would be an INTEGER by default
```

Constants can be used in variable, records, and array definitions:

```
CONSTANT n = 10
DEFINE a ARRAY[n] OF INTEGER
```

Constants can be used at any place in the language where you normally use literals:

```
CONSTANT n = 10
FOR i=1 TO n
  ...
```

Constants can be passed as function parameters, and returned from functions.

Define public constants in a module to be imported by others:

```
PUBLIC CONSTANT pi = 3.14159265
```

For date time constants, the value must be specified as an MDY(), DATETIME or INTERVAL literal:

```
CONSTANT my_date DATE = MDY(12,24,2011)
CONSTANT my_datetime DATETIME YEAR TO SECOND
             = DATETIME(2011-12-24 11:22:33) YEAR TO SECOND
CONSTANT my_interval INTERVAL HOUR(5) TO FRACTION(3)
             = INTERVAL(-54351:50:24.234) HOUR(5) TO FRACTION(3)
```

A constant cannot be used in the `ORDER BY` clause of a static `SELECT` statement, because the compiler considers identifiers after `ORDER BY` as part of the SQL statement (i.e. column names), not as constants:

```
CONSTANT pos = 3
-- Next line will produce an error at runtime
SELECT * FROM customers ORDER BY pos
```

Automatic data type conversion can take place in some cases:

```
CONSTANT c1 CHAR(10) = "123"
```

```
CONSTANT c2 CHAR(10) = "abc"
DEFINE i INTEGER
FOR i = 1 TO c1 -- Constant "123" is converted to 123 integer
   ...
FOR i = 1 TO c2 -- Constant "abc" is converted to zero!
   ...
```

Character constants defined with a string literal that is longer than the length of the data type are truncated:

```
CONSTANT s CHAR(3) = "abcdef"
DISPLAY s  -- Displays "abc"
```

The compiler throws an error when an undefined symbol is used in a constant declaration:

```
CONSTANT s CHAR(c) = "abc"
-- Compiler error: c is not defined.
```

The compiler throws an error when a variable is used in a constant declaration:

```
DEFINE c INTEGER
CONSTANT s CHAR(c) = "abc"
-- Compiler error: c is a variable, not a constant.
```

The compiler throws an error when you try to assign a value to a constant:

```
CONSTANT c INTEGER = 123
LET c = 345
-- Runtime error: c is a constant.
```

The compiler throws an error when the symbol used is not defined as an integer constant:

```
CONSTANT c CHAR(10) = "123"
DEFINE s CHAR(c)
-- Compiler error: c is a not an integer constant.
```

You typically define common special characters with constants:

```
CONSTANT c_esc  = '\x1b'
CONSTANT c_tab  = '\t'
CONSTANT c_cr   = '\r'
CONSTANT c_lf   = '\n'
CONSTANT c_crlf = '\r\n'
```

Examples

Example 1: Defining and using constants

```
CONSTANT
    c1 ="Drink",       # Declares a STRING constant
    c2 = 4711,         # Declares an INTEGER constant
    n = 10,            # Declares an INTEGER constant
    x SMALLINT=1       # Declares a SMALLINT constant

DEFINE a ARRAY[n] OF INTEGER

MAIN
  CONSTANT c1 = "Hello"
  DEFINE i INTEGER
  FOR i=1 TO n
      ...
```

```
  END FOR
  DISPLAY c1 || c2  # Displays "Hello4711"
END MAIN
```

Records

Records allow structured program variables definitions.

- DEFINE ... RECORD on page 294
- Examples on page 296

Understanding records

A record defines a structured variable, where each member can be defined with a specific data type. Records can contain other records, or arrays.

```
DEFINE person RECORD
            id INTEGER,
            name VARCHAR(100),
            birth DATE
       END RECORD
```

Records are typically used to store the values of a database row. Records can be defined according to the columns of a database table as defined in a database schema. Records are used in interactive instructions such as `INPUT` or `DIALOG` for user input, and can be used in `INSERT` and `UPDATE` SQL instructions to update the database table.

```
SCHEMA stores
DEFINE cust RECORD customer.*
-- cust is defined with the column of the customer table
```

DEFINE ... RECORD

Records define structured variables.

Syntax 1 (explicit record definition)

```
DEFINE variable RECORD
  [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
  member {
              datatype
           |
              LIKE [dbname:]tabname.colname
           }
           [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
  [,...]
END RECORD
```

Syntax 2 (database column based record)

```
DEFINE variable RECORD
    [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
    LIKE [dbname:]tabname.*
```

1. *variable* defines the name of the record.
2. *member* is an identifier for a record member variable.

3. *datatype* can be any data type, a record definition, a user defined type, an array definition, a built-in class, an imported package class, or a Java™ class.
4. *dbname* identifies a specific database schema file.
5. *tabname* identifies a database table defined in the database schema file specified by `SCHEMA`.
6. *colname* identifies a database column defined in the database schema file specified by `SCHEMA`.
7. *attribute* is an attribute to extend the record or record member definition with properties.
8. *value* is the value for the record definition attribute, it is optional for boolean attributes.

Usage

A record is an ordered set of variables (called members), where each member is defined with a specific type or in turn, structured type.

Records whose members correspond in number, order, and data type compatibility to a database table can be useful for transferring data from the database to the screen, to reports, or to functions.

In the first form (Syntax 1), record members are defined explicitly:

```
DEFINE rec RECORD
           cust_id INT,
           cust_name VARCHAR(50),
           cust_address VARCHAR(100),
           ...
       END RECORD
```

In the second form (Syntax 2), record members are created implicitly from the table definition found in the database schema file specified by the `SCHEMA` instruction:

```
SCHEMA stock
...
DEFINE rec RECORD LIKE customer.*
```

> **Important:** When using the `LIKE` clause, the data types are taken from the database schema file during compilation. Make sure that the database schema file of the development database corresponds to the production database, otherwise the records defined in the compiled version of your programs will not match the table structures of the production database. Statements like `SELECT * INTO record.* FROM table` would fail.

In the rest of the program, record members are accessed by a dot notation (`record.member`). The notation `record.member` refers to an individual member of a record. The notation `record.*` refers to the entire list of record members. The notation `record.first THRU record.last` refers to a consecutive set of members. (`THROUGH` is a synonym for `THRU`):

```
DISPLAY rec.*
```

Records can be passed as function parameters, and can be returned from functions. However, when passing records to functions, you must keep in mind that the record is expanded as if each individual member would have been passed as parameter:

```
CALL myfunction(rec.*)
```

It is possible to assign and compare records having the same structure, by using the dot star notation:

```
LET rec2.* = rec3.*
...
IF rec1.* == rec2.* THEN
   ...
END IF
```

When comparing records, all members will be compared. If two members are `NULL`, the result of this member comparison results in `TRUE`.

Records can be defined with the `ATTRIBUTES( )` clause, to specify meta-data information for the record. This feature is especially used when defining records for XML-based Web Services. For more details about XML attributes, see Attributes to customize XML serialization on page 2517.

Examples

Example 1: Defining a record with explicit member types

```
MAIN
  DEFINE rec RECORD
              id INTEGER,
              name VARCHAR(100),
              birth DATE
            END RECORD
  LET rec.id = 50
  LET rec.name = 'Scott'
  LET rec.birth = TODAY
  DISPLAY rec.*
END MAIN
```

Example 2: Defining a record with a database table structure

```
SCHEMA stores
DEFINE cust RECORD LIKE customer.*
MAIN
  DATABASE stores
  SELECT * INTO cust.* FROM customer WHERE customer_num=2
  DISPLAY cust.*
END MAIN
```

Example 3: Assigning an comparing records

```
SCHEMA stores
TYPE t_cust RECORD LIKE customer.*
MAIN
  DEFINE cust1, cust2 t_cust
  ...
  INITIALIZE cust1.* TO NULL
  ...
  LET cust2.* = cust1.*
  ...
  IF cust1.* != cust2.* THEN
     DISPLAY "Records are different!"
  END IF
  ...
END MAIN
```

Arrays

Arrays (static or dynamic) allow to handle an ordered collection of elements.

- Understanding arrays on page 297
- DEFINE ... ARRAY on page 297
- Static arrays on page 298
- Dynamic arrays on page 300

- Array methods on page 302
- Copying complete arrays on page 302
- Examples on page 302

Understanding arrays

Arrays can store a one-, two- or three-dimensional set of elements.

The language supports three kind of array types:

- Static arrays - introduced in early versions of the language.
- Dynamic arrays - to be used in new developments.
- Java™ arrays - to define an array referencing Java™ objects.

DEFINE ... ARRAY

An array defines a vector variable with a list of elements.

Syntax 1: Static array definition

```
DEFINE variable ARRAY [ size [,size  [,size] ] ]
    [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
    OF datatype
```

Syntax 2: Dynamic array definition

```
DEFINE variable DYNAMIC ARRAY
    [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
    [ WITH DIMENSION rank ]
    OF datatype
```

Syntax 3: Java™ array definition

```
DEFINE variable ARRAY [ ] OF javatype
```

1. *variable* defines the name of the array.
2. *size* can be an integer literal or an integer constant. The upper limit is **65535**.
3. *rank* an be an integer literal of 1, 2, or 3. Default is 1.
4. *datatype* can be a data type, a record definition, a user defined type, a built-in class, an imported package class, or a Java™ class.
5. *javatype* must be a Java™ class or a simple data type that has a corresponding primitive type in Java™, such as `INTEGER` (int), `FLOAT` (double).
6. *attribute* is an attribute to extend the array definition with properties.
7. *value* is the value for the array definition attribute, it is optional for boolean attributes.

Usage

The `DEFINE ... ARRAY` instruction creates a program variable as an array. The elements of the array can be of a simple type or structured records.

Consider using dynamic arrays instead of static arrays.

Java™-style arrays will only be useful to interface with Java calls.

Static and dynamic arrays can be defined with the `ATTRIBUTES( )` clause, to specify meta-data information for the variable. This feature is especially used when defining variables for XML-based Web

Services. For more details about XML attributes, see Attributes to customize XML serialization on page 2517.

Example

```
DEFINE arr DYNAMIC ARRAY OF RECORD
           p_num INTEGER,
           p_name VARCHAR(50),
           p_phone VARCHAR(20)
       END RECORD
LET arr[1].p_num = 84335
LET arr[1].p_name = "Scott McCallum"
LET arr[1].p_phone = NULL
DISPLAU arr[1].*
```

Static arrays

Defining static arrays

Static arrays can store a one-, two- or three-dimensional array of variables, all of the same type. An array member can be any type except another array (`ARRAY ... OF ARRAY`).

```
MAIN
  DEFINE custlist ARRAY[100] OF RECORD
             id INTEGER,
             name VARCHAR(50)
         END RECORD
  LET custlist[50].id = 12456
  LET custlist[50].name = "Beerlington"
END MAIN
```

Multi-dimentional static arrays

The multi-dimensional array syntax (`ARRAY[`*i*`[,`*j*`[,`*k*`]]]`) specifies static arrays defined with an explicit size for all dimensions. Static arrays have a size limit. The biggest static array size you can define is **65535**.

A single array element can be referenced by specifying its coordinates in each dimension of the array.

Avoid using large static arrays; All elements of static arrays are allocated and initialized when the program starts, even if the array is not used.

```
MAIN
  DEFINE a1 ARRAY[100] OF INTEGER
  LET a1[50] = 12456
  LET a1[5000] = 12456  -- Runtime error!
END MAIN
```

Element types

The elements of a static array variable can be of any data type except an array definition, but elements can be defined as a record containing an array member.

```
MAIN
  DEFINE arr ARRAY[50] OF RECORD
               key INTEGER,
               name CHAR(10),
               address VARCHAR(200),
```

```
               contacts ARRAY[50] OF VARCHAR(20)
       END RECORD
  LET arr[1].key = 12456
  LET arr[1].name = "Scott"
  LET arr[1].contacts[1] = "Bryan COX"
  LET arr[1].contacts[2] = "Mike FLOWER"
END MAIN
```

Passing static arrays to functions

Static arrays are passed by value to functions. This is not recommended, as all array members will be copied on the stack.

A static array cannot be returned from a function.

Consider using dynamic arrays if you need to pass/return a list of elements to/from functions.

Using array methods

Array methods can be used on static arrays; However these methods are designed for dynamic arrays and are not appropriate for static arrays.

Controlling out of bound in static arrays

Controlling of out of bounds index error

By default, when an array index is out of range, fglrun raises error -1326. This is only the case for static arrays: When using a dynamic array, new elements are allocated if the index is greater than the actual array size.

Raising an index out of bounds error is natural for static arrays. However, in some situations, code must execute without error and evaluate expressions using indexes that are greater than the size of the array, especially with boolean expressions in `IF` statements:

```
IF index <= max_index OR arr[index] == some_value THEN
   ...
END IF
```

In this example, as all parts of a boolean expression needs to be evaluated, the runtime system must get the value of the *arr[index]* element.

You can use an FGLPROFILE entry to control the behavior of the runtime system when an array index is out of bounds for a static array:

```
fglrun.arrayIgnoreRangeError = true
```

When this FGLPROFILE entry is set to true, the runtime system will return the first element of the array if the index is <=0 or greater than the size of the array and continue with the normal program flow.

Unless existing code is relying on this behavior, it is better to let the default get array out of bounds errors when the index is invalid.

You may also want to use the compiler directive to control boolean expression evaluation, with the `OPTIONS SHORT CIRCUIT` instruction.

Dynamic arrays

Defining dynamic arrays

Dynamic arrays are defined with the `DYNAMIC ARRAY` syntax and specify an array with a variable size. Dynamic arrays have no theoretical size limit. The elements of dynamic arrays are allocated automatically by the runtime system, according to the indexes used.

```
MAIN
  DEFINE a1 DYNAMIC ARRAY OF INTEGER
  LET a2[5000] = 12456  -- Automatic allocation for element 5000
END MAIN
```

Element types

The elements of a dynamic array variable can be of any data type except an array definition, but elements can be defined as a record containing an array member.

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
               key INTEGER,
               name VARCHAR(30),
               address VARCHAR(200),
               contacts ARRAY[50] OF VARCHAR(20)
       END RECORD
  LET arr[1].key = 12456
  LET arr[1].name = "Scott"
  LET arr[1].contacts[1] = "Bryan COX"
  LET arr[1].contacts[2] = "Mike FLOWER"
END MAIN
```

Automatic element allocation

When a dynamic array element does not exist, it is automatically allocated before it is used. For example, when you assign an array element with the `LET` instruction by specifying an array index greater as the current length of the array, the new element is created automatically before assigning the value. This is also true when using a dynamic array in a`FOREACH` loop or when dynamic array elements are used as r-values, for example in a DISPLAY.

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[50] = 33 -- Extends array size to 50 and assigns 33 to element #50
  DISPLAY a[100] -- Extends array size to 100 and displays NULL
END MAIN
```

Important:

Pay attention to automatic element allocation in dynamic arrays. The following code example creates an additional element because at each iteration, the runtime system must allocate a new element to fetch the row from the database. As result, you need to remove the last element of the array after the `FOREACH` loop:

```
DEFINE arr DYNAMIC ARRAY OF RECORD
               key INTEGER,
               name VARCHAR(30)
       END RECORD,
       x INTEGER
DECLARE c1 CURSOR FOR SELECT ckey, cname FROM mytable
LET x=1
```

```
FOREACH c1 INTO arr[x].*
   LET x=x+1
END FOREACH
CALL arr.deleteElement(x)

-- A more elegant way to fetch rows into an array:
TYPE my_type RECORD LIKE mytable.*
DEFINE arr DYNAMIC ARRAY OF my_type,
       rec my_type,
       x INTEGER
DECLARE c1 CURSOR FOR SELECT * FROM mytable
LET x=1
FOREACH c1 INTO rec.*
   LET arr[x:=x+1].* = rec.*
END FOREACH
```

Passing and returning dynamic arrays to functions

Dynamic arrays are passed (or returned) by reference to/from functions.

The dynamic array can be modified inside the called function, and the caller will see the modifications.

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  CALL fill(a)
  DISPLAY a.getLength() -- shows 2
END MAIN

FUNCTION fill(x)
  DEFINE x DYNAMIC ARRAY OF INTEGER
  CALL x.appendElement()
  CALL x.appendElement()
END FUNCTION
```

Dynamic array size

The `getLength()` array method method returns the number of allocated elements:

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[5000] = 12456
  DISPLAY a.getLength()
END MAIN
```

Dynamic array methods

A set of methods is available to manipulate dynamic arrays. For a complete list, see DYNAMIC ARRAY methods on page 1697.

Using multi-dimentional dynamic arrays

Multi-dimentional dynamic arrays can be defined by using the `WITH DIMENSION` syntax.

Array methods can be used on multi-dimensional arrays with the brackets notation:

```
MAIN
  DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
  DEFINE a3 DYNAMIC ARRAY WITH DIMENSION 3 OF INTEGER
  LET a2[50,100]  = 12456
  LET a2[51,1000] = 12456
  DISPLAY a2.getLength()         -- shows 51
```

```
  DISPLAY a2[50].getLength()     -- shows 100
  DISPLAY a2[51].getLength()     -- shows 1000
  LET a3[50,100,100]  = 12456
  LET a3[51,101,1000] = 12456
  DISPLAY a3.getLength()         -- shows 51
  DISPLAY a3[50].getLength()     -- shows 100
  DISPLAY a3[51].getLength()     -- shows 101
  DISPLAY a3[50,100].getLength() -- shows 100
  DISPLAY a3[51,101].getLength() -- shows 1000
  CALL a3[50].insertElement(10)  -- inserts at 50,10
  CALL a3[50,10].insertElement(1)-- inserts at 50,10,1
END MAIN
```

Array methods

Native BDL arrays and Java arrays can be used to invoke built-in methods.

For the list of native array methods, see DYNAMIC ARRAY methods on page 1697.

For the list of Java array methods, see Java Array type methods on page 1701.

Copying complete arrays

The compiler allows the `.*` notation to assign an array to another array with the same structure. Static array elements are copied by value (except objects and LOB members), while elements of dynamic arrays are copied by reference, even for simple data types. This means that after assigning a dynamic array with the `.*` notation, if you modify an element in one of the arrays, the change will be visible in the other array. You must pay attention to this behavior if you are used to the `.*` notation for simple records.

Note: When assigning a dynamic array with the .* notation, all elements are copied by reference:

```
MAIN
  DEFINE a1, a2 DYNAMIC ARRAY OF RECORD
                key INTEGER
              END RECORD
  LET a1[1].key = 123
  LET a2.* = a1.*
  DISPLAY a2[1].key    -- shows 123
  LET a2[1].key = 456
  DISPLAY a1[1].key    -- shows 456
END MAIN
```

Examples

Example 1: Using static and dynamic arrays

```
MAIN
  DEFINE a1 DYNAMIC ARRAY OF INTEGER
  DEFINE a2 DYNAMIC ARRAY WITH DIMENSION 2 OF INTEGER
  DEFINE a3 ARRAY[10,20] OF RECORD
              id INTEGER,
              name VARCHAR(100),
              birth DATE
            END RECORD
  LET a1[5000] = 12456
  LET a2[5000,300] = 12456
  LET a3[5,1].id = a1[50]
  LET a3[5,1].name = 'Scott'
  LET a3[5,1].birth = TODAY
END MAIN
```

Example 2: Fetching database rows into a dynamic array

Automatic allocation of dynamic array element in the FOREACH statement creates an additional element that needs to be deleted after the loop:

```
SCHEMA stores

MAIN
  DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
  DEFINE index INTEGER

  DATABASE stores

  DECLARE curs CURSOR FOR SELECT * FROM customer
  LET index = 1
  FOREACH curs INTO custarr[index].*
    LET index = index+1
  END FOREACH
  CALL custarr.deleteElement(custarr.getLength())

  DISPLAY "Number of rows found: ", custarr.getLength()
  FOR index=1 TO custarr.getLength()
    DISPLAY custarr[index].*
  END FOR

END MAIN
```

Types

Types can be defined by the programmer to centralize the definition of complex/structured variables.

- Understanding type definition on page 303
- TYPE on page 303
- Using types in programs on page 304
- Examples on page 305

Understanding type definition

The TYPE instruction declares a user defined type, which is based on native data types, records or arrays. Once declared, a type can be referenced in the declaration of program variables, or in other types. Types are typically defined to avoid the repetition of complex structured types. Consider user PUBLIC TYPE definitions to share types accross modules, with IMPORT FGL.

TYPE

Types define a synonym for a base or structured data type.

Syntax:

```
[PUBLIC|PRIVATE] TYPE type-definition [,...]
```

where *type-definition* is:

```
identifier {
   {
     datatype
   |
     LIKE [dbname:]tabname.colname
   }
```

```
   [ ATTRIBUTES( attribute [ = "value" ] [,...] ) ]
```

1. *identifier* is the name of the type to be defined.
2. *datatype* is any data type, record structure, or array definition supported by the language.
3. *attribute* is an attribute to extend the type definition with properties.
4. *value* is the value for the type attribute, it is optional for boolean attributes.

Usage

User-defined types enforce reusability and simplify programming, by centralizing data structure definitions at a single place.

When defining types with the `LIKE` clause, the data types are taken from the database schema file at compile time. Make sure that the schema file of the database schema during development corresponds to the database schema of the production database; otherwise the types defined in the compiled version of your modules will not match the table structures of the production database.

Types can be defined with the `ATTRIBUTES( )` clause, to specify meta-data information for the type. This feature is especially used when defining types for XML-based Web Services. For more details about XML attributes, see Attributes to customize XML serialization on page 2517.

Using types in programs

Define a type as a synonym for an existing data type, or as a shortcut for records and array structures.

After declaring a type, it can be used as a normal data type to define variables.

```
TYPE t_customer RECORD
         cust_num INTEGER,
         cust_name VARCHAR(50),
         cust_addr VARCHAR(200)
    END RECORD
...
     DEFINE c1 t_customer
...
     DEFINE o1 RECORD
                order_num INTEGER,
                customer t_customer,
                ...
            END RECORD
...
     DEFINE custlist DYNAMIC ARRAY OF t_customer
```

The scope of a type is the same as for variables and constants. Types can be global, module-specific, or local to a function.

A good practice is to define types that belong to the same domain in a single .4gl module, and import that module in the modules where the types are needed.

By default, module-specific types are private; They cannot be used by an other module of the program. To make a module type public, add the `PUBLIC` keyword before `TYPE`. When a module type is declared as public, it can be referenced by another module by using the `IMPORT FGL` instruction:

```
-- customers.4gl
PUBLIC TYPE t_ord RECORD
           ord_id INTEGER,
           ord_date DATE,
           ord_total DECIMAL(10,2)
       END RECORD
PUBLIC TYPE t_cust RECORD
           cust_id INTEGER,
           cust_name VARCHAR(50),
```

```
           orders DYNAMIC ARRAY OF t_ord,
           ...
       END RECORD
...

-- main.4gl
IMPORT FGL customers
MAIN
    DEFINE custlist DYNAMIC ARRAY OF t_cust
    ...
END MAIN
```

Examples

Example 1: Defining a type with a record structure

```
TYPE t_customer RECORD
            cust_num INTEGER,
            cust_name VARCHAR(50),
            cust_addr VARCHAR(200)
    END RECORD

MAIN
    DEFINE custrec t_customer
    DEFINE custarr DYNAMIC ARRAY OF t_customer
    DEFINE index INTEGER

    LET custrec.cust_num = 123
    ...

    LET custarr[index].* = custrec.*
    ...

END MAIN
```

Example 2: Defining a type an using it in another module

The following example defines a type in first module, and then uses the type in a report program:

type_order.4gl:

```
PUBLIC TYPE rpt_order RECORD
                order_num INTEGER,
                store_num INTEGER,
                order_date DATE,
                cust_num INTEGER,
                fac_code CHAR(3)
             END RECORD
```

report.4gl:

```
IMPORT FGL type_order

MAIN
  DEFINE o type_order.rpt_order

  CONNECT TO "custdemo"

  DECLARE order_c CURSOR FOR
     SELECT orders.*
        FROM orders ORDER BY cust_num
  START REPORT order_list
```

```
  FOREACH order_c INTO o.*
    OUTPUT TO REPORT order_list(o.*)
  END FOREACH
  FINISH REPORT order_list

END MAIN

REPORT order_list(ro)
  DEFINE ro rpt_order
  FORMAT
    ON EVERY ROW
     PRINT ro.order_num, ...
  ...
```

Advanced features

These topics cover advanced features of the Genero Business Development Language

- Localization on page 307
- Runtime stack on page 336
- Exceptions on page 340
- OOP support on page 349
- XML support on page 351
- Globals on page 353
- Database schema on page 355
- Programs on page 368
- Program execution on page 390
- Deploying mobile apps on page 2572
- Front calls on page 395

Localization

Localization support allows you to implement programs that follow specific language and cultural rules.

Programs execute in a specific *application locale*. Beside the support of a locale specification which defines the character set used by programs, the internationalization of an application requires all strings in the sources that are subject to translation to be extracted and centralized. Localized strings are used to keep application messages and form labels in external resource files, which can be provided in different languages.

- Application locale on page 307
- Localized strings on page 327

Application locale

The *application locale* defines the language (for messages), country or territory (for currency symbols and date formats) and code set (for character set encoding). A program needs to be able to determine its locale and act accordingly, to be portable across different languages and character sets.

This section describes how to define the locale for your programs.

> **Important:** The same code point can represent different characters in different character sets. An invalid locale configuration in one of the components can result in invalid characters in the database. For example, a client application is configured to display glyphs (font) for CP437. If the application gets a 0xA2 (decimal 162) code point, it displays an o-acute character. Now imagine that the DB client is configured with character set CP1252. In this character set, the code point 0xA2 is actually the cent currency sign. As a result, if you insert the o-acute char (0xA2 in CP437) in the database, it will actually be seen as cent sign (0xA2 in CP1252) by the database server. When fetching that character back to the client, the database server returns the 0xA2 code point, which displays correctly as o-acute on the CP437 configured client, and the end user sees what was entered before. But with a different application configured properly with CP1252 and DB client codeset, the end user will see the cent currency sign instead of the o-acute character.

Quickstart guide for locale settings

This is a quick step-by-step guide to properly configure locale settings for your Genero application.

Setting the locale involves different components, which all must be properly configured.

> **Tip:** This is a quickstart guide for locale settings. It is highly recommended that you read the complete set of articles regarding localization.

1. The *application locale* is defined by the character set used in your source files (`.4gl`, `.per`, `.str`). The same character set will be used in the compiled files (`.42m`, `.42f`, `.42s`).
2. Set the operating system locale corresponding to the application locale.
   - On UNIX™ based systems (including Mac OS-X™), define the LANG (or LC_ALL) environment variable. Use `locale -a` command to check if the locale exists on the machine. If not, it must be installed. If not set, LANG defaults to POSIX (ASCII).
   - On Windows™ platforms, check if the regional settings for non-UNICODE applications match the application locale. If the regional settings do no match, you can define the LANG environment variable with a locale name supported by Microsoft™ C Runtime Library, such as `French_France.1252`, or set `LANG=.fglutf8` for the UTF-8 character set.
   - On iOS mobile devices, the application locale is always UTF-8, it cannot be changed.
   - On Android™ mobile devices, the application locale is always UTF-8, it cannot be changed.
3. When using UTF-8 as character encoding, define the **length semantics** with the FGL_LENGTH_SEMANTICS={BYTE|CHAR} environment variable. On server platforms, Genero is using Byte Length Semantics by default for compatibility reasons. It's highly recommended to set FGL_LENGTH_SEMANTICS=CHAR to use Character Length Semantics. On mobile platforms, character length semantics is the default (i.e. FGL_LENGTH_SEMANTICS does not need to be defined when running on a mobile device, it defaults to CHAR, and cannot be set to BYTE).
4. Set the **database client locale** with a character set corresponding to the application locale. For example, with Informix®, this is defined with the CLIENT_LOCALE environment variable. The name of the database client locale is certainly different from the application locale. But remember the application and database client character sets must match. The database server locale might be different from the db client locale.
5. Check the length semantics used by the database. For example, with Oracle, you might want to set the database option `NLS_LENGTH_SEMANTICS='CHAR'`, if the application uses CLS (typically with UTF-8).
6. With UTF-8, use the proper SQL character data type to store UTF-8 data: This data type might be different according to the type of database server. For more details, see SQL character type for Unicode/UTF-8 on page 418.
7. Set the **front-end locale and font**. By front-end, we mean the program the end user interacts with. This can be a Genero front-end or a terminal emulator like Gnome-term, Putty, or a Windows™ Console. When using a Genero front-end, the front-end character set is fixed by the type of the front end and conversion from/to application character set is automatic, but you may need to select a font different from the system default. If you want to execute a TUI application in a terminal emulator, you must be sure that the terminal is configured to display the correct character set. This is for example defined with the *chcp* command on Windows™, or in the "*Set Character Encoding*" menu option of a Gnome-term.
8. Define the date, numeric and monetary formats with the DBDATE, DBMONEY, DBFORMAT environment variables. On server platforms such as Unix and Windows, these default to US formats (month/day/year for dates, the dot as decimal separator and $ as currency symbol). On mobile platforms, these default to the regional settings defines on the device.

Locale and character set basics

Before starting with application/database design, configuration and settings, you must know some basics concerning language and character sets on computers. In this section, we attempt to describe these basics, but we strongly recommend you to carefully read the operating system and database server manuals covering localization or character set handling. You can also find a lot of information about character sets and character encoding on the internet.

Why do I need to care about the locale and character set?

If you don't know what you are doing with character sets, the end user might get strange characters displayed on the screen, and will probably not be able to input non-ASCII characters. In the worst case, as character set conversion can be symmetric for single-byte character sets, the end user might see correct characters on the workstation, but on the back-end you can get invalid characters in the database files. By upgrading to a newer OS, Genero Business Development Language runtime or database system, or

if a character set mapping utility was used somewhere in the chain, you can even get mixed character encoding in the database files.

Characters, code points, character sets, glyphs and fonts

In computers, a character is the unit of information corresponding to a symbol of a natural language. This can be a letter, a digit, a punctuation mark, a mathematic or even musical symbol. To represent a character in memory or in a file, computers must encode the character in a specific numeric value called code point. This code point uniquely identifies a character in a given character set. Mapping a character to a code point is called character encoding. The same code point might represent a different character in several character sets. The glyph is the graphical representation of the character. In other words, it's the way the character is drawn on the screen or on a printer. Computers implement the glyph of characters with fonts, by mapping a code point to a bitmap image or drawing instructions based on math formulas or vector graphics.

The ASCII character set

ASCII stands for the American Standard Code for Information Interchange. ASCII is a well-known character encoding based on the English alphabet. Characters are encoded in a single byte, using the 7 lower bits only. Up to 127 characters, printable and not printable (like control characters), are defined in ASCII. Nearly all other character sets (using 8 bits or multiple bytes) define the first 127 characters as the ASCII character set. Aliases for ASCII include ISO646-US, ANSI_X3.4-1968, IBM367, cp367, and more.

Single-byte character sets (SBCS)

A single-byte character set defines the encoding for characters on a unique byte. The size of a character is always one byte.

Example of single-byte character sets include ISO-8859-1, MS code page CP1252.

Genero Business Development Language supports single-byte character sets.

Double-byte character sets (DBCS)

A double-byte character set defines the encoding for characters on two bytes. The size of a character is always two bytes.

Example of double-byte character sets include UCS-2, used by SQL Server in NCHAR and NVARCHAR columns. Note that UTF-16 is not a (fixed) double-byte character set: You can have characters encoded on 2 or 4 bytes. UCS-2 is actually a subset of UTF-16.

Note that Genero Business Development Language does not support double-byte character sets.

Multibyte character sets (MBCS)

A multibyte character set defines the encoding for characters on a variable number of bytes. The size of a character can be one (usually ASCII chars), two, three or more bytes, depending on the character set.

Example of multibyte character sets are BIG5, EUC-JP, and UTF-8. BIG5 and EUC-JP characters can be one or two bytes long, while UTF-8 characters can be 1, 2, 3 or 4 bytes long (usually a maximum of 3 is sufficient).

Genero Business Development Language supports multibyte character sets.

Character size unit and length semantics

When programming an application for a Latin-based language sush as English, a single-byte character set can be used, and the logical size, storage size and print width of characters is the same. For example, in ISO-8859-1, the ê character takes one logical position, has a storage size of one byte and a print width of one.

When programming an international application using multiple languages and a multibyte character set encoding, you must distinguish three size units:

1. The size in **character unit**, to count or position logical characters used in a string. For example, the strings `abc` and `åôë` have both a length of 3, in character units.

2. The size in **byte unit**, used to encode the character in a given character set. For example, a Latin ê acute character will use a unique byte in the ISO-8859-1 character set, but needs two bytes in UTF-8.
3. The size in **width unit**, used in formatting and alignments. The width is the length of the glyph/font of characters, especially in a fixed font. For example, a latin character will take one width unit, while an asian ideogram will take 2 width units.

Working with byte units in a multibyte character set can be difficult: You need to calculate sizes, lengths and substring offsets in a number of bytes, when the natural way is to count in characters.

Length semantics define the unit to be used for character data type definition, charcater string lengths and positions.

With Byte Length Semantics, a length is expressed in bytes, while Character Length Semantics counts in characters.

The UNICODE Standard

UNICODE is a standard specification to map all possible characters to a numeric value, in order to cover all possible languages in a unique character set. UNICODE defines the mapping of characters to integer codes, but it does not define the exact implementation (i.e. encoding) for a character. Several character sets are based on the UNICODE standard, such as UTF-7, UTF-8, UTF-16, UTF-32, UCS-2, and UCS-4. Each of these character sets use a different encoding method. For example, with UTF-8, the letter **Æ** is encoded with two bytes as 0xC3 and 0xB6, while the same character will be encoded 0x00C6 with UTF-16.

When Microsoft™ Windows™ users talk about UNICODE, they typically mean UCS-2 or UTF-16, while UNIX™ users typically mean UTF-8.

When do I need a UNICODE character set?

With internationalization, people want to use different languages within the same application; for example, to have Chinese, Japanese, English, French and German addresses of customers in their database. UNICODE is a character encoding specification that defines characters for all languages. More and more databases will use a UNICODE character set on the database server, because it "standardizes" all data from different client applications. If needed, the client application can then use a different character set like ISO-8859-1 or BIG5: The database software takes care of character set conversions. However, if the end user needs to deal with different languages, all components of the system (from database backend to GUI front-end) must work in UNICODE.

The UNICODE character set supported by Genero Business Development Language is UTF-8. Double-byte based UNICODE character sets such as UCS-2 or UTF-16 are not supported. The database server can however store character data in another UNICODE character set, as long as the database client is able to handle to convertion to/from UTF-8 for the Genero runtime system.

What is the standard?

At this time, UNICODE tends to be the standard, but unfortunately not all platforms/systems use the same UNICODE character set. Recent UNIX™ distributions define UTF-8 as the default character set locale, XML files are UTF-8 by default, while Microsoft™ Windows™ standard is UTF-16 (NTFS) / UCS-2 (SQL Server).

What is my current character set?

On a UNIX™ box, you have the LANG / LC_ALL environment variables to define the locale. Each process / terminal can set its own locale. By default this is en_US.utf8 on recent UNIX™ systems. You can query for available locales with the *locale -a* command. Some systems come with only a few locales installed, you must then install an additional package to get more languages. You must also define the correct character set in the terminal (xterm or gnome-term), otherwise non-ASCII characters will not display properly.

On Windows™ platforms, for non-UNICODE (i.e. non-UTF-16/UCS-2) applications, you have ACP and OEMCP code pages. ACP stands for ANSI Code Page and were designed by Microsoft™ for first GUI applications, while OEMCP defines old code pages for MS/DOS console applications. You can select the default ACP/OEMCP code pages for non-UNICODE application in the language and regional settings

panel of Windows™ (make sure you define the settings for non-UNICODE applications, this is done in the "Advanced" panel on Windows™ XP). Code page can be changed in each console window with the *chcp* command. With Genero Business Development Language, you can use the LANG environment variable on Windows™ to define the character set for BDL. However, it is strongly recommended to use the default Windows™ system locale and avoid to set LANG on Windows™.

Understanding locale settings

It is critical to understand how the different components of a program handle locale settings. Each component (i.e. runtime system, database client software, front-end, terminal) has to be configured properly to get the correct character set conversions through the whole chain. The chain starts on the end-user workstation with front-end windows and ends in the database storage files.

Figure 20: The Locale Settings schema on page 312 shows the different components of a Genero Business Development Language process.

- The red rectangles show where character set conversion occurs. Conversion can happen in the front-end side, for C-based Web Services extension and in the database client. No conversion is done by the fglrun runtime system.
- In the runtime system (fglrun), the locale and code set support is based on the POSIX C runtime libraries driven by the **setlocale()** standard function. This locale setting is defined by the LC_ALL (or LANG) environment variable. The locale of the runtime system must match the code set of the deployed program modules (42m and 42f files).
- The terminal (for TUI applications) and the C runtime library are represented in magenta rectangles. These elements will use the locale of the runtime system.
- The locale of the database client must match the locale of the runtime system. Each database vendor uses it's own locale configuration system.
- The database server uses it's own locale settings, which can be different from the runtime system / db client locale. You can for example store the data in UTF-8 but have programs using ISO-8859-1.

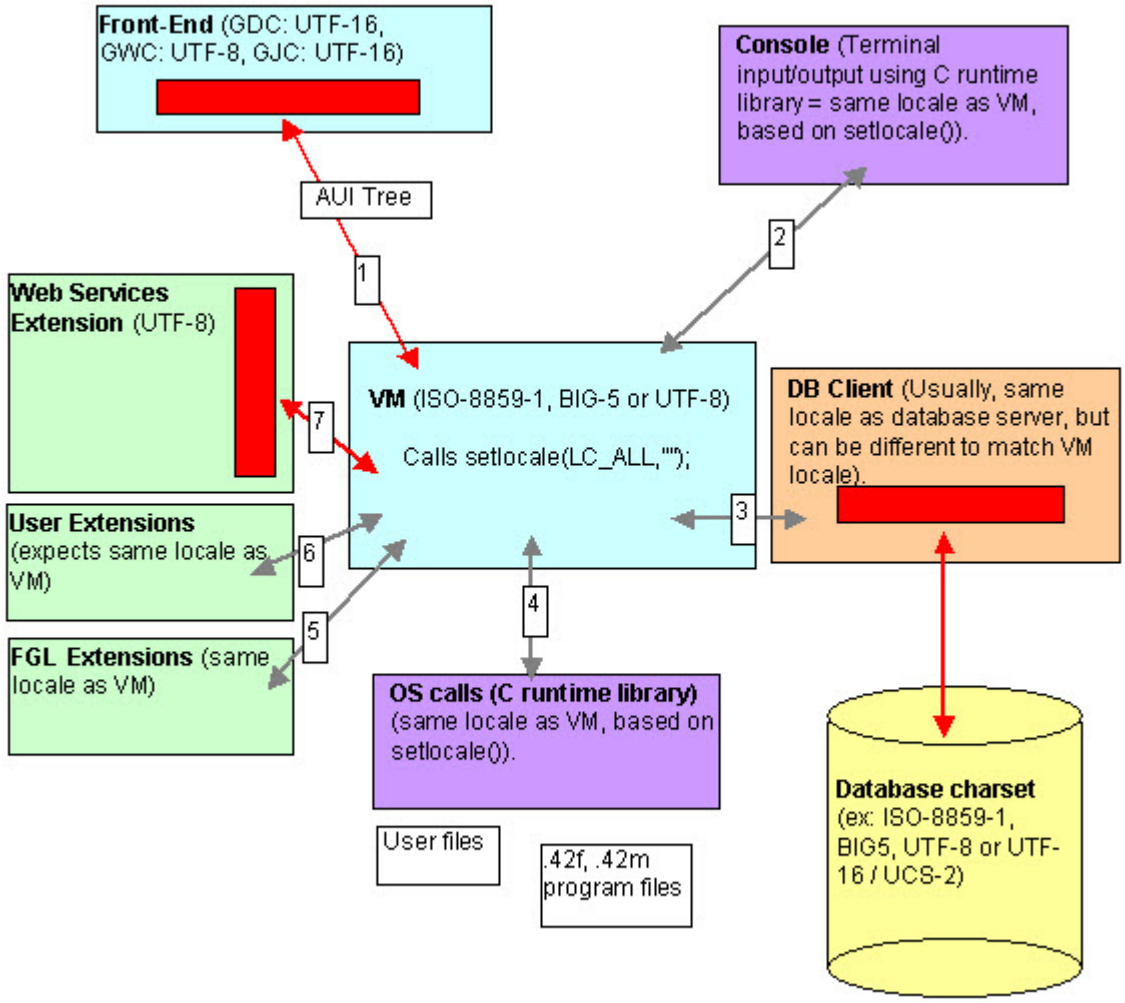


Figure 20: The Locale Settings schema

The typical mistake is to forget to set the runtime system locale (LANG/LC_ALL), or the database client software locale. Systems cannot detect that the current locale is appropriate and don't raise any error, except when a set of bytes does not represent a valid code point in the current codeset. A character string is just a set of bytes; The same code might represent different characters in different code sets. For example, the Latin letter é with acute (UNICODE: U+00E9) will be encoded as 0xE9/233 in CP1252 but will get the code 0x82/130 in CP437. The codes 233 or 130 are valid characters in both code sets, so if the database uses CP1252, 233 will represent an é and 130 will represent a curved quote. If the client application used CP437, the é will be encoded as 130, stored as curved quotes but are retrieved from the database as is and displayed back as é in the CP437 code page. From the front-end side, you can't see that the character in the database in wrong.

Pay attention that on recent UNIX™ systems, the default locale is set to UTF-8. If your application has been developed on an older system, it is probably using a single-byte character set like ISO-8859-1 or CP1252, and program need to be executed in this locale, not in the UTF-8 locale.

It is also important to identify database server character set (i.e. in what code set the characters are stored in the database). Usually the database character set is defined when creating a database entity.

The best way to test if the characters inserted in the database are correct is to use the database vendor SQL interpreter and select rows inserted from a BDL program. The rows most hold non-ASCII data to check if the code of the characters is correct. Some databases support the ASCII() or better, the UNICODE() SQL function to check the code of a character. Use such function to determine the value of a character in the database field. If the character code does not correspond to the expected value in the character set of the database server, there is a configuration mistake somewhere.

If you run a BDL application in TUI mode (or a batch program doing DISPLAYs), you must properly configure the code set in the terminal window (X11 xterm, Windows™ CMD, putty, etc). If the terminal code set does not match the runtime system locale, you will get invalid characters displayed on the screen. On Windows™ platforms, the OEM code page of the CMD window can be queried/changed with the chcp command. On a Gnome terminal, go to the menu "Terminal" - "Set Character Encoding".

Defining the application locale

This section describes the settings defining the application locale, changing the behavior of the compilers and runtime system.

Language and character set settings

Purpose of application locale definition

The locale settings matters at compile time and at runtime. At runtime, the locale changes the behavior of the character handling functions, such as `UPSHIFT` and `DOWNSHIFT`. It also changes the handling of the character strings, which can be single byte or multibyte encoded. Compilation errors will occur if the source files contain characters that do not exist in the encoding defined by the current locale.

Always check that the local environment variable matches the locale of your Genero application, during development and at runtime:

```
$ fglrun -i mbcs
Charmap      : UTF-8
Multibyte    : yes
Stateless    : yes
Length Semantics : CHAR
```

Mobile plaforms

On iOS and Android™ mobile platforms, the locale is automatically defined to be UTF-8. This cannot be changed.

The language conventions and system messages are defined by the device settings.

Windows™ plaforms

On Windows™ platforms, if you don't specify the LANG environment variable, the language and character set defaults to the system locale which is defined by the regional settings for non-Unicode applications. For example, on a US-English Windows™, this defaults to the 1252 code page. You typically leave the default on Windows™ platforms (i.e. you should not set the LANG variable, except if your application uses a different character set as the Windows™ system locale).

On Windows™ platforms, the syntax of the LANG variable is:

```
  language[_territory[.codeset]]
| .codeset
```

For example:

```
C:\ set LANG=English_USA.1252
```

UNIX™ plaforms

On UNIX™-based platforms, The **LC_ALL** (or **LANG**) environment variable defines the global settings for the language used by the application.

With the **LANG** environment variable (or **LC_ALL**, on UNIX™), you define the *language*, the *territory* (aka country) and the *codeset* (aka character set or code page) to be used. The format of the value is normalized as follows, but may be specific on some operating systems:

```
language_territory.codeset
```

For example:

```
$ LC_ALL=en_US.iso88591; export LC_ALL
```

What are possible locales on my platform?

Usually OS vendors define a specific set of values for the *language*, *territory* and *codeset*. For example, on a UNIX™ platform, you typically have the value "en_US.ISO8859-1" for a US English locale, while Microsoft™ Windows™ requires the "English_USA.1252" value. For more details about supported locales, refer to the operating system documentation.

A list of available locales can be found on UNIX™ platform by running the *locale -a* command. You may also want to read the man pages of the *locale* command and the *setlocale* function. On Windows™ platforms, search the Microsoft™ MSDN documentation for "Language and Country/Region Strings".

UNICODE support (UTF-8)

To support multiple languages in your application, you must use UNICODE. The encoding supported by Genero for UNICODE applications is UTF-8.

On UNIX™ platforms, UTF-8 locales are natively supported with LANG/LC_ALL.

On Windows™ platforms, UTF-8 is not well supported by the operating system: Defining the LANG environment variable to code page 65001 will not work. To workaround this limitation, Genero implements UTF-8 support on Windows™ by setting the LANG environment variable to the value `.fglutf8` :

```
C:\ set LANG=.fglutf8
```

Length semantics settings

Understanding length semantics

The length semantics of character string data matters when using a multibyte character set. Length semantics involves data type length specification for database column and program variable definitions, as well as string manipulations (for string lengths, character positions, offsets and substring ranges).

In a single-byte characters set like ISO-8859-1, a character is encoded on one byte. The length of a string can be counted in bytes or characters, the unit does not matter. In other words, the length semantics is identical in bytes or characters, with a single byte encoding. However, with a multibyte character set like UTF-8 or BIG5, a character can be encoded on several bytes. In such case, the unit regarding length semantics matters, because the number of bytes of a character string can be different from the number of characters.

For multibyte characters sets, the language supports Byte Length Semantics (**BLS**) and Character Length Semantics (**CLS**) specification. BLS or CLS usage depends on the current character set of the application. BLS is typically used with a character set such as BIG5, because for historical reasons programmers are used to count 2 bytes for each Asian ideogram. For UTF-8, which is a variable size encoding, CLS should be used instead. CLS simplifies data type definition and string handling when using UTF-8.

Programming areas concerned by length semantics are illustrated in the following code example:

```
SCHEMA shop

# CREATE TABLE mytable (
#     k INT,
#     vc VARCHAR(10)
#         -- what is the unit for the column size and how many
#         -- characters can be stored in this column?
#  )

MAIN
  DEFINE buf, tmp VARCHAR(50)  -- what is the unit for the size?
  DEFINE rec RECORD LIKE mytable  -- what is the size of vc member?
  DEFINE str STRING, len INT

  DATABASE shop

  SELECT LENGTH(vc) INTO len -- What unit use string functions in SQL?
     FROM mytable WHERE k = 45

  LET buf = "abcdef..."  -- How many chars can this variable hold?

  DISPLAY length(buf)  -- In what unit is the length expressed?

  LET tmp = buf[1,5]  -- What is the unit for char positions?

  LET str = buf
  DISPLAY str.getLength() -- What is the unit for the length?
  DISPLAY str.getIndexOf("def") -- What is the unit for the offset?

END MAIN
```

Using Byte Length Semantics

Byte Length Semantics must be used if the current locale defines a multibyte character set different from UTF-8.

Important:

- Byte Length Semantics is the default on UNIX™ and Windows™ platforms.
- Byte Length Semantics cannot be set on mobile platforms.

With BLS, the size of CHAR/VARCHAR program variables is expressed in byte units. In a single-byte character set like ISO-8859-1, every character is encoded on a unique byte, so the number of bytes equals the number of characters. When using BLS with a multibyte character set, you must be aware of the storage size in byte units: Character encoding requires more than one byte, so the number of bytes to store a multibyte string is bigger than the number of characters. For example, in a BIG5 encoding, one Chinese character needs 2 bytes, so if you want to hold a BIG5 string with a maximum of 10 Chinese characters, you must define a CHAR(20). When using UTF-8, characters can take one or several bytes which can use two or three times more storage space as character count. You need to choose the right expansion factor to define CHAR or VARCHAR variables in byte units.

```
-- Using Byte Length Semantics
DEFINE var VARCHAR(10)  -- Can store 10 bytes / 10 single-byte chars.
```

In order to use BLS, you can define the FGL_LENGTH_SEMANTICS environment variable to "BYTE", or just leave it unset, if BLS is the default on your platform. For example, on UNIX™:

```
$ FGL_LENGTH_SEMANTICS="BYTE"
$ export FGL_LENGTH_SEMANTICS
```

Using Char Length Semantics

Character Length Semantics should be used with multibyte character sets such as UTF-8: Migrating to UTF-8 by using CLS will allow you to leave the source code untouched, even when doing complex string/substring manipulations.

The database should typically also use UTF-8 and CLS. If the database uses UTF-8 and only supports BLS, the programs can still use CLS with UTF-8.

> **Important:** Char Length Semantics is the default on iOS and Android™ mobile platforms, and cannot be changed (Byte Length Semantics cannot be used on mobile: only UTF-8 character set is allowed).

With CLS, the size of a CHAR/VARCHAR program variable is expressed in character units, and the number of bytes needed to store these characters is allocated automatically. A VARCHAR(10) variable will hold 10 characters, of any byte length. Further, language functions and class methods dealing with character string length and positions will use character units.

```
-- Using Character Length Semantics
DEFINE var VARCHAR(10)  -- Can store 10 chars in UTF-8, or any encoding.
LET var = "Forêt"  -- 5 chars, that take 6 bytes in UTF-8
DISPLAY length(var)  -- Displays a length of 5 (characters)
DISPLAY "[",var[4,5],"]"    -- Displays [êt]
```

To enable Char Length Semantics, define the FGL_LENGTH_SEMANTICS environment variable to "CHAR". For example, on UNIX™:

```
$ FGL_LENGTH_SEMANTICS="CHAR"
$ export FGL_LENGTH_SEMANTICS
```

Length Semantics in SQL

On the database server side, the length semantics used for character data types varies from a vendor to another. Some databases use BLS, other use CLS, and other support both semantics. For example, Informix® uses BLS only (with a special server configuration parameter `SQL_LOGICAL_CHAR` to define a size conversion ratio). Oracle supports both CLS and BLS at the database, session and even column level, with the `CHAR(10 BYTE|CHAR)` syntax. SQL Server supports non-UCS-2 character sets (Latin1, BIG5) in CHAR/VARCHAR/TEXT columns using BLS the size, while NCHAR/NVARCHAR/NTEXT columns store double-byte UCS-2 characters and use CLS.

This table shows the character data type length semantics of support database servers:

Table 122: Character data type length semantics of supported database servers

Database Engine	Length semantics in character data types	Summary
Oracle	Supports both Byte or Character Length Semantics in character type definition, can be defined globally for the database or at column level. Character string data is stored in database character set for CHAR /VARCHAR columns and in national character set for NCHAR /NVARCHAR columns.	BLS/CLS

Database Engine	Length semantics in character data types	Summary
Informix®	Uses Byte Length Semantics for the size of character columns. Can apply a ratio when creating columns, according to the SQL_LOGICAL_CHARS server configuration parameter. Character string data is stored in the database character set defined by DB_LOCALE.	BLS
IBM® DB2®	Uses Byte Length Semantics for the size of character columns. Character data is stored in the database character set defined by the CODESET of CREATE DATABASE.	BLS
Microsoft™ SQL Server	CHAR / VARCHAR sizes are specified in bytes; Data is stored in the character set defined by the database collation. NCHAR / NVARCHAR sizes are specified in characters; Data is stored in UCS-2. See SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 on page 592 for more details.	BLS/CLS
PostgreSQL	Uses Character Length Semantics for the size of character columns. Character string data is stored in the database character set defined by WITH ENCODING of CREATE DATABASE.	CLS
MySQL	Uses Character Length Semantics for the size of character columns. Character string data is stored in the server character set defined by a configuration parameter.	CLS
SQLite	Uses Character Length Semantics for the size of character columns. Character string data is stored in UTF-8.	CLS

Database Engine	Length semantics in character data types	Summary
Sybase Adaptive Server Enterprise (ASE)	CHAR / VARCHAR sizes are specified in bytes; Data is stored in the db character set. NCHAR / NVARCHAR sizes are specified in characters; Data is stored the db character set. UNICHAR / UNIVARCHAR sizes are specified in characters; Data is stored in UTF-16. See SQL adaptation guide for SAP Sybase ASE 16.x on page 723 for more details.	BLS/CLS

Other SQL elements like functions and operators are affected by the length semantic. For example, Informix® LENGTH() function always returns a number of bytes, while Oracle's LENGTH() function returns a number of characters (use LENGTHB() to get the number of bytes with Oracle).

It is important to understand properly how the database servers handle multibyte character sets. Check your database server reference manual: In most documentations you will find a "Localization" chapter which describes those concepts in detail.

Extracting database schemas

Database schema files (.sch) are used to resolve column data types when compiling .4gl modules and .per form files. This file contains size information for CHAR and VARCHAR types. It is important to identify the unit used by the database columns, to properly define CHAR/VARCHAR variables in programs and fields in forms.

Most database engines (like Oracle, SQL Server, PostgreSQL, Sybase, SQLite) provide catalog tables with column size information in character units. In this case, the fgldbsch tool extracts the column sizes in character units, without further conversion. If the column sizes is provided in bytes by catalog tables, fgldbsch will try to detect character length semantics usage in the database and apply a reduction factor to convert the number of bytes to chars.

As result - independently from the length semantics used in your programs - the CHAR/VARCHAR type sizes in the schema file are always expressed in character units. When using Byte Length Semantics, this makes no difference in a single-byte locale, because one character occupies a single byte. In a multibyte encoding (UTF-8) with BLS, this method garanties that the program variable will not hold more ASCII characters than the database column can hold. When using Character Length Semantics with a multibyte character set, the size in characters will define character type variables in the same unit.

For example, with BLS, a VARCHAR(10 *(bytes or chars)*) column will define a VARCHAR(10 *(bytes)*) in programs. With CLS, a VARCHAR(10 *(chars)*) column will define a VARCHAR(10 *(chars)*) in programs.

Moving from single-byte to UTF-8

Migration to Unicode (UTF-8) is facilitated with Char Length Semantics:

1. Verify that your database uses Char Length Semantics.
2. Convert your sources and string files from your single-byte locale to UTF-8 (iconv).
3. Turn on Char Length Semantics with FGL_LENGTH_SEMANTICS=CHAR.
4. Compile and run your programs untouched.

Collation ordering settings

The runtime system supports a sorting functionality in tables. To sort the data rows, the runtime systems uses the standard C library functions to order character strings.

The environment variable **LC_COLLATE** can be used to control sort order in Genero. You can for example define this variable as "C" or "POSIX" to get a binary sort order.

When using **LC_COLLATE**, set the **LANG** environment variable to define the global locale, if you use **LC_ALL**, it will overwrite all other LC_* variables defined.

Numeric and currency locale settings

The environment variables **LC_MONETARY** and **LC_NUMERIC** are ignored. To perform decimal to/from string conversions, the runtime system uses the DBMONEY or DBFORMAT environment variables. These variables define hundreds / decimal separators and currency symbols for `MONEY` data types.

Date and time locale settings

The environment variable **LC_TIME** is ignored. To perform date to/from string conversions, the runtime system uses by default the DBDATE environment variable.

Database client settings

This section describes the settings defining the locale for the database client. Each database software has its own client character set configuration.

You must properly configure the database client locale in order to send/receive data to the database server, according to the locale used by your application. Both database client locale and application locale settings must match (you cannot have a database client locale in Japanese and a runtime locale in Chinese).

Here is the list of environment variables defining the locale used by the application, for each supported database client:

Table 123: Environment variables defining the locale used by the application for each database client

Database Client	Settings
Oracle database server	The client locale settings can be set with environment variables like **NLS_LANG**, or after connection, with the ALTER SESSION instruction. By default, the client locale is set from the database server locale.
IBM® Informix®	The client locale is defined by the **CLIENT_LOCALE** environment variable. For backward compatibility, if CLIENT_LOCALE is not defined, other settings are used if defined (DBDATE / DBTIME / GL_DATE / GL_DATETIME, as well as standard LC_* variables).
IBM® DB2®	The client locale is defined by the **DB2CODEPAGE** profile variable. You cat set this variable with the db2set command. However, you usually do not need to set this variable: If DB2CODEPAGE is not set, DB2® uses the operating system code page on Windows™ and the LANG/LC_ALL locate setting on UNIX™. When using a UTF-8 locale on Windows™, DB2CODEPAGE must be set to **1208**.

Database Client	Settings
Microsoft™ SQL Server	For **MSV** and **SNC** drivers on Windows™ platforms, the database client locale is defined by the language settings for non-Unicode applications. The current ANSI code page (ACP) is used by the SQL Server client and the Genero runtime system. When using the **FTM** (FreeTDS) driver, the client character set is defined by the **client charset** parameter in freetds.conf or with the **ClientCharset** parameter in the DSN of the odbc.ini file. When using the **ESM** (EasySoft) driver, the client character set is defined by the **Client_CSet** parameter in the DSN of the odbc.ini file. When using CHAR/VARCHAR types in the database and when the database collation is different from the client locale, you must also set the Server_CSet parameter to an iconv name corresponding to the database collation. For example, if Client_CSet=BIG5 and the db collation is Chinese_Taiwan_Stroke_BIN, you must set Server_CSet=BIG5HKSCS, otherwise invalid data will be returned from the server.
PostgreSQL	The client locale can be set with the **PGCLIENTENCODING** environment variable, with the *client_encoding* configuration parameter in postgresql.conf, or after connection, with the SET CLIENT_ENCODING instruction. Check the *pg_conversion* system table for available character set conversions.
Oracle MySQL	The client locale is defined by the ***default-character-set*** option in the MySQL configuration file. The character set could also be changed by program after the connection, with the SET NAMES or SET CHARACTER SET statements, but this not supported: The driver needs to know the character set at connection initialization, and you would have to add this statement in all your programs.
Sybase Adaptive Server Enterprise (ASE)	By default, the Sybase database client character set is defined by the **operating system locale** where the database client runs. On Windows™, it is the ANSI code page of the login session (can be overwritten by setting the LANG environment variable), on UNIX™ it is defined by the LC_CTYPE, LC_ALL or LANG environment variable. You may need to edit the $SYBASE/locales/locales.dat file to map the OS locale name to a known Sybase character set. See Sybase ODBC documentation for more details regarding character set configuration.

See database vendor documentation for more details.

Front-end settings

The host operating system on the front-end workstation must be able to handle the character set and fonts. For instance, a Western-European Windows™ is not configured to handle Arabic applications. If you start an Arabic application, some graphical problems may occur (for instance the title bar won't display Arabic characters, but unwanted characters instead).

The GUI front-end software must support the conversion of the runtime system character set to/from the character set used internally by the client, and must be configured with the correct font to display the characters used by the application. For example, the default font for a front-end installed on an English Windows™ system might not be able to display Japanese characters. You must then change the font in the front-end configuration panel. Refer to the front-end documentation to see how character set conversion and fonts can be configured.

When using a TUI program in a terminal emulator such as Putty, XTerm or even the Windows™ Console, make sure the terminal is configured properly to display the characters of the application locale. For example, on a Windows™ Console you can use the *chcp* command to change the current code page.

Writing programs

Development and runtime character set must match

When writing a form or program source file, you use a specific character set. This character set depends upon the text editor or operating system settings you are using on the development platform. For example, when writing a string constant in a .4gl module, containing Arabic characters, you probably use the ISO-8859-6 character set. The character set used at runtime (during program execution) must match the character set used to write programs.

At runtime, a Genero program can only work in a specific character set. However, by using localized strings, you can start multiple instances of the same compiled program using different locales. For a given program instance the character set used by the strings resource files must correspond to the locale. Make sure the string identifiers use ASCII only.

Byte length semantics and substring expressions

When using Byte Length Semantics (BLS), all character positions in strings are actually byte positions. In a multibyte environment, if you don't pay attention to this, you can end up with invalid characters in strings. For example, an expression using a subscript operator `[x,y]` might refer to a byte position which is in fact in the middle of a multibyte character. If possible, use Character Length Semantics (CLS) with a multibyte locale to avoid such problems, or use only `STRING` methods to parse character strings.

Runtime system messages

While it is recommended to use localized strings to internationalize application messages, runtime system error messages are provided in **.iem** message files. The system message files use the same technique as user defined message files. The default message files (`msg`) are located in the `FGLDIR/msg/en_US` directory.

For backward compatibility with IBM® Informix® 4GL, some of these system error messages are used by the runtime system to display messages during a dialog instruction. For example, end users may get the error message -1309 "*There are no more rows in the direction you are going*" when scrolling an a `DISPLAY ARRAY` list in TUI mode.

If your application language is not English, you will need to translate some of the system messages to a specific locale and language. If your application language is English, you might just want to customize the default messages.

Here are some examples of system messages that can appear at runtime:

Table 124: Examples (subset) of system messages for localized strings

Number	Description
-1204	Invalid year in date.
-1304	Error in field.
-1305	This field requires an entered value.
-1306	Please type again for verification.
-1307	Cannot insert another row - the input array is full.
-1309	There are no more rows in the direction you are going.

To use your own customized system messages, do this:

1. Create a new directory under $FGLDIR/msg, using the same name as your current locale. For example, if `LANG=fr_FR.ISO8859-1`, you must create `$FGLDIR/msg/fr_FR.ISO8859-1`.
2. Copy the original system message source files (.msg) from `$FGLDIR/msg/en_US` to the locale-specific directory.
3. Edit the source files with the **.msg** suffix and translate the messages.
4. Recompile the message files with the fglmkmsg tool to produce **.iem** files. Make sure you have set the correct locale!
5. Run a program to check if the new messages are used.

With this technique, you can deploy multiple message files in different languages and locales in the same FGLDIR/msg directory.

You can use the fglmkmsg tool with the **-r** option to revert a **.iem** file to a source **.msg** file.

There is no need to translate all messages of the **.msg** files: Most of the error messages are unexpected during a program execution and therefore can stay in English. The messages subject of translation can be found in the **4glusr.msg** and **rds.msg** files.

The locale can be set with different environment variables (see setlocale manual pages for more details). To identify the locale name, the runtime system first looks for the LC_ALL value, then LC_CTYPE and finally LANG.

Pay attention to locale settings when editing message files and compiling with fglmkmsg: The current locale must match the locale used in the .msg files.

The **.iem** files used at runtime must match the current locale used by programs. This should be automatic, as long as you put the correct files in the corresponding `$FGLDIR/msg/$LANG` directory.

Using the charmap.alias file

The name of the character set defined within the LANG/LC_ALL environment variables can wary from system to system. For example, on a given platform, the ISO-8859-1 character set may be named "iso88591", while others platform will use "8859-1".

An example of locale configuration on HP/UX:

```
$ export LANG=en_US.iso88591
$ locale
LANG=en_US.iso88591
LC_CTYPE="en_US.iso88591"
LC_COLLATE="en_US.iso88591"
LC_MONETARY="en_US.iso88591"
LC_NUMERIC="en_US.iso88591"
LC_TIME="en_US.iso88591"
LC_MESSAGES="en_US.iso88591"
```

```
LC_ALL=
$ locale charmap
"iso88591.cm"
```

To communicate with other components like front-ends, or identify the encoding of XML files, Genero programs must use a normalized name for character sets. This normalized name must follow the IANA specifications [RFC2978].

In order to convert the operating system specific locale codeset name to an IANA name, the runtime system uses the `charmap.alias` mapping file, located in `$FGLDIR/etc`.

You can add your operating system specific locale, if not listed in the `s` file.

Date, numeric and monetary formats

Dates, numbers and monetary values must be displayed and entered in a format used in the country/region. These formats can be defined with the DBDATE and DBFORMAT environment variables.

Date and numeric format settings matter for data display and data input. For example, when displaying a `DATE` value to a form field, it will implicitely be formatted according to DBDATE. When the user enters a date in a form field bound to a `DATE` variable, the entered digits will be interpreted according the DBDATE.

The default value of these environment variables depends on the type of platform where the program executes:

When using the `FORMAT` field attribute or the `USING` operator to format dates with abbreviated day and month names- by using **ddd** / **mmm** markers - the system uses English-language based texts for the conversion. This means, day (ddd) and month (mmm) abbreviations are not localized according to the locale settings, they will always be in English.

- On destkop/server platforms, the default formats are set for the United States of America:
  - Dates are formatted as `mm/dd/yyyy`.
  - The decimal separator is a dot.
  - The currency symbol is the `$` dollar sign.
- On mobile platforms, the default formats are set according to the regional settings defined on the device.
  - Dates are formatted according to the regional settings.
  - The decimal separator is defined according to the regional settings.
  - The currency symbol is not defined. No currency symbol will display.

    Note: While it is possible to define environment settings for date and numeric formats with FGLPROFILE entries, it is strongly recommended to leave the defaults, to get the expected formats, if the user changes the regional settings on the mobile device.

Using the Ming Guo date format

The Ming Guo (or Minguo) calendar is still used in some Asian regions like Taiwan. This calendar is equivalent to the Gregorian calendar, except that the years are numbered with a different base: In the Ming Guo calendar, the first year (1) corresponds to the Gregorian year 1912, the year the Republic Of China was founded.

Digit-based year Ming Guo date format can be enabled by adding the C1 modifier at the end of the value set for the DBDATE environment variable:

```
$ DBDATE="Y3MD/C1"
$ export DBDATE
```

With this DBDATE setting, dates will be displayed with a year following the Ming Guo calendar, and date input will also be interpreted based on that calendar. For example, if the user enters 90/3/24, it is

equivalent to an input of 2002/3/24 when using the Gregorian calendar. Basically, the runtime system will subtract 1912 or add 1912 respectively when displaying or reading date values).

When using the C1 modifier, the possible values for the Yn symbol are Y4, Y3, Y2.

The `MDY( )` operator is sensitive to the C1 modifier usage in DBDATE. For example, if DBDATE=Y3MD/C1, MDY(3,24,1) will build a date the corresponds in the Gregorian to MDY(3,24,1912).

The `USING` operator supports the c1 modifier as well. The c1 modifier must be specified at the end of the format. You can for example use the following format string: "yyyy-mm-ddc1".

The C2 modifier to use Era names is not supported.

Unlike Informix® 4gl, when using negative years, the minus sign is placed over the left-most zero of the year, to avoid miss-aligned dates.

For example, if DBDATE=Y3MD/C1:

```
MDY(3,2, 1) USING "yyy/mm/ddc1"
MDY(3,2,-1) USING "yyy/mm/ddc1"
```

Will align properly as follows:

```
0001/03/02
-001/03/02
```

Note: Front-ends may not support the Ming Guo calendar for widgets like `DATEEDIT`.

Troubleshooting locale issues

Locale settings (LANG) corrupted on Microsoft™ platforms

On Microsoft™ Windows™ XP / 2000 platforms, some system updates (Services Pack 2) or Office versions do set the LANG environment variable with a value for Microsoft™ applications (for example 1033). Such value is not recognized by Genero as a valid locale specification. Make sure that the LANG environment variable is properly set in the context of Genero applications.

A form is displayed with invalid characters

You may have different codesets on the client workstation and the application server. The typical mistake that can happen is the following: You have edited a form-file with the encoding CP1253; you compile this form-file on a UNIX-server (encoding ISO-8859-7). When displaying the form, invalid characters will appear. This is usually the case when you write your source file under a Windows™ system (that uses Microsoft™ Code Page encodings), and use a Linux™ server (that uses ISO codepages).

Keep in mind that all source files must be created/edited in the encoding of the server (where fglcomp and fglrun will be executed).

Checking the locale configuration on UNIX™ platforms

On UNIX™ systems, the **locale** command without parameters outputs information about the current locale environment.

Once the **LANG** environment variable is set, check that the locale environment is correct:

```
$ export LANG=en_US.ISO8859-1
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE="en_US.ISO8859-1"
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
```

```
LC_PAPER="en_US.ISO8859-1"
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT="en_US.ISO8859-1"
LC_IDENTIFICATION="en_US.ISO8859-1"
LC_ALL=
```

If the locale environment is not correct, then you should check the value of the following environment variables: LC_ALL, LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, ... value.

The following examples show the effect of LC_ALL and LC_CTYPE on locale configuration. The LC_ALL variable overrides all other LC_.... variables values.

```
$ export LANG=en_US.ISO8859-1
$ export LC_ALL=POSIX
$ export LC_CTYPE=fr_FR.ISO8859-15
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE="POSIX"
LC_NUMERIC="POSIX"
LC_TIME="POSIX"
LC_COLLATE="POSIX"
LC_MONETARY="POSIX"
LC_MESSAGES="POSIX"
LC_PAPER="POSIX"
LC_NAME="POSIX"
LC_ADDRESS="POSIX"
LC_TELEPHONE="POSIX"
LC_MEASUREMENT="POSIX"
LC_IDENTIFICATION="POSIX"
LC_ALL=POSIX
$ fglrun -i mbcs
LANG honored: yes
Charmap     : ANSI_X3.4-1968
Multibyte   : no
Stateless   : yes
```

The charset used is the ASCII charset. Clearing the LC_ALL environment variable produces the following output:

```
$ unset LC_ALL
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE=fr_FR.ISO8859-15
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
LC_PAPER="en_US.ISO8859-1"
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT="en_US.ISO8859-1"
LC_IDENTIFICATION="en_US.ISO8859-1"
LC_ALL=
$ fglrun -i mbcs
Error: locale not supported by C library, check LANG.
$ locale charmap
ANSI_X3.4-1968
```

After clearing the LC_ALL value, the value of the variable LC_CTYPE is used. It appears that it is not correct. After clearing this value we get the following output:

```
$ unset LC_CTYPE
$ locale
LANG=en_US.ISO8859-1
LC_CTYPE="en_US.ISO8859-1"
LC_NUMERIC="en_US.ISO8859-1"
LC_TIME="en_US.ISO8859-1"
LC_COLLATE="en_US.ISO8859-1"
LC_MONETARY="en_US.ISO8859-1"
LC_MESSAGES="en_US.ISO8859-1"
LC_PAPER="en_US.ISO8859-1"
LC_NAME="en_US.ISO8859-1"
LC_ADDRESS="en_US.ISO8859-1"
LC_TELEPHONE="en_US.ISO8859-1"
LC_MEASUREMENT="en_US.ISO8859-1"
LC_IDENTIFICATION="en_US.ISO8859-1"
LC_ALL=
$ locale charmap
ISO-8859-1
$ fglrun -i mbcs
LANG honored: yes
Charmap     : ISO-8859-1
Multibyte   : no
Stateless   : yes
```

Verifying if the locale is properly supported by the runtime system

You can check if the LANG/LC_ALL locale is supported properly by using the `-i mbcs` option of the compilers and runner programs:

```
$ fglcomp -i mbcs
Charmap          : UTF-8
Multibyte        : yes
Stateless        : yes
Length Semantics : CHAR
```

The lines printed with this option indicate if the locale can be supported by the operating system libraries. Here is a short description of each line:

Table 125: -i info line descriptions

Verification Parameter	Description
Charmap	This is the normalized IANA name of the character set used by the runtime system to communicate with external components (front-end, I/O of XML files). The mapping from the system locale name to a normalized name is defined in `FGLDIR/etc/charmap.alias`.
Multibyte	This line indicates if the character set is multibyte.
Stateless	A few character sets are using an internal state that can change during the character flow. Only stateless character sets can be supported (the value must be 'yes').

How to retrieve the list of available locales on the system

On UNIX™ systems, the locale command with the parameter '-a' writes the names of available locales.

```
$ locale -a
...
en_US
en_US.iso885915
en_US.utf8
en_ZA
en_ZA.utf8
en_ZW
...
```

How to retrieve the list of available codesets on the system

On UNIX™ systems, the locale command with the parameter '-m' writes the names of available codesets.

```
$ locale -m
...
ISO-8859-1
ISO-8859-10
ISO-8859-13
ISO-8859-14
ISO-8859-15
...
```

Localized strings

Localized strings provide a means of writing applications in which the text of strings can be customized on site.

This string localization feature is a simple way to define external resource files which the runtime system can search, in order to assign text to elements displayed by programs. It can be used to implement internationalization in your application, or to use site-specific text, for example, when business terms are specific to the territory where the application is used.

The localized string resource files (`.42s`) are loaded at runtime and shared by all fglrun processes. Localized strings are used to replace the original strings found in the p-code modules (`.42m`), in the compiled form (`.42f`), and in any XML resource files loaded in the abstract user interface tree (`.4ad` , `.4st` , `.4tb` , etc).

- Steps for application internationalization on page 327
- Creating source string files on page 328
- Localized strings in program sources on page 329
- Localized strings in XML resource files on page 330
- Extracting strings from sources on page 331
- Compiling string files on page 331
- Using localized strings at runtime on page 331
- Predefined application strings on page 334
- Example on page 334

Steps for application internationalization

Follow these steps to internationalize your application.

1. Identify the current character set used in your sources and make sure the application locale (LANG/LC_ALL) is set correctly.
2. In `.4gl` sources, add a % prefix to the strings that must be localized (i.e. translated).
3. In `.per` sources `LAYOUT` section, replace hard-coded form elements like text labels by static `LABEL` form items and define the `TEXT` attributes with a % prefix in the `ATTRIBUTES` section.

4. Extract the strings from the .4gl sources with `fglcomp -m` and use `fglform -m` for .per sources.
5. Organize the generated `.str` source string files (identify duplicated strings and put them in a common file).
6. At this point, the string identifiers (on the left) are the same as the string texts (on the right). These localized strings could be used as is, but it's better to define a normalized identifier for each string, by using ASCII characters only. For example, replace:

   ```
   "Customer List" = "Customer List"
   ```

   with:

   ```
   "customer.list.title" = "Customer List"
   ```
7. In sources, replace the original string text with the new string identifiers. Strings to be replaced can be located by their % prefix. You can, for example, use a script with an utility like the *sed* UNIX™ command to read the `.str` files and apply the changes automatically.
8. Recompile the `.4gl` and `.per` sources (these should be ASCII now, so the locale should not matter).
9. Compile the `.str` files in the locale used by these files, and check whether the application displays the text properly.
10. Copy the existing `.str` files, and translate the string text into another language (making sure the locale is correct).
11. Compile the new `.str` files, and copy the `.42s` files into another distribution directory, defined with the FGLRESOURCEPATH environment variable.

    A set of `.42s` files using the same language and codeset is typically copied in a distribution directory with a name identifying the locale.

    For example:

    ```
    /opt/app/resource/strings/en_US.iso8859-1
      -- English strings in iso8859-1 code-set
    /opt/app/resource/strings/fr_FR.iso8859-1
      -- French strings in iso8859-1 code-set
    /opt/app/resource/strings/jp_JP.utf8
      -- Japanese strings in utf-8 code-set
    ```

Future edits to the `.per` and `.4gl` source files should be done in the ASCII locale, and `.str` string files must be edited with their specific locale.

Creating source string files

A *source string file* contains localized string definitions for a given language (or localization context).

What is a source string file?

A source string file is basically a mapping table that defines an identifier for each string.

After compiling source string files, the programs can load and use a string found according to its identifier (or key).

By convention, the source files of localized strings have the `.str` extension.

Syntax

Define a list of string identifiers, and the corresponding text, by using the following syntax:

```
"string-identifier" = "string-text"
```

For example:

```
"common.button.cancel" = "Cancel"
```

> **Note:** Localized string keys are case sensitive. Consider using lower case characters only to avoid mistakes.

As an alternative, you can define string identifiers as a dot-separated list of identifiers:

```
identifier. [...] = "string-text"
```

For example:

```
common.button.cancel = "Cancel"
```

If needed, you can add comment lines with the # or -- markers, like in other Genero source files:

```
# a comment
-- another comment
```

Special characters

The fglmkstr compiler accepts the backslash "\" as the escape character, to define non-printable characters:

```
\l   \n   \r   \t   \\
```

Example

```
# A comment line
"Original text" = "Original text"
"forms.customer.list" = "Customer List"
"special.characters.backslash" = "\\"
"special.characters.newline" = "\n"
```

Localized strings in program sources

A *localized string* is specified in the source code of program modules or form specification files with the `%"string"` notation, to identify a string that must be replaced at runtime by the corresponding text found in compiled string files. In programs, localized strings can be loaded dynamically with the `LSTR()` operator.

Syntax 1: Static localized string

```
%"sid"
```

1. *sid* is a character string literal that defines both the string identifier and the default text.

Syntax 2: Dynamic localized string

```
LSTR(eid)
```

1. *eid* is a character string expression used at runtime as the string identifier to load the text.

Static localized strings

A static localized string specification begins with a percent sign (%), followed by the identifier of the string which will be used to find the text to be loaded. Since the identifier is a string, you can use any type of

characters, but it is recommended that you use a naming convention. For example, you can specify a path by using several names separated by a dot:

```
MAIN
  DISPLAY %"common.message.welcome"
END MAIN
```

The string after the percent sign defines both the localized string identifier and the default text to be used for extraction, or the default text when no string resource files are provided at runtime.

You can use this notation in form specification files any place where a string literal can be used.

```
LAYOUT
  VBOX
    GROUP g1 (TEXT=%"group01")
...
```

It is not possible to specify a static localized string directly in the area of containers like `GRID`, `TABLE`, `TREE` or `SCROLLGRID`. You can use static label form items to define localized strings in layout labels:

```
LAYOUT
  GRID
  {
    [lab01  |f001             ]
  {
  END
END
ATTRIBUTES
LABEL lab01: TEXT=%"myform.label01";
EDIT f001 = FORMONLY.field01;
END
```

Dynamic localized strings

The language provides a special operator to load a localized string dynamically, using an expression as string identifier. The name of this operator is `LSTR()`.

The following code example builds a localized string identifier with an integer and loads the corresponding string with the `LSTR()` operator:

```
MAIN
  DEFINE n INTEGER
  LET n = 234
  DISPLAY LSTR("str"||n)  -- loads string 'str234'
END MAIN
```

Localized strings in XML resource files

In XML resource files, localized string specification must follow the XML syntax and therefore must be defined as an XML node.

Syntax: Localized string in XML files

```
  <ParentNode attribute = "default" [...] >
     <LStr attribute = "sid" [...] />
  </ParentNode>
```

1. *ParentNode* is the node type of the parent where the localized strings must be applied.
2. *attribute* is the attribute in the parent node that will get the localized string identified by *sid*.
3. *default* is the default text of an attribute, if not localized string is found for *sid*.

4. *sid* is a character string literal that defines both the string identifier and the default text.

Description

In `.42m` p-code modules, the localized strings are coded in a proprietary binary format. But, for XML files such as action defaults files (`.4ad`), the localized strings must be written with a specific node, following the XML standards. To support localized strings in XML files, any file loaded into the Abstract User Interface tree is parsed to search for `<LStr>` nodes. The `<LStr>` nodes define the same attributes as in the parent node with localized string identifiers, for example:

```
<Label text="Hello!" >
  <LStr text="label01" />
</Label>
```

The runtime system automatically replaces corresponding attributes in the parent node (text="Hello!"), with the localized text found in the compiled string files, according to the string identifier (label01). After interpretation, the `<LStr>` nodes are removed from the XML data.

To take effect, a localized attribute in the `<LStr>` node must have a corresponding attribute in the parent node.

Extracting strings from sources

Localized strings can be easily extracted from `.4gl` and `.per` source files.

Use the `fglcomp` and `fglform` compilers with the `-m` option to extract localized strings.

```
$ fglcomp -m mymodule.4gl
```

The compilers dumps all localized string to stdout. This output can be redirected to a file to generate the default source string file with all the localized strings used in the source file. Source string files should then be re-organized, to centralize common messages in a unique `.str` file, and can then be compiled by fglmkstr into `.4st` files to be used by the runtime system.

Compiling string files

The source string files (with `.str` extension) must be compiled to binary files (with `.42s` extension) in order to be loaded by the runtime system.

To compile a source string file, use the `fglmkstr` compiler.

```
$ fglmkstr filename.str
```

The `fglmkstr` tool generates a `.42s` file with the *filename* prefix.

> **Important:** When compiling a `.str` source string file, you must set the locale (character set) corresponding to the encoding used in the `.str` file.

Using localized strings at runtime

Understand the rules for using localized strings at runtime.

Distributing compiled string files

The compiled string files (`.42s`) must be distributed with the program files in a directory specified in the `DBPATH`/`FGLRESOURCEPATH` environment variable.

Setting the correct locale

The locale (`LANG`/`LC_ALL`) corresponding to the encoding used in the `.42s` files must be set before starting the application. If the locale is wrong, the strings will not be loaded properly.

How does the runtime system load the strings?

The `.42s` compiled string resource files are loaded in following order of precedence:

1. The files defined in `FGLPROFILE`,
2. A file having the same name as the current program (`myprog.42m` loads `myprog.42s`),
3. A file with the name "`default.42s`".

For each string file, the runtime system looks in the following directories:

1. The current directory,
2. The path list defined in the `DBPATH`/`FGLRESOURCEPATH` environment variable,
3. The `FGLDIR/lib` directory.

String resource file sharing

Like `.42m` program pcode files, the `.42s` string resource files are shared by all fglrun processes running on the computer: The string file is loaded into memory with the mmap operating system function.

Defining a list of string files in FGLPROFILE

Specify a list of compiled string files with entries in the FGLPROFILE configuration file with the `fglrun.localization` entries.

First, define the total number of files with:

```
fglrun.localization.file.count = integer
```

For each file, define the filename (with the `.42s` extension), including an index number (start index must be 1):

```
fglrun.localization.file.index.name = "filename.42s"
```

Warning switches can be specified in FGLPROFILE.

If the text of a string is not found at runtime, the runtime system can show a warning, for development purposes.

```
fglrun.localization.warnKeyNotFound = boolean
```

By default, this warning switch is disabled.

What happens if a 42s string file is not found?

If the 42s string file was defined with `fglrun.localization.*` FGLPROFILE entries, it is considered as mandatory, and the runtime system will raise error -8006 if the file is not found. If the *`progname`*`.42s` and `default.42s` string files are not found, no error is raised, because these are fallback string resource files.

What happens if a string is not defined in a resource file?

If a localized string is not defined in one of the compiled string files, the runtime system uses the string identifier as default text.

What happens if a string is defined more that once?

When a localized string is defined in several compiled string files, the runtime system uses the first string found.

For example, if the string "hello" is defined in `program.42s` as "hello from program", and in `default.42s` as "hello from default", the runtime system will use the text "hello from program".

Organizing .42s resource files in distribution directories

A set of `.42s` files using the same language and codeset is typically copied in a distribution directory with a name identifying the locale.

For example:

```
/opt/app/resource/strings/en_US.iso8859-1  -- English strings in iso8859-1
 code-set
/opt/app/resource/strings/fr_FR.iso8859-1  -- French strings in iso8859-1
 code-set
/opt/app/resource/strings/jp_JP.utf8       -- Japanese strings in utf-8
 code-set
```

At runtime, specify the string file search path in the `DBPATH`/`FGLRESOURCEPATH` environment variable by adding the name of current locale as sub-directory. For example, to find the correct string files in one of the locale-specific directories shown above, set the `FGLRESOURCEPATH` variable as follows (UNIX™ shell):

```
$ echo $LC_ALL
jp_JP.utf8
$ FGLRESOURCEPATH="$FGLRESOURCEPATH:/opt/app/resource/strings/$LC_ALL"
$ export FGLRESOURCEPATH
$ echo $FGLRESOURCEPATH
/opt/app/forms:/opt/app/resource/strings/jp_JP.utf8
```

Localized string files on mobile devices

On mobile devices, the language is determined by the operating system regional settings.

- On iOS devices (version 8.1), the language is determined by Settings >> General >> International >> Language
- On Android™ devices (version 4.4), the language is determined by Settings >> Language & Input >> Language

The selected language is identified by a locale code following the ISO 639 standard. Below are some language code examples; see the mobile OS documentation for information about available languages and their corresponding ISO 639-x codes.

- `en` - English (for all regions)
- `en_US` - English in the United States
- `en_GB` - English in the United Kingdom

On startup, the mobile app will by default search for localized string files (.42s) in *`appdir/locale-code`*, the application sub-directory having the same name as the locale identifier (with language and category/region codes) (for an English-US locale: *`appdir/`*`en_US`). If the 42s files are not found in this sub-directory, the runtime system tries to load the files from a sub-directory with the language identified only (for an English-US locale: *`appdir/`*`en`). Finally, if the string files are not found in locale-specific directories, the files are loaded directly from *appdir*.

In order to localize your application, you simply need to place your `.42s` localized string files in the appropriate language sub-directory.

> **Note:** If the `.42s` file names do not match the main program name, define the list of localized strings files in app's `fglprofile` file.

If you want to distinguish language categories (Simplified/Tradition Chinese), or if you want to use different texts according to the territory for the same language (English in USA or Great Britain), create language sub-directories with the exact OS locale identifier:

- For English in the USA, use "`en_US`"
- For English in the United Kingdom, use "`en_GB`"

- For English in Canada, use "en_CA"
- etc...

```
appdir/en_US/mystrings.42s
appdir/en_GB/mystrings.42s
appdir/en_CA/mystrings.42s
```

If the language category or region can be ignored, create language sub-directories with names matching the language identifier only:

- For English, use "en"
- For French, use "fr"
- For German, use "de"
- etc...

```
appdir/en/mystrings.42s
appdir/fr/mystrings.42s
appdir/de/mystrings.42s
```

Consider providing a default set of string files (in English) directly under *appdir*, in case if the regional settings of the device do not match one of the locale directories you provide, otherwise the application will stop with error -8006:

```
appdir/mystrings.42s
```

For more details about the mobile app directory structures (*appdir*), see Directory structure for GMA apps on page 2572 and Directory structure for GMI apps on page 2584.

Predefined application strings
The runtime system may need to display text to the user.

For example, the runtime system library includes a report viewer, which displays a form. By default the text is in English, and you may need to localize the text in another language. So the strings of this component must be 'localizable', as in other application strings.

To customize the built-in strings, the runtime system uses the mechanism of localized strings.

All strings used by the runtime system are centralized in a unique file:

```
$FGLDIR/src/default.str
```

which is compiled into:

```
$FGLDIR/lib/default.42s
```

This file is always loaded by the runtime system.

To overwrite the defaults, you can redefine these strings in your own localized string files.

Example
Here is an example using localized strings.

The source string file "common.str" (to be compiled with fglmkstr):

```
"common.accept" = "OK"
"common.cancel" = "Cancel"
"common.yes" = "Yes!"
"common.no" = "No!"
```

The source string file "customer.str" (to be compiled with fglmkstr):

```
"customer.mainwindow.title" = "Customers"
"customer.listwindow.title" = "Customer List"
"customer.l_custnum" = "Number:"
"customer.l_custname" = "Name:"
"customer.c_custname" = "The customer name"
"customer.q_delete" = "Do you want to delete this customer?"
```

The FGLPROFILE configuration file parameters:

```
fglrun.localization.file.count = 1
fglrun.localization.file.1.name = "common.42s"
```

Remark: The 'customer' string file does not have to listed in FGLPROFILE since it is loaded as it has the same name as the program.

The form specification file "customer.per":

```
ACTION DEFAULTS
  ACTION accept (TEXT=%"common.accept")
  ACTION cancel (TEXT=%"common.cancel")
END
LAYOUT (TEXT=%"customer.mainwindow.title")
GRID
{
[lab1       ] [f01           ]
[lab2       ] [f02                   ]
}
END
END
ATTRIBUTES
  LABEL lab1: TEXT=%"customer.l_custnum";
  EDIT f01 = FORMONLY.custnum;
  LABEL lab2: TEXT=%"customer.l_custname";
  EDIT f02 = FORMONLY.custname, COMMENT=%"customer.c_custname";
END
```

The program "customer.4gl" using the strings file:

```
MAIN
  DEFINE rec RECORD
           custnum INTEGER,
           custname CHAR(20)
         END RECORD
  OPEN FORM f1 FROM "customer"
  DISPLAY FORM f1
  INPUT BY NAME rec.*
    ON ACTION delete
       MENU %"customer.mainwindow.title"
                ATTRIBUTES(STYLE="dialog", COMMENT=%"customer.q_delete")
           COMMAND %"common.yes"
           COMMAND %"common.no"
       END MENU
  END INPUT
END MAIN
```

Runtime stack

The runtime stack is used to pass/return values to/from functions.

When passing arguments to a function or when returning values from a function, you are using the *runtime stack*. When you call a function, parameters are pushed on the stack; before the function code executes, parameters are popped from the stack in the local variables defined in the function. On the other hand, each parameter returned by a function is pushed on the stack and popped into variables specified in the `RETURNING` clause of the caller.

Elements are pushed on the stack in a given order, then popped from the stack in the reverse order. This is transparent to the programmer. However, if you want to implement a C extension, you must keep this in mind.

According to the data type, parameters are passed and returned by value or by reference. When an element is passed/returned by value, a complete copy of the value is passed. When an element is passed by reference, only the handle of the object is passed/returned. If the types allows it, elements passed by reference can be manipulated in the called function to modify the value.

Table 126: Function parameter and returning rules by language element type

Mode	Data type or data structure
By value	`BOOLEAN`, `BIGINT`, `INTEGER` , `SMALLINT`, `TINYINT`, `FLOAT` , `SMALLFLOAT`, `DECIMAL`, `MONEY`, `CHAR`, `VARCHAR`, `DATE`, `DATETIME`, `INTERVAL`, records and static arrays (cannot be returned).
By reference	Dynamic arrays, objects (from Java™, built-in or extension classes), `BYTE`/`TEXT`, `STRING` (but cannot be modified)

- Passing simple typed values as parameter on page 336
- Passing a record as parameter on page 337
- Passing a static array as parameter on page 337
- Passing a dynamic array as parameter on page 338
- Passing objects as parameter on page 338
- Passing a TEXT/BYTE as parameter on page 339
- Returning simple typed values from functions on page 339
- Returning dynamic arrays from functions on page 339
- Returning TEXT/BYTE values from functions on page 340
- Implicit data type conversion on the stack on page 340

Passing simple typed values as parameter

Simple data types such as `INTEGER`, `DECIMAL`, `VARCHAR` are passed by value in function parameters. When passing a function parameter by value, the runtime system pushes a copy of the data on the stack.

The `STRING` data type is an exception to this rule for simple types: elements of this type are passed by reference. In fact the runtime system passes a reference to the string value, so the actual string data is not copied on the stack as for other simple types. However, the value of the caller cannot be modified: If a `STRING` parameter gets a new value in a function, a new string reference is created. Passed `STRING` parameters improve performances compared to `CHAR`/`VARCHAR`, with the same semantics as `VARCHAR( )`.

When passing a simple typed value to a function, the local variable receiving the value can be changed without affecting the variable used by the caller:

```
MAIN
  DEFINE c CHAR(10), s STRING
  LET c = "abc"
```

```
  LET s = "def"
  CALL func(c,s)
  DISPLAY c -- Shows "abc"
  DISPLAY s -- Shows "def"
END MAIN

FUNCTION func(pc,ps)
  DEFINE pc CHAR(10), ps STRING
  DISPLAY c -- Shows "abc" (this is a copy of the string)
  DISPLAY s -- Shows "def" (this is the original string)
  LET pc = "zz" -- Assigns new value to local variable
  LET ps = "zz" -- Assigns new value to local variable
END FUNCTION
```

Passing a record as parameter

You can pass a `RECORD` structure as a function parameter with the dot star (`.*`) notation. In this case, the record is expanded and each member of the structure is pushed on the stack. The receiving local variables in the function can then be defined individually or with the same record structure as the caller. The next example illustrates this:

```
MAIN
  DEFINE rec RECORD
             a INT,
             b VARCHAR(50)
         END RECORD
  CALL func_r(rec.*)
  CALL func_ab(rec.*)
END MAIN

-- Function defining a record like that in the caller
FUNCTION func_r(r)
  DEFINE r RECORD
             a INT,
             b VARCHAR(50)
         END RECORD
  ...
END FUNCTION

-- Function defining two individual variables
FUNCTION func_ab(a, b)
  DEFINE a INT, b VARCHAR(50)
  ...
END FUNCTION
```

Passing a static array as parameter

It is possible to pass a complete static array as a function parameter, but this is not recommended. When passing a static array to a function, the complete array is copied on the stack and every element is passed by value. The receiving local variables in the function must be defined with the same static array definition as the caller:

```
MAIN
  DEFINE arr ARRAY[5] OF INT
  CALL func(arr)
END MAIN

-- function defining same static array as the caller
FUNCTION func(x)
  DEFINE x ARRAY[5] OF INT
  ...
```

```
END FUNCTION
```

Note that dynamic arrays are passed by reference.

Passing a dynamic array as parameter

Passing a dynamic array as a function parameter is legal and efficient. When passed as parameter, the runtime system pushes a reference of the dynamic array on the stack, and the receiving local variables in the function can then manipulate the original data.

Returning a dynamic array from a function is also possible: The runtime system pushes the reference of the dynamic array on the stack.

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF INT
  DISPLAY arr.getLength()
  LET arr = init(10)
  DISPLAY arr.getLength()
  CALL modify(arr)
  DISPLAY arr[50]
  DISPLAY arr[51]
  DISPLAY arr.getLength()
END MAIN

FUNCTION init(c)
  DEFINE c INT
  DEFINE x DYNAMIC ARRAY OF INT
  FOR i=1 TO c
     LET x[i] = i
  END FOR
  RETURN x
END FUNCTION

FUNCTION modify(x)
  DEFINE x DYNAMIC ARRAY OF INT
  LET x[50] = 222
  LET x[51] = 333
END FUNCTION
```

Output of the program:

```
0
10
222
333
51
```

Passing objects as parameter

Like other object oriented programming languages, objects of built-in classes or Java™ classes are passed by reference. It would not make much sense to pass an object by value, actually. The runtime pushes the reference of the object on the stack (i.e. the object handler is passed by value), and the reference is then popped to the receiving object variable in the function. The function can then be used to manipulate the original object.

```
MAIN
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL open(ch)
  CALL ch.close()
END MAIN
```

```
FUNCTION open(x)
  DEFINE x base.Channel -- Channel object reference
  CALL x.openFile("filename","r")
END FUNCTION
```

Passing a TEXT/BYTE as parameter

BYTE or TEXT data types define large data object (LOB) handlers internally implemented as "locators". When you pass a BYTE or TEXT to a function, the locator is pushed on the stack and popped to the receiving BYTE or TEXT variable in the function. The actual LOB data is not copied, only the locator is passed by value.

> **Important:** Since the information of the locator structure is copied (like the file name specified with a LOCATE IN FILE instruction). If you modify the locator storage information inside the function with a LOCATE instruction, the locator in the caller will become invalid. Therefore, only read and write the actual data of BYTE and TEXT parameters in functions, do not modify the storage.

Returning simple typed values from functions

Simple data types such as INTEGER, DECIMAL, VARCHAR are returned by value. When returning a simple typed value, the runtime system pushes a copy of the data on the stack. The STRING data type is an exception to this rule: elements of this type are return by mutable reference: the whole string value is not copied on the stack, only the reference to the string value is copied.

```
MAIN
  DEFINE x INTEGER
  LET x = int_add(10,20)
END MAIN

FUNCTION int_add(n1,n2)
  DEFINE n1, n2 INTEGER
  RETURN (n1+n2)
END FUNCTION
```

Returning dynamic arrays from functions

When returned by a function, dynamic arrays are pushed on the stack by reference. Therefore you can create a dynamic array in a function and return it to the caller for usage:

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  LET arr = create_array(10)
  DISPLAY arr.getLength()
END MAIN

FUNCTION create_array(n)
  DEFINE n, i INTEGER
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  FOR i=1 TO n
     LET arr[i] = i
  END FOR
  RETURN arr
END FUNCTION
```

Returning TEXT/BYTE values from functions

When returning a TEXT or BYTE value from a function, the locator is pushed in on the stack. Storage information of the TEXT/BYTE is defined in the locator structure, therefore you can define the storage of the large object variable in a function, initialize the object with a value, and return it.

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  LET arr = create_array(10)
  DISPLAY arr.getLength()
END MAIN

FUNCTION create_array(n)
  DEFINE n, i INTEGER
  DEFINE arr DYNAMIC ARRAY OF INTEGER
  FOR i=1 TO n
     LET arr[i] = i
  END FOR
  RETURN arr
END FUNCTION
```

Implicit data type conversion on the stack

When a value or a reference is popped from the stack, implicit data conversion takes place. This means, for example, that you can pass a string value to a function that defines the receiving variable as a numeric data type; no compilation error will occur, but you can get a runtime error if the string cannot be converted to a numeric. The same principle applies to values returned from functions, since the stack is also used in this case.

```
MAIN
  DEFINE s STRING
  LET s = "123"
  CALL display_integer(s) -- Will be accepted
  LET s = "abc"
  CALL display_integer(s) -- Will fail with conversion error
END MAIN

FUNCTION display_integer(x)
  DEFINE x INTEGER
  DISPLAY x
END FUNCTION
```

Exceptions

Describes exception (error) handling in the programs.

- Understanding exceptions on page 341
- Exception classes on page 341
- Exception actions on page 341
- WHENEVER instruction on page 342
- TRY - CATCH block on page 344
- Tracing exceptions on page 345
- Default exception handling on page 346
- Non-trappable errors on page 346
- Examples on page 346

Understanding exceptions

Exceptions are abnormal runtime events that can be trapped for control.

If an instruction executes abnormally, the runtime system throws exceptions that can be handled by the program.

Specific exception actions can be taken based on the class of the exception.

Runtime errors (i.e. exceptions) can be trapped by a `WHENEVER` exception handler or by a `TRY/CATCH` block. Note that some specific errors cannot be trapped.

A Genero exception is identified by its number and has a description. For a complete list of BDL errors, see Genero BDL errors on page 2297.

Exception handlers are typically used to detect database errors when executing SQL statement. For more details, see SQL execution diagnostics on page 398

Exception classes

Exception classes indirectly define the exception type.

The default action can be changed by specifying the exception class in the `WHENEVER` instruction.

Table 127: Exception classes

Class	Error reason	Default Action
`ERROR (or SQLERROR)`	Language or SQL statement error.	`STOP`
`ANY ERROR (or ANY SQLERROR)`	Language, SQL statement and expression error.	`CONTINUE` (1)
`NOT FOUND`	SQL statements returning status `NOTFOUND`.	`CONTINUE`
`WARNING`	SQL statements setting `SQLCA.SQLAWARN` flags.	`CONTINUE`

For example, the following `WHENEVER` instruction defines the behavior for the `ANY ERROR` exception class:

```
WHENEVER ANY ERROR CONTINUE
```

Exception actions

Exception actions define the type of action to be taken when an exception occurs.

There are five exception actions that can be executed if an exception is raised:

`STOP`
The program is immediately terminated. A message is displayed to the standard error with the location of the related statement, the error number, and the details of the exception.

`CONTINUE`
The program continues normally. The exception is ignored, but can be checked by testing the `STATUS`register, or the SQLCA.SQLCODE register for SQL errors.

`CALL` *exception-function*
The function *exception-function* is called by the runtime system. The function can be defined in any module, and must have zero parameters and zero

	return values. The STATUS variable will be set to the corresponding error number.
GOTO *exception-label*	The program execution continues at the label identified by *exception-label*, as if a GOTO instruction was issued after trapping the exception.
RAISE	This statement instructs the runtime system that the exception must propagated to the calling function. **Important:** Note that WHENEVER[ANY] ERROR RAISE is not supported in a REPORT routine.

WHENEVER instruction

Use the WHENEVER instruction to define how exceptions must be handled for the rest of the module.

Syntax

```
WHENEVER exception-class
 exception-action
```

where *exception-class* is one of:

```
{ [ANY] ERROR
| [ANY]  SQLERROR
|  NOT FOUND
|  WARNING
}
```

and *exception-action* is one of:

```
{ CONTINUE
| STOP
| CALL function
| RAISE
| GOTO label
}
```

1. *function* can be any function name defined in the program.
2. *label* must be a label defined in the current program block (main, function or report routine).

Usage

The WHENEVER instruction defines the exception handling by associating an *exception class* with an *exception action*.

> **Important:** The scope of a WHENEVER instruction is similar to a C preprocessor macro: It is local to the module defines the error handling for the rest of the module, unless a new WHENEVER instruction is encountered by the compiler, or a TRY/CATCH block is used.

This code example shows a typical WHENEVER instruction usage:

```
WHENEVER ERROR CONTINUE
DROP TABLE mytable -- SQL error will be ignored
CREATE TABLE mytable ( k INT, c VARCHAR(20) )
WHENEVER ERROR STOP
IF SQLCA.SQLCODE != 0 THEN
   ERROR "Could not create the table..."
```

```
END IF
```

Exception classes `ERROR` and `SQLERROR` are synonyms (compatibility issue). The previous example could have used `WHENEVER SQLERROR` instead of `WHENEVER ERROR`.

Actions for classes `ERROR`, `WARNING` and `NOT FOUND` can be set independently:

```
WHENEVER ERROR STOP
WHENEVER WARNING CONTINUE
WHENEVER NOT FOUND GOTO not_found_handler
...
```

When using the `WHENEVER ... CALL function` instruction, the program flow will go to the specified function and the return to the code block where the exception occured:

```
MAIN
    DEFINE x INTEGER
    WHENEVER ANY ERROR CALL error_handler
    -- WHENEVER handler takes effect
    LET x = 1/0
    DISPLAY "Back in MAIN..."
END MAIN

FUNCTION error_handler()
    DISPLAY "error_handler: ", STATUS
END FUNCTION

-- output:

error_handler:       -1202
Back in MAIN...
```

> **Note:** In a `WHENEVER ... CALL` instruction, you do not handle to specify braces after the function name.

A TRY/CATCH blocks takes precedence over the last `WHENEVER` instruction, see the following example:

```
MAIN
    DEFINE x INTEGER
    WHENEVER ANY ERROR CONTINUE
    -- WHENEVER handler takes effect
    LET x = 1/0
    DISPLAY "WHENEVER: ", STATUS
    -- WHENEVER handler is hidden by TRY/CATCH block
    TRY
        LET x = 1/0
    CATCH
        DISPLAY "CATCH   : ", STATUS
    END TRY
    -- WHENEVER handler takes again effect
    CALL func()
END MAIN

FUNCTION func()
    DEFINE x INTEGER
    LET x = 1/0
    DISPLAY "WHENEVER: ", STATUS
END FUNCTION

-- Output:

WHENEVER:       -1202
CATCH   :       -1202
```

```
WHENEVER:       -1202
```

The RAISE option can be used to propagate exceptions to the caller, which typically traps the error in a TRY/CATCH block:

```
-- main.4gl
IMPORT FGL myutils
MAIN
    TRY
       -- Pass a NULL form name to get error -1110
       CALL mutils.open_form(NULL)
    CATCH
       DISPLAY "Error: ", status
    END TRY
END MAIN

-- myutils.4gl
FUNCTION open_form(fn)
    DEFINE fn STRING
    WHENEVER ERROR RAISE -- Propagate exceptions to caller
    OPEN FORM f1 FROM fn
END FUNCTION
```

Important: WHENEVER [ANY] ERROR RAISE is not supported in a REPORT routine.

TRY - CATCH block

Use TRY / CATCH blocks to trap runtime exceptions in a delimited code block.

Syntax:

```
TRY
   instruction
   [...]
CATCH
   instruction
   [...]
END TRY
```

Usage:

Any language instruction in the TRY block will be executed until an exception is thrown. After an exception the program execution continues in the CATCH block. If no CATCH block is provided, the execution continues after END TRY.

If no exception is raised by the statements between the TRY and CATCH keywords, the instructions in the CATCH section are ignored and the program flow continues after END TRY.

This code example shows a TRY block executing an SQL statement:

```
TRY
    SELECT COUNT(*) INTO num_cust FROM customers WHERE ord_date <= max_date
CATCH
    ERROR "Error caught during SQL statement execution:", SQLCA.SQLCODE
END TRY
```

A TRY block be compared with WHENEVER ANY ERROR GOTO. Here is the equivalent of the previous code example:

```
WHENEVER ANY ERROR GOTO catch_error
```

```
    SELECT COUNT(*) INTO num_cust FROM customers WHERE ord_date <= max_date
    GOTO no_error
LABEL catch_error:
WHENEVER ERROR STOP
    ERROR "Error caught during SQL statement execution:", SQLCA.SQLCODE
LABEL no_error
```

The TRY statement can be nested in other TRY statements. In this example, the instruction in line #5 will be executed in case of SQL error:

```
TRY
    TRY
        SELECT COUNT(*) INTO num_cust FROM customers
    CATCH
        ERROR "Try block 2: ", SQLCA.SQLCODE
    END TRY
CATCH
    ERROR "Try block 1: ", SQLCA.SQLCODE
END TRY
```

The WHENEVER ERROR RAISE instruction can be used module-wide to define the behavior when an exception occurs in a function that is called from a TRY / CATCH block. If an exception occurs in a statement after the WHENEVER ERROR RAISE instruction, the program flow returns from the function and raises the exception as if it had occurred in the code of the caller. If the exception in thrown in the MAIN block, the program stops because the exception cannot be processed by a caller. In this example, the instruction in line #5 will be executed if an exception occurs in the *cust_report()* function:

```
MAIN
    TRY
        CALL cust_report()
    CATCH
        ERROR "An error occurred during report execution: ", STATUS
    END TRY
END MAIN

FUNCTION cust_report()
    WHENEVER ERROR RAISE
    START REPORT cust_rep ...
    ...
END FUNCTION
```

> **Important:** It is not possible to set a debugger break point at TRY, CATCH or END TRY: The TRY statement is a pseudo statement, the compiler does not generate p-code for this statement.

Tracing exceptions

Exception can be logged in a file when using the STARTLOG( ) function.

Exceptions will be automatically logged in a file, if all the following conditions are true:

- The STARTLOG function has been previously called to specify the name of the exception logging file.
- The exception action is set to CALL, GOTO or STOP. Exceptions are not logged when the action is CONTINUE or RAISE.
- The exception class is an ERROR, ANY ERROR or WARNING. NOT FOUND exceptions cannot be logged.

In other words, errors will not be logged in the case of WHENEVER { [ANY] ERROR | WARNING } CONTINUE, or when controlled by a TRY/CATCH block.

Each log entry contains:

- The system-time
- The location of the related instruction (source-file, line)

- The error-number
- The text of the error message, giving human-readable details for the exception

Default exception handling

By default, `WHENEVER ANY ERROR` action is to `CONTINUE` the program flow.

You can force the runtime system to execute the action defined with `WHENEVER ERROR` exception class with the following FGLPROFILE entry:

```
fglrun.mapAnyErrorToError = true
```

When this entry is set to true, `ET_EXPRESSION` expression errors such as a division by zero will be trapped and execute the action defined by the last `WHENEVER ERROR` instruction, the default being `STOP` the program with error display.

```
-- FGLPROFILE env var is defined to file with:
--    fglrun.mapAnyErrorToError = true

MAIN
  DEFINE x INT
  WHENEVER ERROR CALL my_error_handler
  LET x = 1 / 0   -- error handler will be called here
  DISPLAY "It continues...."
END MAIN

FUNCTION my_error_handler()
  DISPLAY "Handler: ", STATUS
END FUNCTION
```

Non-trappable errors

Some specific Genero runtime errors are not trappable.

If a non-trappable error occurs, neither WHENEVER instructions, nor TRY/CATCH blocks can trap the error: The runtime system will display the error message to the standard error stream, file an error log record if STARTLOG was previously called, and the program is stopped.

Non-trappable errors are typically fatal errors that generally deny further program execution. For example, the errors -1320, -1328 cannot be trapped.

Examples

Example 1: Defining a error handler function

This code example defines a WHENEVER ERROR handler function called `my_error_handler`. After connecting to the database, a `SELECT` statements tries to fetch a row from a table that does not exist, and raises SQL error -217 when connected to Informix®:

```
MAIN
  WHENEVER ERROR CALL my_error_handler
  DATABASE stores
  SELECT dummy FROM systables WHERE tabid=1
END MAIN

FUNCTION my_error_handler()
  DISPLAY "Error:", STATUS
  EXIT PROGRAM 1
END FUNCTION
```

Program output:

```
Error:      -217
```

Example 2: SQL error handling with WHENEVER

This code shows a typical SQL error handling block. It uses `WHENEVER ERROR CONTINUE` before executing SQL statements, tests the `SQLCA.SQLCODE` register for errors after each SQL instruction, and resets the default exception handler with `WHENEVER ERROR STOP` after the set of SQL commands to be controlled:

```
MAIN
  DEFINE
      tabname VARCHAR(50),
      sqlstmt STRING,
      rowcount INTEGER

  # In the DATABASE statement, no error should occur...
  DATABASE stores

  # But next SELECT may fail, if the user enters an invalid table name.
  WHENEVER ERROR CONTINUE
  PROMPT "Enter a table name:" FOR tabname
  LET sqlstmt = "SELECT COUNT(*) FROM " || tabname
  PREPARE s FROM sqlstmt
  IF sqlca.sqlcode THEN
     DISPLAY "SQL Error occurred:", sqlca.sqlcode
     EXIT PROGRAM 1
  END IF
  EXECUTE s INTO rowcount
  IF sqlca.sqlcode THEN
     DISPLAY "SQL Error occurred:", sqlca.sqlcode
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP

  ... (more instructions, stopping the program in case of error)

END MAIN
```

Program output in case of invalid table name:

```
SQL Error occurred:      -217
```

Example 3: Typical TRY / CATCH block

This example uses a `TRY/CATCH` block to trap errors. In this case, we try to connect to an invalid database, which will raise an SQL error and make the program flow go to the line after the `CATCH` statement:

```
MAIN
    TRY
        DATABASE invalid_database_name
        DISPLAY "Will not be displayed"
    CATCH
        DISPLAY "Exception caught, SQL error: ", SQLCA.SQLCODE
    END TRY
END MAIN
```

Program output (with Informix®):

```
Exception caught, SQL error:        -329
```

Example 4: TRY / CATCH in conjuction with WHENEVER

This code illustrates the fact that a `TRY/CATCH` block can be used in conjunction with a `WHENEVER` instruction: The program first executes a `WHENEVER ANY ERROR` to define an error handler named *foo* and later it uses a `TRY/CATCH` block to trap expression errors. In this example, we intentionally force a division by zero. After the `TRY/CATCH` block, we force another division by zero error, which will call the *foo* error handler:

```
MAIN
  DEFINE i INTEGER
  WHENEVER ANY ERROR CALL foo
  TRY
    DISPLAY "Next exception should be handled by the catch statement"
    LET i = i / 0
  CATCH
    DISPLAY "Exception caught, status: ", STATUS
  END TRY
  -- Previous error handler is restored after the TRY - CATCH block
  LET status = 0
  DISPLAY "Next exception should be handled by the foo function"
  LET i = i / 0
END MAIN

FUNCTION foo()
  DISPLAY "Function foo called, status: ", STATUS
END FUNCTION
```

Program output:

```
Next exception should be handled by the catch statement
Exception caught, status:      -1202
Next exception should be handled by the foo function
Function foo called, status:      -1202
```

Example 5: WHENEVER RAISE expection propagation

This example shows the usage of `WHENEVER ... RAISE` to propagate a potential exception to the caller. First the program defines the *foo* function as exception handler with `WHENEVER ANY ERROR CALL foo`, then it calls the *do_exception* function, which instructs the runtime system to propagate a potential error to the caller. As result, the division by zero in line #13 will be caught by the error handler defined in the `MAIN` block and call the *foo* function:

```
MAIN
    DEFINE i INTEGER
    WHENEVER ANY ERROR CALL foo
    DISPLAY "Next function call will generate an exception"
    DISPLAY do_exception(100, 0)
    WHENEVER ANY ERROR STOP -- reset default handler for rest of program
    ...
END MAIN

FUNCTION do_exception(a, b)
    DEFINE a, b INTEGER
    WHENEVER ANY ERROR RAISE
    RETURN a / b
END FUNCTION
```

```
FUNCTION foo()
    DISPLAY "Exception caught, status: ", STATUS
END FUNCTION
```

Program output:

```
Next function call will generate an exception
Exception caught, status:    -1202
```

OOP support

Describes Object Oriented Programming basics in the language.

- Understanding classes and objects on page 349
- DEFINE ... package.class on page 349
- Distinguish class and object methods on page 349
- Working with objects on page 350
- What class packages exist? on page 351

Understanding classes and objects

The Genero language supports basic Object Oriented Programming (OOP) concepts.

Classes are grouped into packages which are: a) build in and directly usable, b) available as libraries which need to be imported with the `IMPORT` instruction.

It is not possible to define classes with the language.

DEFINE ... *package.class*

Object reference variables allow to manipulate class instances.

Syntax:

```
DEFINE object package.class
```

1. *object* is the variable that references the object.
2. *package* is the name of the package the class comes from.
3. *class* is the name of the class.

Distinguish class and object methods

Class methods can be invoked from the class, while object methods can only be invoked from the variable referencing the object.

Methods can be invoked like regular functions, by passing parameters and/or returning values, and can be used in expressions when they return a scalar value.

Class methods

Class methods are called by using the class identifier as prefix, with the period as separator. The class identifier includes the package name and class name.

```
package.classname.method( parameter [,...] )
```

For example, to call the `refresh()` method of the `Interface` class, which is part of the `ui` package:

```
CALL ui.Interface.refresh()
```

Object methods

Object methods are called through the variable referencing the object. To use object methods, the object must exist. Call the object methods by using the object variable as a prefix, with a period as the separator.

```
object.method( parameter [,...] )
```

For example, to call the `setFieldActive()` method of an object of the `Dialog` class, which is part of the `ui` package:

```
DEFINE d ui.Dialog
LET d = ui.Dialog.getCurrent()
CALL d.setFieldActive("cust_addr", FALSE)
```

Working with objects

This topic introduces to basic object usage in Genero BDL.

In order to use an object in your program:

1. define an object variable using the class identifier.
2. instantiate the object; this is usually done by invoking a class method.
3. call object methods to manipulate the created object.

```
DEFINE n  om.DomDocument, b DomNode
LET n = om.DomDocument.create("Stock")
LET b = n.getDocumentElement()
```

The object *n* is instantiated using the `create()` class method of the DomDocument class. The object *b* is instantiated using the `getDocumentElement()` object method of the DomDocument class. This method returns the DomNode object that is the root node of the DomDocument object *n*.

The object variable only contains the reference to the object. For example, when passed to a function, only the reference to the object is copied onto the stack.

You do not have to destroy objects. This is done automatically by the runtime system for you, based on a reference counter.

```
MAIN
  DEFINE d  om.DomDocument
  LET d = om.DomDocument.create("Stock")  -- Reference counter = 1
END MAIN  -- d is removed, reference counter = 0 => object is destroyed.
```

You can pass object variables to functions or return them from functions. Objects are passed by reference to functions. In this example, the function creates the object and returns its reference on the stack:

```
FUNCTION createStockDomDocument()
  DEFINE d  om.DomDocument
  LET d = om.DomDocument.create("Stock")  -- Reference counter = 1
  RETURN d
END FUNCTION  -- Reference counter is still 1 because d is on the stack
```

Another part of the program can get the result of that function and pass it as a parameter to another function.

Example

```
MAIN
  DEFINE x  om.DomDocument
  LET x = createStockDomDocument()
  CALL writeStockDomDocument( x )
```

```
END MAIN

FUNCTION createStockDomDocument()
  DEFINE d  om.DomDocument
  LET d = om.DomDocument.create("Stock")
  RETURN d
END FUNCTION

FUNCTION writeStockDomDocument( d )
  DEFINE d  om.DomDocument
  DEFINE r om.DomNode
  LET r = d.getDocumentElement()
  CALL r.writeXml("Stock.xml")
END FUNCTION
```

What class packages exist?

A set of utility packages including useful classes are part of the distribution.

Built-in packages such as `ui`, `om` and `base`, are part of the runtime system and can be referenced directly. Extension packages such as `util`, `os`, `com` and `xml` need to be loaded explicitly with the `IMPORT` instruction, at the beginning of program modules.

Recent versions of the language support Java™ classes. Note however that using Java™ will create a Java™ Virtual Machine (JVM) that will be part of the runtime system process.

XML support

Introduces to DOM and SAX standards and describes the XML utility classes built-in the language.

These classes are useful to perform basic XML processing and manipulate the abstract user interface tree.

Use the full-featured XML classes provided in the web services extension for other needs.

- DOM and SAX standards on page 351
- DOM and SAX built-in classes on page 351
- Limitations of XML built-in classes on page 352
- Exception handling with XML classes on page 352
- Controlling the user interface with XML classes on page 352

DOM and SAX standards

DOM and SAX are both programming interfaces that can work with XML.

The *DOM* (Document Object Model) is a programming interface specification being developed by the World Wide Web Consortium (W3C) that lets a programmer create and modify HTML pages and XML documents as full-fledged program objects. DOM is a full-fledged object-oriented, complex but complete API, providing methods to manipulate the full XML document as a whole. DOM is designed for small XML trees manipulation.

The *SAX* (Simple API for XML) is a programming interface for XML, simpler than DOM. SAX is event-driven, streamed-data based, and designed for large trees.

DOM and SAX built-in classes

The DOM and SAX APIs both contain a set of built-in classes.

The DOM API is composed of:

- The `om.DomDocument` class, that defines the interface to a DOM document. Instances of this class can be used to identify and manipulate an XML tree. *DomNode* object manipulation methods are provided by this class.
- The `om.DomNode` class, that defines the interface to an DOM node. Instances of this class can be used to identify and manipulate a branch of an XML tree. Child nodes and node attributes management methods are provided by this class.
- The `om.NodeList` class, to handle a list of DomNode objects.

The SAX API is composed of:

- The `om.SaxAttributes` class represents a set of element attributes. It is used with an `om.XmlReader` or an `om.XmlWriter` object.
- The `om.XmlReader` class, that is defined to read XML. The XML document processing is based on SAX events.
- The `om.XmlWriter` class, that is defined to write XML. The XML document processing is based on SAX events.
- The `om.SaxDocumentHandler` class, which provides an interface to implement a SAX driver using functions defined in a .4gl module loaded dynamically.

Limitations of XML built-in classes

The built-in XML classes are provided for convenience, to help you manipulate XML content easily without loading a complete external XML library such as Java™ XML classes or a C-based XML libraries.

The features of these built-in classes are limited to basic XML usage. For example, there is no DTD / XML Schema validation done; you can create the same attribute twice or set an invalid attribute value. You must take care to follow the definition of the XML document when using these classes.

For a complete XML support, use the full-featured XML classes provided in the web services extension.

Exception handling with XML classes

Errors can occur while using XML built-in classes.

For example, calling methods of a SAX handler in an invalid order raises the runtime error -8004.

By default, the program stops in case of exception. XML errors can be trapped with the `WHENEVER ERROR` or `TRY/CATCH` exception handlers of Genero. If an error occurs during a method call of an XML class, the runtime system sets the STATUS variable.

This code example shows the trapping of XML classes errors.

```
MAIN
  DEFINE w om.SaxDocumentHandler
  LET w = om.SaxDocumentHandler.createFileWriter("sample.xml")
  TRY
    CALL w.endDocument()
  CATCH
    DISPLAY "ERROR: ", STATUS
  END TRY
END
```

Controlling the user interface with XML classes

The runtime system represents the user interface of a program with a DOM tree. User interface elements can be manipulated with the DOM and SAX built-in classes.

However, you must pay attention when modifying the AUI tree directly through the use of these classes. Invalid node or attribute creation can lead to unpredictable results.

Globals

Global variables can be shared among all modules of a program.

- Understanding global blocks on page 353
- GLOBALS on page 353
- Rules for globals usage on page 353
- Database schema in globals on page 354
- Content of a globals file on page 354
- Examples on page 354

Understanding global blocks

Global symbols can be defined with the GLOBALS instruction

The `GLOBALS` instruction can be used to declare variables, constants and types for the whole program.

> **Important:** Defining global elements shared by all modules of a program is an old programming concept. To increase code re-usability and readability, avoid global elements in your programs. Use modular concepts instead, by defining `PUBLIC` variables, constants and types in modules that will be imported into other modules with the `IMPORT FGL` instruction.

GLOBALS

The GLOBALS / END GLOBALS block and the GLOBALS instruction.

Syntax 1: Global block declaration

```
GLOBALS
  declaration-statement
  [,...]
END GLOBALS
```

1. *declaration-statement* is a variable, constant or type declaration.

Syntax 2: Importing definitions from a globals file

```
GLOBALS "filename"
```

1. *filename* is the name of a file containing the definition of globals.
2. Use this syntax to include global declarations in the current module.

Rules for globals usage

Follow the rules described in this topic in order to use globals properly.

In order to extend the scope of variables, constants or user types to the whole program, define a module containing a `GLOBALS ... END GLOBALS` block and including this global module with the `GLOBALS "filename"` statement in other modules.

The *filename* must contain the `.4gl` suffix. It can be a relative or an absolute path. To specify a path, the slash (/) directory separator can be used for UNIX™ and Windows™ platforms.

If you modify the globals file, you must recompile all the modules that include the file.

If a local element has the same name as another variable that you declare in the `GLOBALS` statement, only the local variable is visible within its scope of reference.

You can declare several `GLOBALS .. END GLOBALS` blocks in the same module.

A `GLOBALS` file must not contain any executable statement.

Do not write a declaration statement outside a GLOBALS ... END GLOBALS block in a GLOBALS file.

You do not need to compile the source file containing the GLOBALS block. However, it is recommended to compile the globals file to detect errors.

You can declare several GLOBALS "*filename*" instructions in the same module.

Although you can include multiple GLOBALS ... END GLOBALS statements in the same application, do not declare the same identifier within more than oneGLOBALS declaration. Even if several declarations of a global elements defined in multiple places are identical, declaring any global element more than once can result in compilation errors or unpredictable runtime behavior.

A GLOBALS block can hold GLOBALS "*filename*" instructions. In such case, the specified files will be included recursively.

Using global elements is not recommended, prefer to export module elements with the PUBLIC keyword, and include the module into other modules with the IMPORT FGL instruction.

Database schema in globals

Globals files can define the database schema to be used by the compiler to resolve DEFINE ... LIKE statements.

The schema specification must appear before the GLOBALS keyword starting the globals block.

The schema specification is propagated to the modules including the globals file defining the database schema. These modules can use DEFINE ... LIKE without an explicit SCHEMA instruction.

Further, when using the DATABASE instruction instead of SCHEMA, if the module including the globals contains the MAIN block, the DATABASE specification of the globals file will be propagated and result in an implicit database connection at runtime.

Example

```
SCHEMA stores
GLOBALS
  DEFINE cust_rec LIKE customer.*
  ...
END GLOBALS
```

Content of a globals file

A globals file should only contain a GLOBALS ... END GLOBALS block.

Because the GLOBALS block can also be defined in regular modules, it is possible to include a source containing more than a GLOBALS block. When including such module, the sections before and after the GLOBALS block are ignored by the compiler. The source defining the global elements can be compiled individually.

For example, it is legal to define a module A with a GLOBALS ... END GLOBALS block, followed by function definitions. This module can be compiled and functions will be taken into account. Module A can then be included in module B with a GLOBALS "*filename*" instruction, and when compiling module B the function definitions of the included module A will be ignored. IMPORT instructions before the a GLOBALS ... END GLOBALS block will also be ignored in such case.

Examples

Example 1: Multiple GLOBALS file

Module "labels.4gl": This module defines the text that should be displayed on the screen

```
GLOBALS
```

```
  CONSTANT g_lbl_val = "Index:"
  CONSTANT g_lbl_idx = "Value:"
END GLOBALS
```

Module "globals.4gl": Declares a global array and a constant containing its size

```
GLOBALS "labels.4gl" -- this statement could be line 2 of main.4gl

GLOBALS
  DEFINE g_idx ARRAY[100] OF CHAR(10)
  CONSTANT g_idxsize = 100
END GLOBALS
```

Module "database.4gl": This module could be dedicated to database access

```
GLOBALS "globals.4gl"

FUNCTION get_id()
  DEFINE li INTEGER
  FOR li = 1 TO g_idxsize -- this could be a FOREACH statement
      LET g_idx[li] = g_idxsize - li
  END FOR
END FUNCTION
```

Module "main.4gl": Fill in the global array and display the result

```
GLOBALS "globals.4gl"

MAIN
  DISPLAY "Initializing constant values for this application..."
  DISPLAY "Filling the data from function get_idx in module database.4gl..."
  CALL get_id()
  DISPLAY "Retrieving a few values from g_idx"
  CALL display_data()
END MAIN

FUNCTION display_data()
  DEFINE li INTEGER
  LET li = 1
  WHILE li <= 10 AND li <= g_idxsize
      DISPLAY g_lbl_idx CLIPPED || li || " " || g_lbl_val CLIPPED ||
 g_idx[li]
      LET li = li + 1
  END WHILE
END FUNCTION
```

Database schema

Defines database table structures with column type information to be reused in program variable definitions.

- Understanding database schemas on page 356
- SCHEMA on page 356
- Structure of database schema files on page 357
- Database schema extractor options on page 364

Understanding database schemas

Database schemas hold the definition of the database tables and columns.

In program sources or form specification files, you must specify the database schema file with the SCHEMA instruction. When the database schema is specified, you can define program variables by referencing the database table or column name. The program variables will get the type of the database column as defined in the schema file.

> **Note:** To improve compilation time, the `fglcomp` compiler will automatically generate a `.42d` index file from the `.sch` schema file, in the same directory as the `.sch` file. When the `.sch` file changes, the `.42d` index file is re-generated, and can be safely removed, if you want to cleanup your projet.

The FGLDBPATH environment variable can be used to define a list of directories where the compiler can find database schema files.

The schema files contain the column data types (.sch file), validation rules (.val file), and tty display attributes (.att) for columns.

> **Note:** The .val and .att files are supported for backward compatibility and should not be used in new developments.

The data types, display attributes, and validation rules are taken from the database schema files during compilation. Make sure that the schema files of the development database correspond to the production database, otherwise the elements defined in the compiled version of your modules and forms will not match the table structures of the production database.

Program variables can be defined with the LIKE keyword to get the data type defined in the schema files:

```
SCHEMA stores
MAIN
   DEFINE custrec RECORD LIKE customer.*
   DEFINE name LIKE customer.cust_name
   ...
END MAIN
```

The database schema files are generated with the fgldbsch tool from the system tables of an existing database.

> **Note:** It is strongly recommended that you regenerate the schema files when upgrading to a new compiler version. Bug fixes and new data type support can required schema file changes. If the schema file holds data type codes that are unknown to the current version, the compilers will raise the error -6634.

The fgldbsch must connect to the database server, with a db user allowed to query the database system tables (for example, INFORMATION_SCHEMA in a MySQL database).

> **Note:** For some type of databases, the table owner is mandatory to extract schema information. If you do not specify the -ow option in the comment line, fgldbsch will take the -un user name as default. If you do not use the -un/-up options because you are using indirect database connection with FGLPROFILE settings to identify the database user, or if the database user is authenticated by the operating system, the fgldbsch tool will try to identify the current database user after connection and use this name as table owner to extract the schema.

SCHEMA

Identifies the database schema files to be used for compilation.

Syntax 1

```
SCHEMA dbname
```

Syntax 2

```
[DESCRIBE] DATABASE dbname
```

1. *dbname* identifies the name of the database schema file to be used.

Usage

The SCHEMA *dbname* instruction defines the database schema files to be used for compilation, where *dbname* identifies the name of the database schema file to be used.

[DESCRIBE] DATABASE is supported for backward compatibility, use the SCHEMA instruction instead. The [DESCRIBE] DATABASE defines the compilation database schema and the default connection when the program starts, while SCHEMA defines only the compilation database schema.

The *dbname* database name must be expressed explicitly; it cannot be a variable as in a regular DATABASE instruction inside a program block.

Use the SCHEMA instruction outside any program block, before a variable declaration with DEFINE LIKE instructions. SCHEMA must precede any program block in each module that includes a DEFINE...LIKE declaration or INITIALIZE...LIKE and VALIDATE...LIKE statements. It must also precede any DEFINE...LIKE declaration of module variables.

Database schema information such as data types for DEFINE...LIKE are taken from the schema files **during compilation**. Make sure that the database schema file of the development database corresponds to the production database; otherwise the program variables defined in the p-code modules will not match the table structures of the production database.

The *dbname* can be written with different syntaxes:

```
  database
| database @ server
| "string"   -- for ex: "//server/database"
```

Such database specification is IBM® Informix® specific and should be avoided. Use simple database identifiers only, in lowercase.

When using a simple identifier for the database name, the compiler converts the name to lowercase before searching the schema file. However, if a double quoted string is used as database name, the name will be used as is to find the schema file.

With the SCHEMA instruction, the name of the database schema during development can be different from the name of the database source used at runtime.

> **Note:** To handle uppercase characters in the database name you must quote the name: SCHEMA "myDatabase"

Example

```
SCHEMA dev_db -- Compilation database schema
DEFINE rec RECORD LIKE customer.*
MAIN
   DATABASE prod_db -- Runtime database specification
   SELECT * INTO rec.* FROM customer WHERE custno=1
END MAIN
```

Structure of database schema files

A database schema is composed by three files (.sch, .val, .att)

- Column Definition File (.sch) on page 358

- Column Validation File (.val) on page 361
- Column Video Attributes File (.att) on page 363

Column Definition File (.sch)
The .sch database schema file contains the data types of database table columns.

Description

The data type of program variables or form fields used to hold data of a given database column must match the data type used in the database. The definition of these elements is simplified by centralizing the information in external .sch files, which contain column data types.

In form files, you can directly specify the table and column name in the field definition in the `ATTRIBUTES` section of forms.

In programs, you can define variables with the data type of a database column by using the `LIKE` keyword.

As column data types are extracted from the database system tables, you may get different results with different database servers. For example, Informix® provides the DATE data type to store simple dates in year, month, and day format (= `DATE` FGL type), while Oracle stores dates as year to second (= `DATETIME YEAR TO SECOND` FGL type).

The table describes the fields you will find in a row of the .sch file:

Table 128: Structure of the .sch file

Pos	Type	Description
1	STRING	Database table name.
2	STRING	Column name.
3	SMALLINT	Coded column data type. If the column is NOT NULL, you must add 256 to the value.
4	SMALLINT	Coded data type length.
5	SMALLINT	Ordinal position of the column in the table.

Next table shows the data types and their corresponding type code that can be present in a .sch schema file:

Table 129: Database Schema file (.sch) data type codes

Data type name	Data type code (field #3)	Data type length (field #4)
CHAR	0	Maximum number of characters or bytes (see note)
SMALLINT	1	Fixed length of 2
INTEGER	2	Fixed length of 4
FLOAT / DOUBLE PRECISION	3	Fixed length of 8
SMALLFLOAT / REAL	4	Fixed length of 4
DECIMAL	5	If the decimal is defined with a precision and scale, the length is computed using this formula:

Data type name	Data type code (field #3)	Data type length (field #4)

		`length = (precision * 256) + scale`
		If the decimal is defined as a floating point decimal (i.e. with no scale), the length is computed as follows:
		`length = (precision * 256) + 255`

SERIAL	6	Fixed length of 4
DATE	7	Fixed length of 4
MONEY	8	The length is computed using this formula: `length = (precision * 256) + scale` A MONEY cannot be defined with a floating point, is has always a scale.
Unused	9	*N/A*

DATETIME	10	For DATETIME types, the length is determined using the next formula:
		`length = ( digits * 256 ) + ( qual1 * 16 ) + qual2`
		where *digits* is the total number of digits used when displaying the datetime value. For example, a DATETIME YEAR TO MINUTE (YYYY-MM-DD hh:mm) uses 12 digits.
		The *qual1* and *qual2* elements identify datetime qualifiers according to this list:
		• 0 = YEAR
		• 2 = MONTH
		• 4 = DAY
		• 6 = HOUR
		• 8 = MINUTE
		• 10 = SECOND
		• 11 = FRACTION(1)
		• 12 = FRACTION(2)
		• 13 = FRACTION(3)
		• 14 = FRACTION(4)
		• 15 = FRACTION(5)
		For example, a DATETIME YEAR TO MINUTE size length is computed as follows:
		`(12 * 256) + (0 * 16) + 8 = 3080`

BYTE	11	Length of descriptor
TEXT	12	Length of descriptor
VARCHAR	13	Maximum number of characters or bytes (see note) If the length is positive: `length = ( min_space * 256 ) + max_size`

Data type name	Data type code (field #3)	Data type length (field #4)
		If length is negative: `length + 65536 = ( min_space * 256 ) + max_size`

INTERVAL	14	For INTERVAL types, the length is determined using the next formula:
		`length = ( digits * 256 ) + ( qual1 * 16 ) + qual2`
		where *digits* is the total number of digits used when displaying the interval value. For example, a INTERVAL HOUR(5) TO FRACTION(3) (hhhhh:mm:ss.fff) uses 12 digits.
		The *qual1* and *qual2* elements identify datetime qualifiers according to this list:
		• 0 = YEAR
		• 2 = MONTH
		• 4 = DAY
		• 6 = HOUR
		• 8 = MINUTE
		• 10 = SECOND
		• 11 = FRACTION(1)
		• 12 = FRACTION(2)
		• 13 = FRACTION(3)
		• 14 = FRACTION(4)
		• 15 = FRACTION(5)
		For example, a INTERVAL HOUR(5) TO FRACTION(3) size length is computed as follows:
		`(12 * 256) + (6 * 16) + 13 = 3181`

NCHAR	15	Maximum number of characters or bytes (see note)
NVARCHAR	16	Maximum number of characters or bytes (see note)
INT8	17	Fixed length of 10 (size of int8 structure) In programs, will be converted to a `BIGINT` type.
SERIAL8	18	Fixed length of 10 (size of int8 structure) In programs, will be converted to `BIGINT` type.
BOOLEAN (SQLBOOL)	45	Boolean type, in the meaning of Informix® front-end SQLBOOL (sqltype.h)
BIGINT	52	Fixed length of 8 (bytes)
BIGSERIAL	53	Fixed length of 8 (bytes)
VARCHAR2	201	Maximum number of characters In programs, will be converted to a `VARCHAR` type.

Data type name	Data type code (field #3)	Data type length (field #4)
NVARCHAR2	202	Maximum number of characters In programs, will be converted to a VARCHAR type.

Note: Data type length (field #4) is a SMALLINT value encoding the length or composite length of the type. For character string types, the unit of the length used to define character program variables and form fields depends on the length semantics.

Informix® SERIAL types

When the database schema defines SERIAL, BIGSERIAL or SERIAL8 types, form fields referencing the serial column with get the `NOENTRY` attribute automatically, except if defined with the `TYPE LIKE` syntax.

Informix® DISTINCT types

Informix® IDS version 9.x and higher allow you to define DISTINCT types from a base types with the CREATE DISTINCT TYPE instruction. In the syscolumns table, Informix® identifies distinct types in the coltype column by adding the 0x0800 bit (2048) to the base type code. For example, a distinct type defined with the VARCHAR built-in type (i.e. code 13) will be identified with the code 2061 (13 + 2048). Informix® sets additional bits when the distinct type is based on the LVARCHAR or BOOLEAN opaque types: If the base type is an LVARCHAR, the type code used in coltype gets the 0x2000 bit set (8192) and when the base type is BOOLEAN, the type code gets the 0x4000 bit (16384).

When extracting a schema from an Informix® database defining columns with DISTINCT types, the schema extractor will keep the original type code of the distinct type in the .sch file for columns using distinct types based on built-in types (with the 0x0800 bit set). Regarding the exception of opaque types, BOOLEAN-based distinct types get the code 45 (+ 256 if NOT NULL), and LVARCHAR-based distinct types are mapped to the code 201 (+ 256 if NOT NULL) if the -cv option enables conversion from LVARCHAR to VARCHAR2.

The fglcomp and fglform compilers understand the distinct type code bit 0x0800, so you can define program variables with a DEFINE LIKE instruction based on a column that was created with a distinct Informix® type.

Example

```
customer^customer_num^258^4^1^
customer^customer_name^256^50^2^
customer^customer_address^0^100^3^
order^order_num^258^4^1^
order^order_custnum^258^4^2^
order^order_date^263^4^3^
order^order_total^261^1538^4^
```

Column Validation File (.val)

The .val database schema file holds functional and display attributes of database table columns.

Description

The .val file holds default attributes and validation rules for database columns.

Important: The form field attribute definition in the .val file is supported for backward compatibility. Do not use this feature in new developments.

In form files, the attributes are taken from the .val file as defaults if the corresponding attribute is not explicitly specified in the field definition of the `ATTRIBUTES` section. The attributes in the .val file can be considered as a default configuration for a form field.

In programs, you can validate variable values in accordance with the `INCLUDE` attribute by using the `VALIDATE` instruction.

The .val file can be generated by fgldbsch from the IBM® Informix® specific `syscolval` table, or can be edited by an external column attributes editor.

This table describes the structure of the .val file:

Table 130: Structure of the .val file

Pos	Type	Description
1	STRING	Database table name.
2	STRING	Column name.
3	STRING	Column property name.
4	STRING	Column property value.

The supported attribute definitions are:

Table 131: Supported attribute definitions of the .val file

Attribute Name	Description
`AUTONEXT`	Defines the `AUTONEXT` attribute. When this attribute is defined, value is `YES`.
`CENTURY`	Defines the `CENTURY` attribute. The value must be one of: `R`, `C`, `F`, or `P`.
`COLOR`	Defines the `COLOR` attribute. The value is a color identifier (RED, GREEN, BLUE, ...)
`COMMENTS`	Defines the `COMMENTS` attribute. The value is a quoted string or Localized String (`%"xxx"`).
`DEFAULT`	Defines the `DEFAULT` attribute. Number, quoted string or identifier (`TODAY`).
`FORMAT`	Defines the `FORMAT` attribute.

Attribute Name	Description
	The value is a quoted string.
INCLUDE	Defines an include list as the INCLUDE attribute. Value must be a list: (*value* [,...]), where *value* can be a number, quoted string or identifier (TODAY).
INVISIBLE	Defines the INVISIBLE attribute. When this attribute is defined, value is YES.
JUSTIFY	Defines the JUSTIFY attribute. The value must be one of: LEFT, CENTER or RIGHT.
PICTURE	Defines the PICTURE attribute. The value is a quoted string.
SHIFT	Corresponds to the UPSHIFT and DOWNSHIFT attributes. Values can be UP or DOWN.
VERIFY	Defines the VERIFY attribute. When this attribute is defined, value is YES.

Example

```
customer^customer_name^SHIFT^UP^
customer^customer_name^COMMENTS^"Name of the customer"^
order^order_date^DEFAULT^TODAY^
order^order_date^COMMENTS^"Creation date of the order"^
```

Column Video Attributes File (.att)

The .att database schema file contains the default video attributes of database table columns.

Description:

The **.att** file is generated by fgldbsch from the IBM® Informix® specific syscolatt table.

> **Important:** The form field video attributes definition in the .att file is supported for backward compatibility. Do not use this feature in new developments.

This table describes the structure of the .att file:

Table 132: Structure of the .att file

Pos	Type	Description
1	STRING	Database table name.
2	STRING	Column name.
3	SMALLINT	Ordinal number of the attribute record.
4	STRING	COLOR attribute (coded).
5	CHAR(1)	INVERSE attribute (y/n).
6	CHAR(1)	UNDERLINE attribute (y/n).
7	CHAR(1)	BLINK attribute (y/n).
8	CHAR(1)	LEFT attribute (y/n).
9	STRING	FORMAT attribute.
10	STRING	Condition.

Database schema extractor options

The fgldbsch tool extracts the schema description for an existing database.

Schema information is extracted from the database catalog tables. fgldbsch detects the type of the database server after connection and queries the appropriated system catalog tables.

The database system must be available and the database client environment must be set properly in order to connect to the database engine and generate the schema files.

Generate the database schema files in the directory when the source code resided or in one of the directories listed in the FGLDBPATH environment variable.

- Specifying the database source on page 365
- Specifying the database driver on page 365
- Passing database user login and password on page 365
- Data type conversion control on page 365
- Specifying the table owner on page 366
- Force extraction of system tables on page 366
- Specifying the output file name on page 366
- Extracting definition of a single table on page 366
- Controlling the character case on page 366
- Using the verbose mode on page 367

- IBM Informix synonym tables on page 367
- IBM Informix shadow columns on page 367
- Running schema extractor in old mode on page 367

Specifying the database source

Run fgldbsch with the `-db dbname` option to identify the database source to which to connect. The `dbname` and related options can be present in the FGLPROFILE file. Otherwise, related options have to be provided with the fgldbsch command.

```
fgldbsch -db test1
```

Specifying the database driver

The database driver can be specified with the `-dv dbdriver` option, if the default driver is not appropriate.

```
fgldbsch -db test1 -dv dbmora
```

Passing database user login and password

If the operating system user is not the database user, you can provide a database user name and password respectively with the `-un` and `-up` options.

```
fgldbsch -db test1 -un scott -up fourjs
```

Data type conversion control

The fglcomp and fglform compilers expect known language data types (FGL types) in the schema file. While most data types correspond to IBM® Informix® SQL data types, some databases (including Informix®) can use specific types that do not map to an FGL type. Therefore, data types in the schema file are generated from the system catalog tables according to some conversion rules.

Type conversion can be controlled with the `-cv` option. Each character position of the string passed by this option represents a line in the conversion table of the corresponding source database. Give a conversion code for each data type (for example: `-cv` AABAAAB).

When using X as conversion code, the columns using the corresponding data types will be ignored and not written to the **.sch** file. This is particularly useful in the case of auto-generated columns like SQL Server's *uniqueidentifier* data type, when using a `DEFAULT NEWID()` clause.

Run the tool with the `-ct` option to see all the data type conversion tables, or use the `-cx` *dbtype* option to display the conversion table for a given database type (*dbtype* must be ifx, ora, db2, msv, pgs, mys, ...).

```
fgldbsch -cx ifx
...
-------------------------------------------------------------------
Informix        Informix A          Informix B
-------------------------------------------------------------------
1 BOOLEAN       BOOLEAN (t=45)      CHAR(1)
2 INT8          INT8                DECIMAL(19,0)
3 SERIAL8       SERIAL8             DECIMAL(19,0)
4 LVARCHAR(m)   VARCHAR2(m)         VARCHAR2(m)
5 BIGINT        BIGINT              DECIMAL(19,0)
6 BIGSERIAL     BIGSERIAL           DECIMAL(19,0)
-------------------------------------------------------------------
(ns) = Not supported in 4gl.
...

fgldbsch -db test1 -cv BAAABB
```

```
                        123456
```

In the above example, the -cv option instructs fgldbsch to use the types of the "Informix® A" column for all original column types except for BOOLEAN, BIGINT and BIGSERIAL, which must be converted to a VARCHAR2(m) FGL type.

The IBM® Informix® LVARCHAR(m) type can be converted by default to a VARCHAR2(m) pseudo type (code 201), which will be identified as a VARCHAR(m) by compilers.

In schema files, VARCHAR2(m) (type code 201) is equivalent to VARCHAR(m) (type code 13), without the 255 bytes limitation of the original Informix® VARCHAR type.

Not all native data types can be converted to FGL types. For example, user-defined types or spatial types are not supported by the language. When a table column with such unsupported data type is found, fgldbsch stops and displays an error to bring the problem to your eyes. Use the -ie option of fgldbsch to ignore the database tables having columns with unsupported types. When this option is used, none of the table columns definition will be written to the schema file.

Specifying the table owner

With some databases, the owner of tables is mandatory to extract a schema, otherwise you could get multiple definitions of the same table in the **.sch** schema file if tables with the same name exist in different database user schemas. To prevent such mistakes, you can specify the schema owner with the -ow owner option. If this option is not used, fgldbsch will use the database login name passed with the -un user option. This is usually the case with SQL Server and Sybase, where the owner of tables is "dbo".

```
fgldbsch -db test1 -un scott -up fourjs -ow dbo
```

Force extraction of system tables

By default fgldbsch does not generate system table definitions. You may want to use the -st option to extract schema information of system tables.

```
fgldbsch -db test1 -st
```

Specifying the output file name

By default, the generated schema files get the name of the database source specified with the -db option. If needed, you can force the name of the schema file with the -of name option. Specify the output file name without the **.sch** extension. This name will also be used to generate the files containing column validation rules and column attributes (extracted from IBM® Informix® syscolval and syscolatt tables).

```
fgldbsch -db test1 -of myschema
```

Extracting definition of a single table

In some cases, you may just want to extract schema file of new created tables. You can achieve this by using the -tn tabname option, to extract schema information of a specific table.

```
fgldbsch -db test1 -tn customers
```

Controlling the character case

By default, table and column names are converted to lower case letters to enforce compatibility with IBM® Informix®. You can force lower case, upper case or case-sensitive generation by using the -cl, -cu or -cc options.

```
fgldbsch -db test1 -cc
```

As a general rule, it is strongly recommended to keep table and column names in lowercase, in all areas (including the objects created in the database entity).

Using the verbose mode

Use the `-v` option to get verbose output from fgldbsch:

```
fgldbsch -db test1 -v
```

Do not base other tools or development procedures on the output format, the output can change in later versions.

IBM® Informix® synonym tables

When using an IBM® Informix® database, fgldbsch extracts synonyms. By default, only PUBLIC synonyms are extracted to avoid duplicates in the .sch file when the same name is used by several synonyms by different owners.

If you want to extract PRIVATE synonyms, you must use the -ow option to specify the owner of the tables and synonyms.

IBM® Informix® shadow columns

Starting with IBM® Informix® IDS version 11.50.xC1, you can create shadow columns on tables by using DDL options such as ADD VERCOLS. These columns are visible in the system catalog tables and would be listed in the column descriptions of the .sch schema file. However, as shadow columns are not part of the SELECT * list, it is not expected to get these columns in the schema file.

By default, the fgldbsch tool will not extract shadow columns from an IBM® Informix® database. You can use the `-sc` option to force the extraction of shadow columns:

```
fgldbsch -db test1 -sc
```

Running schema extractor in old mode

The fgldbsch program can be executed in old mode by specifying the `-om` option as first parameter, followed by the database source. You can pass the `-c` and `-r` options after the database source:

```
fgldbsch -om test1 -c -r
```

Use this mode for IBM® Informix® databases only.

The `-c` option is equivalent to `-cv BBBBBBBBB` in the default mode: Columns defined with an SQL type that is not a native Genero type will be converted to an equivalent type (see -cv and -ct options for more details).

If the `-r` option is specified, the schema extractor will ignore columns defined with unsupported SQL types. Unsupported types have no equivalent FGL type to store and handle the value, such as BLOB or CLOB for example. Understand that unlike the `-ie` option, which skips the whole table definition, `-r` will exclude table columns with unsupported types, but the other columns defined with supported types will be written to the .sch file. Thus, a record declared with `DEFINE RECORD` *`rec`* `LIKE` *`table`*`.*` (from a partial schema definition of a table) cannot be used in a `SELECT * INTO` *`rec`*`.*` statement, because the number of columns in the database table is different from the record definition.

Note also that when using the old mode, fgldbsch will extract system catalog tables (informix.sys*) for IBM® Informix® databases.

Programs

Explains program structure basics and global instructions/registers.

- Structure of a program on page 368
- Structure of a module on page 368
- The MAIN block on page 370
- Importing modules on page 371
- Predefined constants on page 376
- Configuration options on page 378
- Program registers on page 387
- Program execution on page 390

Structure of a program

The structure of a program consists of `MAIN` and `FUNCTION` blocks defined in several modules.

The program starts from the `MAIN` block. The instruction blocks contain statements that are be executed by the runtime system in the order that they appear in the code. Program blocks cannot be nested, nor any program block divided among more than one source code module.

Some instructions can include other instructions. Such instructions are called *compound statements*. Every compound statement of the language supports the `END` *`statement`* keyword (where *statement* is the name of the compound statement), to mark the end of the compound statement construct within the source code module. Most compound statements also support the `EXIT` *`statement`* keywords, to transfer control of execution to the statement that follows the `END` *`statement`* keywords. By definition, every compound statement can contain at least one statement block, a group of one or more consecutive statements. In the syntax diagram of a compound statement, a statement block always includes this element.

Structure of a module

A module defines a set of program elements such as functions, report routines, types, constants and variables.

Syntax

The declaration order of elements defined in a program module is constrained. Define module elements in the following way:

```
[ compiler-options
| import-statement [...]
| schema-statement
| globals-inclusion
| constant-definition [...]
| type-definition [...]
| variable-definition [...]
]

[ MAIN-block ]

[ dialog-block
| function-block
| report-routine
    [...] ]
]
```

1. *compiler-options* are described in OPTIONS (Compilation) on page 378.
2. *import-statement* imports an external module, see Importing modules on page 371.

3. *schema-statement* defines a database schema for the compilation.
4. *globals-inclusion* includes a globals file.
5. *constant-definition* defines constants.
6. *type-definition* defines user types.
7. *variable-definition* defines variables.
8. *MAIN-block* declares the main block of the program.
9. *dialog-block* declares a declarative dialog.
10. *function-block* declares a function.
11. *report-routine* declares a report routine.

Usage

A module defines a set of program elements that can be used by other modules when defined as `PUBLIC`, or to be local to the current module when defined as `PRIVATE`. Program elements are user-defined types, variables, constants, functions, report routines, and declarative dialogs.

A module can import other modules with the `IMPORT FGL` instruction. A module can define functions, reports, module variables, constants and types, as well as declarative dialogs.

Program modules are written as .4gl source files and a compiled to .42m files. Compiled modules (.42m files) can be linked together to create a program. However, linking is supported for backward compatibility only. The preferred way is to define module dependencies with the `IMPORT FGL` instruction. For better code re-usability, module elements can be shared by each other with by qualifying module variables, constants, types and function with `PRIVATE` or `PUBLIC` keywords. `PUBLIC` module elements can be referenced in other modules.

Example

```
OPTIONS SHORT CIRCUIT
IMPORT FGL cust_data
SCHEMA stores

PRIVATE CONSTANT c_title = "Customer data form"
PUBLIC TYPE t_cust RECORD LIKE customer.*
PRIVATE DEFINE cust_arr DYNAMIC ARRAY OF t_cust

MAIN
    ...
END MAIN

DIALOG cust_dlg()
    INPUT BY NAME cust_rec.*
       ...
    END INPUT
END DIALOG

FUNCTION cust_display()
   ...
END FUNCTION

FUNCTION cust_input()
   ...
END FUNCTION

REPORT cust_rep(row)
   ...
END REPORT
```

The MAIN block

The `MAIN` block is the starting point of the program.

Syntax

```
MAIN
    [ define-statement
    | constant-statement
    | type-statement
    ]
    { [defer-statement]
    | fgl-statement
    | sql-statement
    }
    [...]
END MAIN
```

1. *define-statement* defines function arguments and local variables.
2. *constant-statement* can be used to declare local constants.
3. *type-statement* can be used to declare local user defined type.
4. *defer-statement* defines how to handle signals in the program.
5. *fgl-statement* is any instruction supported by the language.
6. *sql-statement* is any static SQL instruction supported by the language.

Usage

When the runtime system executes a program, after some initialization, it gives control to the `MAIN` program block.

The `MAIN` block typically consists of a set of interruption handling instructions, runtime configuration options, database connection and a call to the main function of the program.

Example

```
IMPORT cust_module

MAIN
    DEFINE uname, upswd STRING

    DEFER INTERRUPT
    DEFER QUIT

    OPTIONS FIELD ORDER FORM,
            INPUT WRAP,
            HELP FILE "myhelp"

    CALL get_login() RETURNING uname, upswd
    WHENEVER ERROR CONTINUE
    CONNECT TO "stores" USER uname USING upswd
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE < 0 THEN
       DISPLAY "Error: Could not connect to database."
       EXIT PROGRAM 1
    END IF

    CALL cust_module.customer_input()

END MAIN
```

Importing modules

Use the `IMPORT ...` instruction to import BDL, C or Java external modules in the current module.

The `IMPORT {JAVA|FGL}` instruction can be used to declare the usage of an external module. All (public) symbols of the external module can be referenced in the current module.

The `IMPORT {JAVA|FGL}` instruction must be the first instruction in the current module. If you specify this instruction after `DEFINE`, `CONSTANT` or `GLOBALS`, fglcomp will report a syntax error.

The `IMPORT {JAVA|FGL}` instruction can import a compiled Genero module, a Java™ class or a C extension library:

- `IMPORT FGL` *`modulename`*: Imports a Genero module implementing functions, reports, types and variables.
- `IMPORT JAVA` *`classname`*: Imports a Java™ class or class element.
- `IMPORT` *`libname`*: Imports a C extension implementing functions and variables.

  Note: The name specified after the `IMPORT FGL` or `IMPORT JAVA` instruction is case-sensitive: Program module (.4gl) or Java™ class must exactly match the file name. However, for backward compatibility, C extension library names are converted to lowercase by the compiler (therefore, we recommend you to use lowercase file names for C extensions). A character case mismatch will be detected on UNIX™ platforms, but not on Windows™ where the file system is not case-sensitive. Regarding the usage of imported symbols in the rest of the code (i.e. not the `IMPORT` instruction): C extensions and Genero symbols are case-insensitive, while Java™ symbols are case-sensitive.

IMPORT *C-Extension*

The `IMPORT` instruction imports c extension module elements to be used by the current module.

Syntax

```
IMPORT filename
```

1. *filename* is the identifier (without the file extension) of the C extension module to be imported.

Usage

Using `IMPORT` *`libname`* instructs the compiler and runtime system to use the *libname* C extension for the current module.

Important: At runtime, all imported C extension modules are loaded when the program starts.

The name of the module specified after the `IMPORT` keyword is converted to lowercase by the compiler. Therefore it is recommended to use lowercase file names only.

The C extension must exist as a shared library (.DLL or .so) and be loadable (environment variables must be set properly). C extension modules used with the `IMPORT` instruction do not have to be linked to fglrun: The runtime system loads dependent C extension modules dynamically.

The FGLLDPATH environment variable specifies the directories to search for the C extension modules. You may also have to setup the system environment properly (i.e. PATH on Windows™ and LD_LIBRARY_PATH on UNIX™) if the C extension library depends from other libraries.

By default, the runtime system tries to load a C extension module with the name `userextension`, if it exists. This simplifies the migration of existing C extensions; you just need to create a shared library named userextension.so (or userextension.dll on Windows™), and copy the file to one of the directories defined in FGLLDPATH.

IMPORT FGL *module*

The `IMPORT FGL` instruction imports module symbols.

Syntax

```
IMPORT FGL modulename
```

1. *modulename* is the identifier (without the file extension) of the module to be imported.

Usage

With `IMPORT FGL modulename`, the symbols of the named .42m module can be referenced in the current module.

> **Important:** At runtime, the imported modules are only loaded on demand, when the program flow reaches an instruction that uses an element of the imported module. For example, when calling a function or when assigning a (public) module variable of the imported module.

The name specified after the `IMPORT FGL` instruction is case-sensitive.

The imported module symbols that can be referenced are:

- Public functions
- Public constants
- Public types
- Public module variables

The `PRIVATE`/`PUBLIC` modifiers can be used to hide / publish symbols to other modules. Functions are by default public, for backward compatibility. The next example declares a module variable that can be used by other modules, and a private function to be used only locally:

```
PUBLIC DEFINE custlist DYNAMIC ARRAY OF RECORD
  id INT,
  name VARCHAR(50),
  address VARCHAR(200)
END RECORD
...
PRIVATE FUNCTION myfunction()
...
```

When importing modules with the `IMPORT FGL` instruction, you instruct the fglcomp compiler and fglrun runtime system to load/check the specified modules, and there is no longer a need to link programs or use libraries.

With `IMPORT FGL`, the compiler can check the number of parameters and returning values in functions calls, and the completion in source code editors is improved as it can suggest all imported symbols.

Imported modules should be compiled before compiling the importing module. The FGLLDPATH environment variable specifies the directories to search for the `.42m` modules used by `IMPORT FGL`.

However, if the 42m file of the imported module is not existing, or is older as the corresponding source file, fglcomp will automatically compile the imported module. To avoid implicit compilation of imported modules, use the `--implicit=none` option of fglcomp. If the .42m file exists but the .4gl source file cannot be found, fglcomp imports the .42m file as is.

> **Important:** Auto-compilation of imported modules is only supported if the imported module is in the current directory. Modules located in other directories and found width FGLLDPATH must alredy be compiled.

No circular references are allowed. For example when module A imports module B, which in turn imports module A, you cannot compile one of the modules because the 42m file of the imported module is needed. Thus fglcomp will give error -8403, indicating that the imported module cannot be found:

Module "mod_a.4gl":

```
IMPORT FGL module_b
FUNCTION func_a()
  CALL func_b()
END FUNCTION
```

Module "mod_b.4gl":

```
IMPORT FGL module_a
FUNCTION func_b()
  CALL func_a()
END FUNCTION
```

Traditional linking is still supported for backward compatibility. To ease migration from traditional linking to imported modules, you can mix `IMPORT FGL` usage with fgllink. By default, even when `IMPORT FGL` is used, fglcomp does not raise an error if a referenced function is not found in the imported modules. This is mandatory to compile the 42m file to be linked later with the module defining the missing function. Use the `-W implicit` or the `--resolve-calls` option to check for imported functions.

When the `-W implicit` option is used and at least one `IMPORT FGL` is defined in the module, fglcomp will print warning -8406 for any referenced function that cannot be found in the imported modules. This option is silently ignored if no `IMPORT FGL` is used in the module.

To enable full symbol resolution by the compiler, use the `--resolve-calls` option. This option will force the compiler to check all function symbols referenced in a module, and raise error -8406, if a symbol could not be found in the imported modules. This option is typically used in programs that are only using `IMPORT FGL` and do not longer use the link phase.

When migrating existing projects using traditional linking, after compiling all the .4gl sources, consider using the `--print-imports` option of fglrun to print the `IMPORT FGL` suggestions for all the modules specified in the command line. This option will try to resolve all symbols as during linking, but instead of producing a .42r program, it will list the import instructions to be added in each module, and thus avoid linking:

```
$ cat main.4gl
MAIN
    CALL func1()
END MAIN
$ cat mod1.4gl
FUNCTION func1()
   CALL func2()
END FUNCTION
$ cat mod2.4gl
FUNCTION func2()
   CALL func1()
END FUNCTION

$ fglrun --print-imports main.42m mod1.42m mod2.42m
-- in main.4gl
IMPORT FGL mod1

-- in mod1.4gl
IMPORT FGL mod2

-- in mod2.4gl
# Cyclic import: IMPORT FGL mod1
#   caused by CALL func1
```

If a symbol is defined twice with the same name in two different modules, the symbol must be qualified by the name of the module. This feature overcomes the traditional 4gl limitation requiring unique function

names within a program. In the next example, both imported modules define the same "init()" function, but this can be resolved by adding the module name followed by a dot before the function names:

```
IMPORT FGL orders
IMPORT FGL customers
MAIN
  CALL orders.init()
  CALL customers.init()
  ...
END MAIN
```

If a symbol is defined twice with the same name in the current and the imported module, an unqualified symbol will reference the current module symbol. The next example calls the "init()" function with and without a module qualifier, the second call will reference the local function:

```
IMPORT FGL orders
MAIN
  CALL orders.init()  -- orders module function
  CALL init()  -- local function
  ...
END MAIN
FUNCTION init()
  ...
END FUNCTION
```

Example

Module "account.4gl":

```
PRIVATE DEFINE current_account VARCHAR(20)

PUBLIC FUNCTION set_account(id)
  DEFINE id VARCHAR(20)
  LET current_account = id
END FUNCTION
...  -- File: myutils.4gl
PRIVATE DEFINE initialized BOOLEAN

PUBLIC TYPE t_prog_info RECORD
        name STRING,
        version STRING,
        author STRING
     END RECORD

PUBLIC FUNCTION init()
  LET initialized = TRUE
  ...
END FUNCTION

PUBLIC FUNCTION fini()
  LET initialized = FALSE
  ...
END FUNCTION
```

Module "myutils.4gl":

```
PRIVATE DEFINE initialized BOOLEAN

PUBLIC TYPE t_prog_info RECORD
        name STRING,
        version STRING,
```

```
        author STRING
    END RECORD

PUBLIC FUNCTION init()
  LET initialized = TRUE
  ...
END FUNCTION

PUBLIC FUNCTION fini()
  LET initialized = FALSE
  ...
END FUNCTION
```

Module "program.4gl":

```
IMPORT FGL myutils
IMPORT FGL account
DEFINE filename STRING
DEFINE proginfo t_prog_info  -- Type is defined in myutils
MAIN
  LET proginfo.name = "program"
  LET proginfo.version = "0.99"
  LET proginfo.author = "scott"
  CALL myutils.init()  -- with module prefix
  CALL set_account("CFX4559")  -- without module prefix
END MAIN
```

IMPORT JAVA *classname*

The `IMPORT JAVA` instruction imports Java™ module elements.

Syntax

```
IMPORT JAVA classname
```

1. *classname* is the identifier of the Java™ class to be imported.

Usage

Using `IMPORT JAVA classname`, you can import and use a Java™ class.

> **Important:** At runtime, the imported Java™ classes are only loaded on demand, when the program flow reaches an instruction that uses the class. For example, when the reaching the declaration of a variable defined to reference an object of a Java™ class.

The name specified after the `IMPORT JAVA` instruction is case-sensitive.

The CLASSPATH environment variable defines the directories for Java™ packages. See the Java™ documentation for more details.

Actually *classname* must be a path with package names separated by a dot, so the actual syntax for `IMPORT JAVA` is:

```
IMPORT JAVA [ packagename . [...] ] filename
```

It is allowed to write several `IMPORT JAVA` instruction with the same class; Compilation will succeed to mimic the Java™ import rules. However, you should avoid this:

```
IMPORT JAVA java.util.regex.Matcher
 IMPORT JAVA java.util.regex.Matcher
```

Predefined constants

The language defines a set of global constants that can be used in the programs.

- NULL on page 376
- TRUE on page 376
- FALSE on page 377
- NOTFOUND on page 377

NULL

The `NULL` constant is provided as the "nil" value.

Syntax

```
NULL
```

Usage

When comparing variables to `NULL`, use the `IS NULL` operator, not the equal operator.

If an element of an expression is null, the expression is evaluated to `NULL`.

Variables are initialized to `NULL` or to zero, according to their data type.

Empty character string literals (`""`) are equivalent to `NULL`.

`NULL` cannot be used with the = equal comparison operation, you must use `IS NULL`.

Example

```
MAIN
  DEFINE s CHAR(5)
  LET s = NULL
  DISPLAY "s IS NULL evaluates to:"
  IF s IS NULL THEN
    DISPLAY "TRUE"
  ELSE
    DISPLAY "FALSE"
  END IF
END MAIN
```

TRUE

`TRUE` is a predefined constant to be used in boolean expressions.

Syntax

```
TRUE
```

Usage

`TRUE` is a predefined constant that can be used as a boolean value in boolean expressions.

The `TRUE` constant is equal to 1 (one).

`TRUE` and `FALSE` are typically used as return values of functions that give a binary result.

Example

```
MAIN
```

```
  DEFINE short BOOLEAN
  LET short = is_short("abcdef")
  IF short THEN
     DISPLAY "String is short."
  END IF
END MAIN

FUNCTION is_short(s)
  DEFINE s STRING
  IF s.getLength() < 10 THEN
     RETURN TRUE
  ELSE
     RETURN FALSE
  END IF
END FUNCTION
```

FALSE
FALSE is a predefined constant to be used in boolean expressions.

Syntax

```
FALSE
```

Usage

FALSE is a predefined constant that can be used as a boolean value in boolean expressions.

The FALSE constant is equal to 0 (zero).

TRUE and FALSE are typically used as return values of functions that give a binary result.

Example

```
MAIN
  DEFINE odd BOOLEAN
  LET odd = is_odd(125763)
  IF odd THEN
     DISPLAY "Number is odd."
  END IF
END MAIN

FUNCTION is_odd(value)
  DEFINE value INTEGER
  IF value MOD 2 = 1 THEN
     RETURN TRUE
  ELSE
     RETURN FALSE
  END IF
END FUNCTION
```

NOTFOUND
NOTFOUND is a predefined constant used to check if an SQL statement returns rows.

Syntax

```
NOTFOUND
```

Usage

The NOTFOUND constant is used to test the execution status of an SQL statement returning a result, to check whether rows have been found.

The NOTFOUND constant is equal to 100.

You typically compare SQLCA.SQLCODE to NOTFOUND, after a SELECT statement execution.

Example

```
MAIN
  DATABASE stores
  SELECT tabid FROM systables WHERE tabid = 1
  IF SQLCA.SQLCODE = NOTFOUND THEN
    DISPLAY "No row was found"
  END IF
END MAIN
```

Configuration options

Compiler and runtime system can be controlled with several configuration settings.

- OPTIONS (Compilation) on page 378
- OPTIONS (Runtime) on page 379
- Runtime configuration in FGLPROFILE on page 386
- DEFER INTERRUPT / QUIT on page 387

OPTIONS (Compilation)

OPTIONS outside program blocks defines semantics of the language for the compiler.

Syntax

```
OPTIONS
{ SHORT CIRCUIT
} [,...]
```

Usage

The OPTIONS statement used before any MAIN, FUNCTION or REPORT program block defines language semantics options, that will take effect for the current module only. Unlike runtime options, compiler options cannot be changed during program execution.

The statement to define compiler options must be placed before the MAIN block in the main module, or before the first FUNCTION / REPORT block in other modules:

The OPTIONS compiler directive allows to control following features:

- Controlling semantics of AND / OR operators on page 379

Example

```
OPTIONS SHORT CIRCUIT
MAIN
    DISPLAY "Global Options example"
END MAIN
```

Controlling semantics of AND / OR operators

The `OPTIONS SHORT CIRCUIT` defines the semantics of `AND`/`OR` operators.

When using `OPTIONS SHORT CIRCUIT` at the beginning of a module, the runtime system will optimize the evaluation of boolean expressions involving `AND` and `OR` operators, by using the *short-circuit evaluation* method (also called *minimal evaluation* method). This behavior is enabled for the whole module.

By default, the behavior of `AND` and `OR` operators is to evaluate all operands on the left and right side of the operator. In fact this is not required: If the left operand of the `AND` evaluates to `FALSE`, there is no need to evaluate the right operand, because the result of the `AND` operator will be false, anyway. Similarly, when the left operand of an `OR` expression evaluates to `TRUE`, there is not need to evaluate the right operand, since the result of the boolean expression will be true, anyway.

This method can improve performances and simplify programming. However, existing code may rely on the fact that all parts of a boolean expression are evaluated, especially when calling functions that do some processing. By using the short-circuit evaluation method, it is unsure that the function used in the right operand of AND/OR will be called, because it depends on the result of the left operand.

By using short-circuit evaluation, it is possible to reference a dynamic array in the same boolean expression, after checking that the index is in the current array element range:

```
IF x<=arr.getLength() AND arr[x].order_date > TODAY THEN
    ...
END IF
```

With the default `AND` semantics, in this code, the right operand is also evaluated. If the `x` index is greater as the array length, new array elements will be automatically created in the expression on the right of the `AND` operator. To avoid this situation, you are forced to write following code, when `OPTIONS SHORT CIRCUIT` is not used:

```
IF x<=arr.getLength() THEN
  IF arr[x].order_date > TODAY THEN
     ...
  END IF
END IF
```

OPTIONS (Runtime)

The `OPTIONS` instruction inside program blocks controls program behavior at runtime.

Syntax

```
OPTIONS options-clause [,...]
```

Usage

Use the `OPTIONS` instruction inside a function block to control the behavior of the runtime system for rest of the program execution.

A program can execute successive `OPTIONS` statements at different places in the code.

The runtime `OPTIONS` statement allows to control following runtime features:

- Defining the position of reserved lines on page 380
- Defining default TTY attributes on page 381
- Defining field tabbing order on page 382
- Defining the field input loop on page 381
- Application termination on page 383
- Front-end termination on page 383
- Defining the message file on page 384

- Defining control keys on page 384
- Setting default screen modes for sub-programs on page 385
- Enabling/disabling SQL interruption on page 386

Defining the position of reserved lines

The `OPTIONS` *`element`* `LINE` defines position of dedicated screen lines.

Syntax

```
OPTIONS
{ MENU LINE line-value
| MESSAGE LINE line-value
| COMMENT LINE {OFF|line-value}
| PROMPT LINE line-value
| ERROR LINE line-value
| FORM LINE line-value
}
```

Usage

The `OPTIONS` statement can define the positions of reserved lines for menus, forms and messages. Reserved window lines are used in TUI mode. These options are not required in GUI mode, as most have no effect on the display, except when using the traditional mode, where program windows are rendered as in a dumb terminal.

- `COMMENT LINE` specifies the position of the comment line. The comment line displays messages defined with the `COMMENT` attribute in the form specification file. The default is (`LAST-1`) for the `SCREEN`, and `LAST` for all other windows. You can hide the comment line with `COMMENT LINE OFF`.
- `ERROR LINE` specifies the position on the screen of the error line that displays the text of the `ERROR` statement. The default is the `LAST` line of the `SCREEN` window.
- `FORM LINE` specifies the window line where forms are displayed. The default is (`FIRST+2`), or line 3 of the current window.
- `MENU LINE` specifies the position of the menu line. This line displays the menu name and options, as defined by the `MENU` statement. The default is the `FIRST` line in the window.
- `MESSAGE LINE` specifies the position of the message line. This reserved line displays the text of the `MESSAGE` statement. The default is (`FIRST+1`), or line 2 of the current window.
- `PROMPT LINE` specifies the position of the prompt line where the text of `PROMPT` statements is displayed. The default value is the `FIRST` line in the window.

You can specify any of the following positions for each reserved line:

Table 133: Reserved line expressions

Expression	Description
`FIRST`	The first line of the screen or window.
`FIRST + integer`	A relative line position from the first line.
`integer`	An absolute line position in the screen or window.
`LAST - integer`	A relative line position from the last line.
`LAST`	The last line of the screen or window.

Defining default TTY attributes

The `OPTIONS {INPUT|DISPLAY} ATTRIBUTES` defines default TTY attributes for dialogs and display statements.

Syntax

```
OPTIONS { INPUT | DISPLAY } ATTRIBUTES ({FORM|WINDOW|attributes)
```

Usage

`OPTIONS INPUT ATTRIBUTES` defines the default color and terminal effect attributes that will be used in subsequent dialog statement.

`OPTIONS DISPLAY ATTRIBUTES` defines the default attributes for display statements.

The display attributes are based on dumb terminal (i.e. TTY) possibilities, but will be rendered accordingly on GUI mode. Graphical front-ends can be configured to render TTY attributes is a specific way. Instead of TTY based attributes, consider using presentation styles in new developments.

Any display attribute defined by the `OPTIONS` statement remains in effect until the runtime system encounters a statement that redefines the same attribute. This can be another `OPTIONS` statement, or an `ATTRIBUTE` clause in one of the following statements:

- `CONSTRUCT`
- `INPUT`
- `DISPLAY`
- `DIALOG`
- `INPUT ARRAY`
- `DISPLAY ARRAY`
- `OPEN WINDOW`

The `ATTRIBUTE` clause in these statements only redefines the attributes temporarily. After the window closes or after the dialog statement terminates, the runtime system restores the attributes from the most recent `OPTIONS` statement.

The `FORM` keyword in `INPUT ATTRIBUTE` or `DISPLAY ATTRIBUTE` clauses instructs the runtime system to use the input or display attributes of the current form. Similarly, you can use the `WINDOW` keyword of the same clauses to instruct the program to use the input or display attributes of the current window. You cannot combine the `FORM` or `WINDOW` attributes with any other attributes.

This table shows the valid input and display attributes:

Table 134: Input and display attributes

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The TTY color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL`	The TTY font attribute of the displayed text.
`REVERSE, BLINK, UNDERLINE`	The TTY video attribute of the displayed text.

Defining the field input loop

The `OPTIONS INPUT [NO] WRAP` instructions defines field wrapping in dialogs.

Syntax

```
OPTIONS INPUT [NO] WRAP
```

Usage

By default, an interactive statement such as `CONSTRUCT` or `INPUT` terminates when the focus leaves the last field controlled by the dialog instruction.

The `OPTIONS INPUT WRAP` instruction can change this behavior, causing the cursor to move from the last field to the first, repeating the sequence of fields until the dialog is validated or canceled.

The `INPUT NO WRAP` option restores the default input loop behavior.

Example

```
MAIN
    OPTIONS INPUT WRAP
    ...
END MAIN
```

Defining field tabbing order

Syntax

```
OPTIONS FIELD ORDER { CONSTRAINED | UNCONSTRAINED | FORM }
```

Usage

Tabbing order is used in interactive instructions such as `INPUT`, `INPUT ARRAY` or `CONSTRUCT`, where individual fields can get the focus.

The `FIELD ORDER` runtime option defines the default behavior when moving from field to field with the TAB and SHIFT-TAB keys in GUI mode, and with the Up / Down arrow keys in TUI mode.

By default, the tabbing order is defined by the list of fields used by the program instruction. This corresponds to `FIELD ORDER CONSTRAINED` option, which is the default.

When using `FIELD ORDER UNCONSTRAINED` in TUI mode, the Up and Down arrow keys will move the cursor to the field above or below the current field, respectively. When using the default `FIELD ORDER CONSTRAINED` option, the Up and Down arrow keys move the cursor to the previous or next field, respectively. If `FIELD ORDER UNCONSTRAINED` is used, the `Dialog.fieldOrder` FGLPROFILE entry is ignored.

The `UNCONSTRAINED` option can only be supported in TUI mode, with a simple form layout. It is not recommended to use this option in sGUI mode.

The `FIELD ORDER FORM` option instructs interactive instructions to use the tabbing order defined by the `TABINDEX` attributes of the current form fields. With this option, tabbing order can be defined in the layout of the form, independently from the program instruction. This is the preferred way in GUI mode. When `FIELD ORDER FORM` is used, the `Dialog.fieldOrder` FGLPROFILE entry is ignored.

Example

Form "form1.per":

```
LAYOUT
GRID
{
  First name:  [f001            ] Last name:   [f002
  ]
  Address:     [f003
  ]
}
END
```

```
END

ATTRIBUTES
EDIT f001 = FORMONLY.fname, TABINDEX = 2;
EDIT f002 = FORMONLY.lname, TABINDEX = 1;
EDIT f003 = FORMONLY.address, TABINDEX = 0;
END
```

Module "main.4gl":

```
MAIN
  DEFINE fname, lname CHAR(20), address CHAR(50)

  OPTIONS INPUT WRAP

  OPEN FORM f1 FROM "f1"
  DISPLAY FORM f1

  OPTIONS FIELD ORDER CONSTRAINED
  INPUT BY NAME fname, address, lname

  OPTIONS FIELD ORDER UNCONSTRAINED
  INPUT BY NAME fname, address, lname

  OPTIONS FIELD ORDER FORM
  INPUT BY NAME fname, address, lname

END MAIN
```

Application termination

The `OPTIONS TERMINATE SIGNAL` defines a callback function in case of SIGTERM signal.

Syntax

```
OPTIONS ON TERMINATE SIGNAL CALL function
```

Usage

The `OPTIONS ON TERMINATE SIGNAL CALL` *`function`* defines the function that must be called when the application receives the SIGTERM signal. With this option, you can control program termination. If this statement is not called, the program is stopped with an exit value of SIGTERM (15).

On Microsoft™ Windows™ platforms, the function will be called in the following cases:

- The console window that the program was started from is closed.
- The current user session is terminated (i.e. the user logs off).
- The system is shut down.

Use the `OPTIONS ON TERMINATE SIGNAL CALL` *`function`* instruction with care, and do not execute complex code in the callback function. The code should only contain simple and short cleanup operations; Any interactive instruction must be avoided.

Front-end termination

The `OPTIONS CLOSE APPLICATION` instruction defines the callback function in case of front-end termination.

Syntax

```
OPTIONS ON CLOSE APPLICATION CALL function
```

Usage

The `OPTIONS ON CLOSE APPLICATION CALL` *`function`* can be used to execute specific code when the front-end stops. For example, when the front-end program is stopped, when the user workstation session is ended, or when the workstation is shut down.

Before stopping, the front-end sends a internal event that is trapped by the runtime system. When a callback function is specified with this program option command, the application code that was executing is canceled, and the callback function is executed before the program stops.

Use the `OPTIONS ON CLOSE APPLICATION CALL` *`function`* instruction with care, and do not execute complex code in the callback function. The code should only contain simple and short cleanup operations; Any interactive instruction must be avoided.

A front-end program crash or network failure is not detected and cannot be handled by this instruction.

Defining the message file
The `OPTIONS HELP FILE` instruction defines the name of the message file.

Syntax

```
OPTIONS HELP FILE filename
```

Usage

The `OPTIONS HELP FILE` instruction specifies an expression that returns the filename of a help file. This filename can also include a pathname. Messages in this file can be referenced by number in form-related statements, and are displayed at runtime when the user presses the Help key.

By default, message files are searched in the current directory, then DBPATH / FGLRESOURCEPATH environment variable is scanned to find the file.

Defining control keys
The `OPTIONS` *`action`* `KEY` instruction defines physical keys for common dialog actions.

Syntax

```
OPTIONS
{ INSERT
| DELETE
| NEXT
| PREVIOUS
| ACCEPT
| HELP
} KEY key-name
```

Usage

This `OPTIONS` clause can specify physical keys to support logical key functions in the interactive instructions.

The physical key definition options are only provided for backward compatibility with the TUI mode. Use the action defaults configuration to define accelerator keys for actions.

Description of the keys:

- The `ACCEPT KEY` specifies the key that validates a `CONSTRUCT`, `INPUT`, `DIALOG`, `INPUT ARRAY`, or `DISPLAY ARRAY` statement.

  The default `ACCEPT KEY` is `ESCAPE`.
- The `DELETE KEY` specifies the key in `INPUT ARRAY` statements that deletes a screen record.

The default DELETE KEY is F2.

- The INSERT KEY specifies the key that opens a screen record for data entry in INPUT ARRAY.

  The default INSERT KEY is F1.
- The NEXT KEY specifies the key that scrolls to the next page of a program array of records in an INPUT ARRAY or DISPLAY ARRAY statement.

  The default NEXT KEY is F3.
- The PREVIOUS KEY specifies the key that scrolls to the previous page of program records in an INPUT ARRAY or DISPLAY ARRAY statement.

  The default PREVIOUS KEY is F4.
- The HELP KEY specifies the key to display help messages.

  The default HELP KEY is CONTROL-W.

You can specify the following keywords for the physical key names:

Table 135: Keywords for physical key names

Key Name	Description
ESC or ESCAPE	The ESC key (not recommended, use ACCEPT instead).
INTERRUPT	The interruption key (on UNIX™, interruption signal).
TAB	The TAB key (not recommended).
CONTROL-*char*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X
F1 through F255	A function key.
LEFT	The left arrow key.
RETURN or ENTER	The return key.
RIGHT	The right arrow key.
DOWN	The down arrow key.
UP	The up arrow key.
PREVIOUS or PREVPAGE	The previous page key.
NEXT or NEXTPAGE	The next page key.

You might not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many UNIX™ systems.

Setting default screen modes for sub-programs

The OPTIONS RUN IN instruction defines the TTY mode to run sub-programs.

Syntax

```
OPTIONS RUN IN {FORM|LINE} MODE
```

Usage

When using character terminals, the runtime system recognizes two screen display modes: line mode (`IN LINE MODE`) and formatted mode (`IN FORM MODE`). The `OPTIONS` and `RUN` statements can explicitly specify a screen mode. The `OPTIONS` statement can set separate defaults for these statements.

After `IN LINE MODE` is specified, the terminal is in the same state (in terms of stty options) as when the program began. This usually means that the terminal input is in cooked mode, with interruption enabled, and input not available until after a newline character has been typed.

The `IN FORM MODE` keywords specify raw mode, in which each character of input becomes available to the program as it is typed or read.

By default, a program operates in line mode, but so many statements take it into formatted mode (including `OPTIONS` statements that set keys, `DISPLAY`, `OPEN WINDOW`, `DISPLAY FORM`, and other screen interaction statements), that typical programs are actually in formatted mode most of the time.

When the `OPTIONS` statement specifies `RUN IN FORM MODE`, the program remains in formatted mode if it currently is in formatted mode, but it does not enter formatted mode if it is currently in line mode.

When the `OPTIONS` statement specifies `RUN IN LINE MODE`, the program remains in line mode if it is currently in line mode, and it switches to line mode if it is currently in formatted mode.

Enabling/disabling SQL interruption
The `OPTIONS SQL INTERRUPT` instruction enables or disables SQL statement interruption.

Syntax

```
OPTIONS SQL INTERRUPT { ON | OFF }
```

Usage

The `OPTIONS SQL INTERRUPT` instruction controls interruption event detection during the execution of long running SQL statements.

Pay attention to the fact that not all database servers support SQL interruption.

By default, SQL interruption is off.

Runtime configuration in FGLPROFILE

The behavior of the runtime system can be controlled with FGLPROFILE configuration parameters.

- Responding to CTRL_LOGOFF_EVENT on page 386

Responding to CTRL_LOGOFF_EVENT
FGLPROFILE `fglrun.ignoreLogoffEvent` controls program behavior in case of logoff events on Windows™ platforms.

Syntax

```
fglrun.ignoreLogoffEvent = true
```

Usage

On Windows™ platforms, when the user disconnects, the system sends a CTRL_LOGOFF_EVENT event to all console applications. When the runtime system receives this event, it stops immediately.

On a Windows™ Terminal Server, if an Administrator user closes his session, a CTRL_LOGOFF_EVENT is sent to all console applications started by ANY user connected to the machine (even if these applications were not started by the administrator).

To prevent the runtime system from stopping on a logoff event, you can use the `fglrun.ignoreLogoffEvent` entry in the FGLPROFILE configuration file. If this entry is set to `true`, the CTRL_LOGOFF_EVENT event is ignored by the runtime system.

As a result, when the administrator user disconnects on a Windows™ Terminal Server, programs started by remote users would not stop.

DEFER INTERRUPT / QUIT
The `DEFER` instruction defines the program behavior when *interruption* or *quit* signals are received.

Syntax

```
DEFER { INTERRUPT | QUIT }
```

Usage

The `DEFER` instruction controls the behavior of the program when an *interruption* or *quit* signal has been received.

`DEFER INTERRUPT` and `DEFER QUIT` instructions can only appear in the MAIN block.

`DEFER INTERRUPT` indicates that the program must continue when it receives an *interrupt* signal. By default, the program stops when it receives an *interrupt* signal.

Once deferred, you cannot reset to the default behavior.

When an *interrupt* signal is caught by the runtime system and `DEFER INTERRUPT` is used, the `INT_FLAG` global variable is set to true by the runtime system.

Interrupt signals are raised on terminal consoles when the user presses a key like CTRL-C, depending on the stty configuration. When a program is displayed through a front end, no terminal console is used; therefore, users cannot send interrupt signals with the CTRL-C key. To send an interruption request from the front end, you must define an 'interrupt' action view.

`DEFER QUIT` indicates that the program must continue when it receives a *quit* signal. By default, the program stops when it receives a *quit* signal.

When a *quit* signal is caught by the runtime system and `DEFER QUIT` is used, the `QUIT_FLAG` global variable is set to true by the runtime system.

Program registers

Predefined global registers can be used in programs to detect errors, signals and events.

- STATUS on page 387
- INT_FLAG on page 388
- QUIT_FLAG on page 389

STATUS
`STATUS` is a predefined variable that contains the execution status of the last instruction.

Syntax

```
STATUS
```

Usage

`STATUS` is a predefined variable that contains the execution status of the last program instruction.

`STATUS` allows to get diagnostic of procedural, interactive, and SQL instructions.

The data type of STATUS is INTEGER.

> **Note:** While STATUS can be modified by hand, it is not recommended except in specific situations as shown in the STATUS example.

STATUS is typically used with WHENEVER ERROR CONTINUE or WHENEVER ERROR CALL, or TRY/CATCH blocks, to identify the type of error that occurred.

STATUS will be set for expression evaluation errors only when WHENEVER ANY ERROR is used.

After an SQL statement execution, STATUS contains the value of SQLCA.SQLCODE.

STATUS is set to an error code when an instruction produces an error, or it is reset to zero when non-assignment instructions succeed. A typical mistake is to test STATUS after a DISPLAY STATUS instruction, written after an SQL statement:

```
WHENEVER ERROR CONTINUE
DELETE FROM _invalid_table_name_ where col = 1
WHENEVER ERROR STOP
DISPLAY "STATUS:", STATUS   -- this DISPLAY instruction reset STATUS to zero
IF STATUS<0 THEN            -- Will never be the case, since STATUS==0
   DISPLAY "SQL Error!"
   EXIT PROGRAM 1
END IF
```

> **Tip:** Use SQLCA.SQLCODE for SQL error detection, and use STATUS for other language instructions.

Example

```
MAIN
  DISPLAY is_number(NULL)
  DISPLAY is_number("abc")
  DISPLAY is_number("-12.45")
END MAIN

FUNCTION is_number(s)
  DEFINE s STRING
  DEFINE f FLOAT, l_status INTEGER
  IF length(s)==0 THEN
     RETURN FALSE
  END IF
  WHENEVER ANY ERROR CONTINUE
  LET STATUS=0 # Needed, as STATUS won't be set if succeeds
  LET f = s
  LET l_status = STATUS
  WHENEVER ANY ERROR CONTINUE
  IF l_status == 0 THEN
     RETURN TRUE
  ELSE
     RETURN FALSE
  END IF
END FUNCTION
```

INT_FLAG

INT_FLAG is a predefined variable set to TRUE when an interruption event is detected.

Syntax

```
INT_FLAG
```

Usage

INT_FLAG is set to TRUE by the runtime system when an interruption event is detected by the runtime system. The interruption event is raised when the user presses the interruption key, or when the graphical front-end sends an interruption event while the program is running in a procedure or SQL query.

INT_FLAG must be used with the DEFER INTERRUPT configuration instruction. If the DEFER INTERRUPT instruction is not specified, and interruption signal will stop the program execution.

When the interruption event arrives during a procedural instruction (FOR loop), the runtime system sets INT_FLAG to TRUE. It is up to the program to check the INT_FLAG variable.

When the interruption event arrives during an interactive instruction (INPUT, CONSTRUCT), the runtime system sets INT_FLAG to TRUE and exits from the interactive instruction. It is recommended that you test INT_FLAG after an interactive instruction to check whether the input has been cancelled.

Once INT_FLAG is set to TRUE, it must be reset to FALSE in order to detect a new interruption event.

INT_FLAG will also be used by the runtime system as diagnostic flag for predefined action block execution such as ON INSERT in DISPLAY ARRAY.

Example

```
MAIN
  DEFER INTERRUPT
  LET INT_FLAG = FALSE
  INPUT BY NAME ...
     AFTER INPUT
        IF INT_FLAG THEN
           MESSAGE "The input is canceled."
        END IF
     ...
  END INPUT
  ...
END MAIN
```

QUIT_FLAG

QUIT_FLAG is a predefined variable set to TRUE when a quit event is detected.

Syntax

```
QUIT_FLAG
```

Usage

QUIT_FLAG is set to TRUE when a quit event is detected by the runtime system. The quit event is raised when the user presses the quit signal key ([Ctrl]+[Backslash]), or when another process sends the quit signal to the runtime system process.

QUIT_FLAG must be used with the DEFER QUIT configuration instruction. If the DEFER QUIT instruction is not specified, and quit signal will stop the program execution.

When the quit event arrives during a procedural instruction (FOR loop), the runtime system sets QUIT_FLAG to TRUEand continues the program execution. It is up to the program to check the QUIT_FLAG variable.

When the quit event arrives during an interactive instruction (INPUT, CONSTRUCT), the runtime system sets QUIT_FLAG to TRUE and continues with the execution of the interactive instruction.

Once QUIT_FLAG is set to TRUE, it must be reset to FALSE to detect a new quit event.

Example

```
MAIN
  DEFINE n INTEGER
  DEFER QUIT
  LET QUIT_FLAG = FALSE
  FOR n = 1 TO 1000
     IF QUIT_FLAG THEN EXIT FOR END IF
     ...
  END FOR
END MAIN
```

Program execution

This section describes program execution and language instructions related to program execution.

- Executing programs on page 390
- RUN on page 391
- EXIT PROGRAM on page 394
- BREAKPOINT on page 394

Executing programs

There are different ways to execute compiled programs, according to the configuration and the development or production context.

Prerequisites before executing a program

Make sure that all required environment variables are properly defined, such as FGLPROFILE, FGLGUI, FGLSERVER, FGLLDPATH, LANG/LC_ALL.

To display program forms in graphical mode, the GUI front-end must run on the computer defined by FGLSERVER, and all network security components (i.e. firewalls) must allow TCP connections on the port defined by this environment variable.

Verify the database client environment settings, and check that the database server is running and can be accessed, for example by using a database vendor specific tool to execute SQL commands.

Starting a program from the command line on the server

A program can be executed with the `fglrun` tool from the server command line:

```
fglrun myprogram
```

This method is typically used in development context. After compiling the programs and forms, for example with the `make` utility, execute the programs with `fglrun`.

> **Note:** The file extension (`.42m` or `.42r`) can be omitted. If no file extension is specified, `fglrun` will try to load *`progname`*`.42r`, then *`progname`*`.42m`.

Executing sub-programs from a parent program with RUN

Sub-programs can be executed from the main program with the `RUN` instruction. There can be limitations, according to the platform where the parent program executes.

Starting a program from the front-end

It is also possible to start programs on the application server from the platform where the front-end resides.

This is actually the typical way to start applications in a production environment.

- For a desktop front-end (GDC) application, define application shortcuts and use rlogin/ssh network protocols to start programs on the server or by using HTTP through a web server (GAS).
- For a web-browser application (GWC), configure the application server (GAS) to run applications from an URL.
- For a mobile device application (GMI/GMA), in a configuration where the programs run on a GAS application server, use the "`runOnServer`" front call, to start a program from the GAS.

Starting programs on a mobile device

After deploying program files on a mobile device, it can be executed as a local application, typically with a tap on the application icon.

- For a GMA (Android™) application, program files and GMA must be bundled together in an `.apk` Android package to be deployed. For more details, see Deploying mobile apps on Android devices on page 2572.
- For a GMI (iOS) application, program files and GMI must be bundled together in an `.ipa` package to be deployed. For more details, see Deploying mobile apps on iOS devices on page 2584.
- To start programs on an application server from a small embedded mobile application (starter), use the `runOnServer` front call. For more details, see Running mobile apps on an application server on page 2595.

Common app directories on mobile platforms

On mobile devices, you can use the following APIs to get common directories:

1. `base.Application.getProgramDir` `on page 1705` returns the directory path where the main `.42m` is located. Consider this location read-only and safe (no other app can access it).
2. `os.Path.pwd` `on page 2004` returns the path to the current working directory. When a mobile application is started, the GMA and the GMI set the working directory to the default application directory. Consider this location read-write and safe (no other app can access it).
3. The front call `standard.feInfo/dataDirectory` returns the front-end side temporary directory. Storage on this directory may be erased by the OS. On an embedded mobile application, as the runtime and the front-end run on the same system, the program can use this front call to retrieve a temporary directory and use the path to store temporary files. Consider this location read-write and unsafe. Applications executed remotely through a `runOnServer` front call, can use the `sandboxRunOnServer` directory under the directory returned by the `feInfo/dataDirectory` front call, to exchange files with the embedded application.

RUN

The `RUN` instruction executes the command passed as argument.

Syntax

```
RUN command
  [ IN {FORM|LINE} MODE ]
  [ RETURNING variable | WITHOUT WAITING ]
```

1. *command* is a string expression with the command to be executed.
2. *variable* is an integer variable receiving the execution status of the command.

Understanding the RUN command

The RUN instruction hands the argument command to the command interpreter. When not specifying the WITHOUT WAITING clause, the calling process waits for the called process to finish execution. Otherwise, the calling process waits the command termination.

Important: The RUN instruction has limited support on mobile platforms.

- The RUN instruction is not supported on mobile devices, because of operating system limitations.
- RUN *command* WITHOUT WAITING is not supported when programs run on an application server and display on a mobile device, because the Genero GUI protocol is not able to handle multiple connections at the same time.

Defining the command execution shell

In order to execute the command line, the RUN instruction uses the OS-specific shell defined in the environment of the current user. On UNIX™, this is defined by the **SHELL** environment variable. On Windows™, this is defined by **COMSPEC**. On Windows™, the program defined by the **COMSPEC** variable must support the ***/c*** option as CMD.EXE.

Waiting for the subprocess

By default, the runtime system waits for the end of the execution of the command, suspending the execution of the current program. After executing the command, the display of the parent program is restored.

If you specify WITHOUT WAITING, the specified command line is executed as a background process, and generally does not affect the visual display. This clause can be used when the command takes some time to execute, and the parent program does not need the result to continue. It is also typically used in GUI mode to start another program. Do not use this clause in TUI mode when the sub-program displays forms, otherwise both programs would run simultaneously on the same terminal.

Catching the execution status

The RETURNING clause saves the termination status code of the command that RUN executes in a program variable of type SMALLINT. Examine the variable after execution to determine the next action to take. A status code of zero usually indicates that the command has terminated normally. A non-zero exit status indicates an error.

Important:

The execution status provided by the RETURNING clause is platform-dependent. On UNIX™ systems, the value is composed of two bytes having different meanings. On Windows™ platforms, the execution status is usually zero for success, not zero if an error occurred.

On UNIX™ systems, the lower byte (x mod 256) of the return status defines the termination status of the RUN command. The higher byte (x / 256) of the return status defines the execution status of the program. On Windows™ systems, the value of the return status defines the execution status of the program.

IN LINE MODE and IN FORM MODE

When using the TUI mode, programs operate by default in *line mode*, but as many statements take it into *form mode* (including OPTIONS statements that set keys, DISPLAY, OPEN WINDOW, DISPLAY FORM>, and other screen interaction statements), typical interactive TUI programs are actually in *form mode* most of the time.

According to the type of command to be executed, you may need to use the IN {LINE|FORM} MODE clause with the RUN instruction. It defines how the terminal or the graphical front-end behaves when running the child process.

Besides RUN, the OPTIONS, START REPORT and REPORT statements can explicitly specify a screen mode. If no screen mode is specified in the RUN command, the current value from the OPTIONS statement is used. This is, by default, IN LINE MODE. The default screen mode for PIPE specifications in reports is IN FORM MODE.

When the RUN statement specifies IN FORM MODE, the program remains in *form mode* if it is currently in *form mode*, but it does not enter *form mode* if it is currently in *line mode*. When the prevailing RUN option specifies IN LINE MODE, the program remains in *line mode* if it is currently in *line mode*, and it switches to *line mode* if it is currently in *form mode*. This also applies to the PIPE option.

Typically, if you need to run another interactive program, you must use the IN LINE MODE clause:

- In TUI mode, the terminal is in the same state (in terms if *tty* options) as when the program began. Usually the terminal input is in cooked mode, with interrupts enabled and input not becoming available until after a newline character is typed.
- In GUI mode, if the WITHOUT WAITING clause in used, the front-end is warned before the child process is started (this causes a first network round-trip) After the child is started, the front-end is warned that the command was executed (second network round-trip). If the RUN command must wait for child termination (i.e. no WITHOUT WAITING clause is used), no particular action is taken.

However, if you want to execute a subprocess running silently (batch program without output), you must use the IN FORM MODE clause:

- In TUI mode, the screen stays in *form mode* if it was in *form mode*, which saves a clear / redraw of the screen. The FORM mode specifies the terminal raw mode, in which each character of input becomes available to the program as it is typed or read.
- In GUI mode, no particular action is taken to warn the front-end (there is no need to warn the front-end for batch program execution).

To summarize, no matter if you are in TUI or GUI mode, run silent (batch) programs in FORM MODE, and if the program to run is interactive, displays messages to the terminal, or if you don't known what it does, use the LINE MODE (witch is the default).

A good practice is to encapsulate child program and system command execution in functions.

Example

```
MAIN
  DEFINE result SMALLINT
  CALL runApplication("app2 -p xxx")
  CALL runBatch("ls -l", FALSE) RETURNING result
  CALL runBatch("ls -l > /tmp/files", TRUE) RETURNING result
END MAIN

FUNCTION runApplication(pname)
  DEFINE pname, cmd STRING
  LET cmd = "fglrun " || pname
  IF fgl_getenv("FGLGUI") == 0 THEN
    RUN cmd
  ELSE
    RUN cmd WITHOUT WAITING
  END IF
END FUNCTION

FUNCTION runBatch(cmd, silent)
  DEFINE cmd STRING
  DEFINE silent STRING
  DEFINE result SMALLINT
  IF silent THEN
    RUN cmd IN FORM MODE RETURNING result
  ELSE
    RUN cmd IN LINE MODE RETURNING result
```

```
  END IF
  IF fgl_getenv("OS") MATCHES "Win*" THEN
    RETURN result
  ELSE
    RETURN ( result / 256 )
  END IF
END FUNCTION
```

EXIT PROGRAM

The `EXIT PROGRAM` instruction terminates the execution of the program.

Syntax

```
EXIT PROGRAM [ exit-code ]
```

1. *exit-code* is a valid integer expression that can be read by the process which invoked the program.

Usage

Use the `EXIT PROGRAM` instruction to stop the execution of the current program instance.

exit-code must be zero by default for normal, successful program termination.

exit-code is converted into a positive integer between 0 and 255 (8 bits).

Example

```
MAIN
  DISPLAY "Emergency exit."
  EXIT PROGRAM -1
  DISPLAY "This will never be displayed."
END MAIN
```

BREAKPOINT

The `BREAKPOINT` instruction sets a program breakpoint when running in debug mode.

Syntax

```
BREAKPOINT
```

Usage

Normally, to set a breakpoint when you debug a program, you must use the *break* command of the debugger. But in some situations, you might need to set the breakpoint in program sources. Therefore, the `BREAKPOINT` instruction has been added to the language.

When you start fglrun in debug mode with the -d option, if the program flow encounters a `BREAKPOINT` instruction, the program execution stops and the debug prompt is displayed, to let you enter a debugger command. The `BREAKPOINT` instruction is ignored when not running in debug mode.

Example

```
MAIN
  DEFINE i INTEGER
  LET i=123
```

```
  BREAKPOINT
  DISPLAY i
END MAIN
```

Front calls

Front call functions execute on the platform where the front-end is installed.

- Understanding front calls on page 395
- ui.Interface.frontCall on page 395
- User-defined front calls on page 397

Understanding front calls

Front calls execute a native function on the front-end platform.

In your Genero program, use the `ui.Interface.frontCall()` class method to invoke front-end functions. When calling a user function from programs, specify a module name and a function name. Input and output parameters can be passed/returned in order to transmit/receive values to/from the front-end. A typical example is an "open file" dialog window that allows you to select a file from the front-end workstation file system.

> **Important:** Some front calls are specific to the platform or front-end technology and may not by supported. For example, it is not possible to execute a shell command (shellexec) with the Web Browser front-end.

A set of front-end functions is built-in by default in front-ends. However, it is possible to write your own functions in order to extend the front-end possibilities.

> **Tip:** While you can use DDE/OLE APIs to manipulate Microsoft™ Office documents, there are freeware alternatives such as the Apache POI Java™ library which can be used with the Java™ Interface. For an example, see Java™ Interface: Example 2 .

ui.Interface.frontCall

`ui.Interface.frontCall` performs a function call to the current front-end.

Syntax

```
ui.Interface.frontCall(
   module STRING,
   function STRING,
   [ parameter-list ],
   [ returning-list ] )
```

1. *module* defines the shared library or classpath where the function is implemented.
2. *function* defines the name of the function to be called.
3. *parameter-list* is a list of input parameters.
4. *returning-list* is a list of output parameters.

> **Important:** The *returning-list* variables are passed by reference to the `frontCall()` method.

Usage

The `ui.Interface.frontCall()` class method can be used to execute a procedure on the front-end workstation through the front-end software component. You can for example launch a front-end specific application like a browser or a text editor, or manage the clipboard content.

The method takes four parameters:

1. The module, identifying the shared library (.so or .DLL) or the Java class (GMA) implementing the front call function.
2. The function of the module the be executed.
3. The list of input parameters, using the square brace notation.
4. The list of output parameters, using the square brace notation.

Input and output parameters are provided as a variable list of parameters, by using the square braces notation (`[param1,param2,...]`). Input parameters can be an expression supported by the language; output parameters must be variables only, to receive the returning values. An empty list is specified with `[]`. Output parameters are optional: If the front call returns values, they will be ignored by the runtime system.

Simple front call example:

```
FUNCTION call()
  DEFINE info STRING
  CALL ui.Interface.frontCall( "standard", "feInfo", ["feName"], [info] )
END FUNCTION
```

Some front calls need a file path as parameter. File paths must follow the syntax of the front end workstation file system. You may need to escape backslash characters in such parameters. The next example shows how to pass a file path with a space in a directory name to a front-end running on a Microsoft™ Windows™ workstation:

```
FUNCTION call()
   DEFINE path STRING, res INTEGER
   LET path = "\"c:\\work dir\\my report.doc\""
   -- This is: "c:\work dir\my report.doc"
   CALL ui.Interface.frontCall( "standard", "shellExec", [path], [res] )
END FUNCTION
```

Front call error handling

Exception handling instructions can be used to check the execution status of a front call. Both `WHENEVER ERROR` directives or `TRY/CATCH` block can surround the front call to avoid program stop in case of error, and check the error number returned in the `STATUS` variable.

> **Note:** There is not need to surround front calls with exception handlers such as `TRY/CATCH`, if the front call is always supposed to execute without error. For example, the feInfo front call will never produce an exception.

Example of front call error handling with a `TRY/CATCH` block:

```
FUNCTION takePhoto()
   DEFINE path STRING
   TRY -- This front call may fail if the front-end is not a mobile device:
       CALL ui.Interface.frontCall( "mobile", "takePhoto", [], [path] )
   CATCH
       MESSAGE "Cannot take photo: ", STATUS, " ", err_get(STATUS)
       LET path = NULL
   END TRY
   RETURN path
END FUNCTION
```

If the front call module name or the function name is invalid, the errors -6331 or -6332 will be raised, respectively.

If the front call execution failed for some reason, the error -6333 will be raised. The description of the problem can be found in the second part of the error message, returned by a call to the ERR_GET() function.

The error -6334 can be raised in case of input or output parameter mismatch. The control of the number of input and output parameters is in the hands of the front-end. Most of the standard front calls have optional returning parameters and will not raise error -6334, if the output parameter list is left empty. However, front-end specific extensions or user-defined front-end functions may return an invalid execution status in case of input or output parameter mismatch, raising error -6334. If the front-end sends an call execution status of zero (OK), and the number of returned values does not match the number of program variables, the runtime system will set unmatched program variables to `NULL`. As a general rule, the program should provide the expected input and output parameters as specified in the documentation.

User-defined front calls

Extend the Genero language possibilities by implementing your own front-end functions.

For more details, see User-defined front calls on page 1615.

SQL support

These topics cover SQL support in the Genero Business Development Language.

- SQL programming on page 398
- Database connections on page 457
- Database transactions on page 480
- Static SQL statements on page 486
- Dynamic SQL management on page 500
- Result set processing on page 504
- Positioned updates/deletes on page 514
- SQL insert cursors on page 517
- SQL load and unload on page 524
- SQL adaptation guides on page 529

SQL programming

Covers topics about interacting with a database server using SQL.

- SQL basics on page 398
- SQL security on page 410
- SQL portability on page 412
- SQL performance on page 452

SQL basics

- SQL execution diagnostics on page 398
- The SQLCA diagnostic record on page 401
- SQL error identification on page 402
- SQL interruption on page 405
- Debugging SQL on page 406
- Cursors and connections on page 406
- Implicit database connection on page 407
- The database utility library on page 408
- Handling nested transactions on page 408
- Transaction blocks across connections on page 409
- The base.SQLHandle built-in class on page 410

SQL execution diagnostics

If an SQL statement execution failed, error description can be found in the `SQLCA.SQLCODE`, `SQLSTATE`, `STATUS` and `SQLERRMESSAGE` predefined registers.

Trapping SQL errors

By default, SQL errors stop program execution and display the error message to the standard output. Most SQL statements executed by a program should not return an error and thus do not require error trapping. However, in some cases, a program must keep the control when an SQL error occurs. For example, when connecting to the database, the user might enter an invalid password that will raise a login denied error. The program must trap such SQL connection error to return to the login dialog and let the user enter a new login and password.

To trap potential SQL errors, surround the SQL statements to be checked either with a `WHENEVER ERROR` exception handler or with a `TRY / CATCH` block:

```
-- WHENEVER ERROR handler
WHENENEVER ERROR CONTINUE
  INSERT INTO orders VALUES ( rec_ord. * )
  IF SQLCA.SQLCODE = -75623 THEN
     ...
  END IF
WHENEVER ERROR STOP -- restore the default

-- TRY/CATCH block
TRY
  INSERT INTO orders VALUES ( rec_ord. * )
CATCH
  IF SQLCA.SQLCODE = -75623 THEN
     ...
  END IF
END TRY
```

Using SQLCA.SQLCODE

SQL error codes are provided in the `SQLCA.SQLCODE` register. This register always contains an IBM® Informix® error code, even when connected to a database different from IBM® Informix®.

`STATUS` is the global language error code register, set for any kind of error (even non-SQL). When an SQL error occurs, the error held in `SQLCA.SQLCODE` is copied into `STATUS`.

Use `SQLCA.SQLCODE` for SQL error management, and `STATUS` to detect errors with other language instructions.

When connecting to a database different from IBM® Informix®, the database driver tries to convert the native SQL error to an IBM® Informix® error which will be copied into the `SQLCA.SQLCODE` and `STATUS` registers. If the native SQL error cannot be converted, `SQLCA.SQLCODE` and `STATUS` will be set to **-6372** (a general SQL error), you can then check the native SQL error in `SQLCA.SQLERRD[2]`. The native SQL error code is always available in `SQLCA.SQLERRD[2]`, even if it could not be converted to an IBM® Informix® error.

Using SQLSTATE

`SQLSTATE` contains an error code that follows ISO/ANSI standard error specification, but not all database servers support this register. Using `SQLSTATE` for SQL error checking should be the preferred way for portable SQL programming, as long as the target databases support this feature.

The `SQLSTATE` codes are defined by the ANSI/ISO standard specification, however not all database types support this standard.

Table 136: SQLSTATE error codes support per database server type

Database Server Type	Supports SQLSTATE errors
IBM® DB2® UDB (UNIX™)	Yes, since version 7.1
IBM® Informix®	Yes, since IDS 10
Microsoft™ SQL Server	Yes, since version 8 (2000)
MySQL	Yes
Oracle Database Server	Not in version 10.2
PostgreSQL	Yes, since version 7.4

Database Server Type	Supports SQLSTATE errors
Sybase ASE	Yes

Centralize SQL error checking

SQL error identification sometimes requires complex code, checking different error numbers that can be RDBMS-specific. Therefore, it is strongly recommended that you centralize SQL error identification in a function. This will allow you to write RDBMS-specific code, when needed, only once.

For maximum SQL portability, centralize SQL error checking in functions, to test either `SQLCA.SQLCODE` or `SQLSTATE`, according to the target database, and define your own error identifiers with constants:

```
CONSTANT SQLERR_INVALID_DATABASE = -1001,
         SQLERR_INVALID_USER = -1002,
         ...

FUNCTION do_connect()
  DEFINE uname, upswd VARCHAR(100)
  WHILE TRUE
    CALL login() RETURNING uname, upswd
    TRY
       CONNECT TO "stores" USER uname USING upswd
    CATCH
       CASE check_sql_error()
       WHEN SQLERR_INVALID_DATABASE
          DISPLAY SQLERRMESSAGE
          EXIT PROGRAM 1 -- Fatal error: Stop!
       WHEN SQLERR_INVALID_USER
          ERROR "Invalid login, try again"
          CONTINUE WHILE
       END CASE
    END TRY
    EXIT WHILE
  END WHILE
END FUNCTION
```

SQL error messages

`SQLERRMESSAGE` contains the database-specific error message. These messages are different for every database type and should only be used to print or log SQL execution diagnostic information.

SQL warnings

Some SQL instructions can produce SQL Warnings. Compared to SQL Errors which do normally stop the program execution, SQL Warnings indicate a minor issue that can often be ignored. For example, when connecting to an IBM® Informix® database, a warning is returned to indicate that a database was opened, and an other warning might be returned if that database supports transactions. None of these facts are critical problems, but knowing that information can help for further program execution.

If an SQL Warning is raised, `SQLCA.SQLCODE` / `STATUS` remain zero, and the program flow continues. To detect if an SQL Warning occurs, the SQLCA.SQLAWARN register must be used. `SQLCA.SQLAWARN` is defined as a `CHAR(7)` variable. If `SQLCA.SQLAWARN[1]` contains the `W` letter, it means that the last SQL instruction has returned a warning. The other character positions (`SQLCA.SQLAWARN[2-8]`) may contain W letters. Each position from 2 to 8 has a special meaning according to the database server type, and the SQL instructions type.

If `SQLCA.SQLAWARN` is set, you can also check the `SQLSTATE` and `SQLCA.SQLERRD[2]` registers to get more details about the warning. The `SQLERRMESSAGE` register might also contain the warning description.

In the next example, the program connects to a database and displays the content of the SQLCA.SQLAWARN register. When connecting to an IBM® Informix® database with transactions, the program will display [WW W]:

```
MAIN
  DATABASE stores
  DISPLAY "[", sqlca.sqlawarn, "]"
END MAIN
```

By default SQL Warnings do not stop the program execution. To trap SQL Warnings with an exception handle, use the WHENEVER WARNING instruction, as shown in this example.

```
MAIN
  DEFINE cust_name VARCHAR(50)
  DATABASE stores
  WHENEVER WARNING STOP
  SELECT cust_lname, cust_address INTO cust_name
       FROM customer WHERE cust_id = 101
  WHENEVER WARNING CONTINUE
END MAIN
```

The SELECT statement in this example uses two columns in the select list, but only one INTO variable is provided. This is legal and does not raise an SQL Error, however, it will set the SQLCA.SQLAWARN register to indicate that the number of target variables does not match the select-list items.

See also WHENEVER WARNING exception.

Display detailed debug information in case of internal driver error

If an unexpected problem happens within the database driver, the driver will return the error **-6319** (internal error in the database library). When this SQL error occurs, set the FGLSQLDEBUG environment variable to get more details about the internal error.

The SQLCA diagnostic record

The SQLCA variable is a predefined record containing SQL statement execution information.

The SQLCA record definition

The SQLCA record is defined as follows:

```
DEFINE SQLCA RECORD
  SQLCODE INTEGER,
  SQLERRM VARCHAR(71),
  SQLERRP CHAR(7),
  SQLERRD ARRAY[6] OF INTEGER,
  SQLAWARN CHAR(7)
END RECORD
```

1. SQLCODE contains the SQL execution code (0 = OK, 100 = not row found, <0 = error).
2. SQLERRM contains the error message parameter.
3. SQLERRP is not used at this time.
4. SQLERRD[1] is not used at this time.
5. SQLERRD[2] contains the last SERIAL or the native SQL error code.
6. SQLERRD[3] contains the number of rows processed in the last statement (server dependent).
7. SQLERRD[4] contains the estimated CPU cost for the query (server dependent).
8. SQLERRD[5] contains the offset of the error in the SQL statement text (server dependent).
9. SQLERRD[6] contains the ROWID of the last row that was processed (server dependent).
10. SQLAWARN contains the ANSI warning represented by a W character at a given position in the string.

11.SQLAWARN[1] is set to W when any of the other warning characters have been set to W.
12.SQLAWARN[2-7] have specific meanings, see database server documentation for more details.

Usage

SQLCA stands for the *SQL Communication Area* variable.

The SQLCA can be used to get an SQL execution diagnostic. Error and warning information can be found in this structure.

The SQLCA record is filled after each SQL statement execution.

SQLCA is not designed to be modified by user code, it must be used as a read-only record.

Portability

SQLCA.SQLCODE will be set to a specific IBM® Informix® SQL error code, if the database driver can convert the native SQL error to an IBM® Informix® SQL error. In case of error, SQLCA.SQLERRD[2] will hold the native SQL error produced by the database server.

Other SQLCA record members are specific to IBM® Informix® databases. For example, after inserting a row in a table with a SERIAL column, SQLCA.SQLERRD[2] contains the new generated serial number. After an SQL error, SQLCA.SQLERRD[2] contains the native SQL error. The SQLCA.SQLERRD[3] member may be set with the number of processed rows, if the database client provides the API. Other SQLCA.SQLERRD[n] members must be considered as not portable.

Example

```
MAIN
  WHENEVER ERROR CONTINUE
  DATABASE stores
  SELECT COUNT(*) FROM foo   -- Table should not exist!
  DISPLAY SQLCA.SQLCODE, SQLCA.SQLERRD[2]
END MAIN
```

SQL error identification

Identify SQL exceptions in your programs with SQLCA.SQLCODE.

Every database type has its own error numbers. Portable SQL code must take care of this when checking for SQL errors in programs.

The IBM® Informix® compatible error code is stored in the SQLCA.SQLCODE register. This is done to simplify migration to another database type. Existing code based on Informix® error numbers does not need to be modified.

Database drivers map native SQL errors to Informix SQL errors, as listed in the following table:

Table 137: Native SQL error to Informix SQL error mappings

Informix SQL	Oracle DB	SQL Server	IBM DB2	PostgreSQ	MySQL	Sybase ASE	SQLite	Netezza
-201	900:902, 905:911, 914, 917, 920:931, 933:936, 938:940, 946, 950,	102, 170, 101, 1103, 3005, 3014	-101, -104, -106, -108, -109, -127, -142, -143	03000, 42000, 42501, 42601	1064, 1121	102, 156	N/A	21

Informix SQL	Oracle DB	SQL Server	IBM DB2	PostgreSQ	MySQL	Sybase ASE	SQLite	Netezza
	954, 957, 958, 962, 964, 966:971, 978:979, 982, 984, 985, 990, 992:996, 998:999							
-204	N/A	3016	-103	N/A	N/A	N/A	N/A	N/A
-206	903, 942	3701, 4004	-204	42P01	1146, 1051	207, 208, 3701	N/A	29
-217	904	4005	-205, -206	42703	1054	N/A	N/A	31
-236	913, 947	1200	-117	N/A	N/A	213	N/A	N/A
-244	N/A	1222	N/A	N/A	N/A	12205	N/A	N/A
-251	N/A	N/A	-125	N/A	N/A	N/A	N/A	N/A
-253	972	2014	-107	N/A	N/A	N/A	N/A	N/A
-254	1008, 1475	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-255	N/A	N/A	N/A	25P01	N/A	N/A	N/A	N/A
-257	1000	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-263	54	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-268	1	2601, 2627	-803	23505	1062	2601	N/A	N/A
-280	N/A	N/A	-102	N/A	N/A	N/A	N/A	N/A
-282	N/A	N/A	-105	N/A	N/A	N/A	N/A	N/A
-294	937	N/A	-119, -122	N/A	N/A	N/A	N/A	N/A
-316	N/A	N/A	-605	N/A	N/A	N/A	N/A	N/A
-324	960	N/A	-203	N/A	N/A	N/A	N/A	N/A
-350	1408	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-360	N/A	N/A	-118	N/A	N/A	N/A	N/A	N/A
-371	1452	N/A	-603, -673	N/A	N/A	N/A	N/A	N/A
-382	1756	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-387	1017, 1045	715, 4002, 4003, 4008	-1403, -1404	N/A	1045	N/A	N/A	24
-388	1536	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-391	1400, 1407	N/A	-407	22004, 23502	N/A	N/A	N/A	N/A

Informix SQL	Oracle DB	SQL Server	IBM DB2	PostgreSQ	MySQL	Sybase ASE	SQLite	Netezza
-400	1002	N/A	N/A	N/A	N/A	N/A	N/A	N/A
-517	N/A	N/A	-602	N/A	N/A	N/A	N/A	N/A
-530	2290	N/A	-193	23514	N/A	548	N/A	N/A
-551	N/A	N/A	-613	N/A	N/A	N/A	N/A	N/A
-674	N/A	2812	N/A	N/A	N/A	14216	N/A	N/A
-681	N/A	2812	-121	N/A	N/A	N/A	N/A	N/A
-691	2291	547	-530	23503	1452	546	19	N/A
-743	955	6000, 6006, 6008	N/A	N/A	N/A	2714	N/A	N/A
-930	1033, 1034, 12154, 12203, 12224, 12500, 12560	11, 17, 708, 709, 711, 4014, 17142	-1013	08000, 08001, 08004, 08006, 08007, 08000	1044	4002	N/A	N/A
-942	N/A	N/A	-903	N/A	N/A	N/A	N/A	N/A
-1202	N/A	N/A	-801	N/A	N/A	N/A	N/A	N/A
-1218	N/A	3048, 3049, 3050	-180, -181	N/A	N/A	N/A	N/A	N/A
-1260	932	N/A	-190	N/A	N/A	N/A	N/A	N/A
-1279	1401	N/A	-433, -99998	N/A	N/A	N/A	N/A	N/A
-1349	1722	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Sometimes the native error code of the database cannot be converted to an Informix® error code. In such case, the `SQLCA.SQLCODE` register will be set to **-6372**. To properly identify an SQL error, the native SQL error code is also provided in the `SQLCA.SQLERRD[2]` register.

Centralize SQL error identification in a function:

```
-- sqlerr.4gl module

PUBLIC CONSTANT SQLERRTYPE_FATAL = -1
PUBLIC CONSTANT SQLERRTYPE_LOCK  = -2
PUBLIC CONSTANT SQLERRTYPE_CONN  = -3

FUNCTION lastSqlErrorType()
   CASE
     WHEN SQLCA.SQLCODE == -201
            OR SQLCA.SQLERRD[2] == ...
       RETURN SQLERR_FATAL
     WHEN SQLCA.SQLCODE == -263
            OR SQLCA.SQLCODE == -244
            OR SQLCA.SQLERRD[2] == ...
```

```
        RETURN SQLERR_LOCK
      ...
   END CASE
END FUNCTION
```

You can then easily use this function after every SQL statement in your programs:

```
IMPORT FGL sqlerr
MAIN
   DATABASE stores
   WHENEVER ERROR CONTINUE
   UPDATE customer SET cust_address = NULL
     WHEN cust_name IS NULL
   IF lastSqlErrorType() == SQLERRTYPE_LOCK THEN
       ...
   END IF
   ...
END MAIN
```

SQL interruption
Interrupt long running SQL queries, or interrupt waiting queries because data is locked.

If the database server supports SQL interruption, a program can interrupt a long running SQL statement.

SQL interruption is not enabled by default. use the `OPTIONS SQL INTERRUPT ON` program option to turn on SQL interruption.

With `OPTIONS SQL INTERRUPT ON`, when the program gets an interruption event (a SIGINT signal from the system, or an interrupt event from the front-end), the running SQL statement is stopped, the `INT_FLAG` global variable is set to `TRUE`, and `SQLCA.SQLCODE` is set with error -213.

SQL interrupt must be used in conjunction with signal handling instructions `DEFER INTERRUPT` and `DEFER QUIT`, otherwise the program would stop immediately in case of interruption event.

SQL interruption results in abnormal SQL statement execution and raises a runtime error. Therefore, the SQL statement that can be subject of interruption must be protected by a `WHENEVER ERROR` exception handler.

```
MAIN
  DEFINE n INTEGER
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE test1
  WHENEVER ERROR CONTINUE
  -- Start long query (self join takes time)
  -- From now on, user can hit CTRL-C in TUI mode to stop the query
  SELECT COUNT(*) INTO n FROM customers a, customers b
       WHERE a.cust_id <> b.cust_id
  IF SQLCA.SQLCODE == -213 THEN
     DISPLAY "Statement was interrupted by user..."
     EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  ...
END MAIN
```

When SQL interruption is supported by the database server type that is different from IBM® Informix®, the database drivers will return error -213 in case of interruption, to behave as in IBM® Informix®.

> **Important:** Not all database servers support SQL interruption.

Table 138: Database server support of SQL interruption

Database Server Type	SQL Interruption API	SQL error code for interrupted query
IBM® DB2® UDB (Since version 9.x)	SQLCancel()	Native error -952
IBM® Informix®	sqlbreak()	Native error -213
Microsoft™ SQL Server (Only 2005+ with SNC driver)	SQLCancel()	SQLSTATE HY008
MySQL	KILL QUERY command	Native error -1317
Oracle Database Server	OCIBreak()	Native error -1013
PostgreSQL	PQCancel()	SQLSTATE 57014
Sybase ASE	ct_cancel()	SQLSTATE HY008
SQLite	sqlite3_interrupt()	Native error SQLITE_ABORT

Debugging SQL
Set the FGLSQLDEBUG environment variable to print SQL debug info.

SQL debug information is printed by the runtime system when the FGLSQLDEBUG environment variable is defined. This variable can be set to an integer value from 0 to 10, according to the debugging details you want to see. The debug messages are sent to the standard error stream. If needed, you can redirect the standard error output into a file.

UNIX™ (shell) example:

```
FGLSQLDEBUG=3
export FGLSQLDEBUG
fglrun myprog 2>sqldbg.txt
```

An SQL debug header is printed before executing the underlying ODI driver code. If the driver code crashes or raises an assertion, you can easily find the last SQL instruction that was executed by the program, and report to you support center.

Cursors and connections
Several database connections can be opened simultaneously with the `CONNECT TO` instruction. Once connected, you can `DECLARE` cursors or `PREPARE` statements to be used in parallel within different connection contexts. This section describes how to use SQL cursors and SQL statements in a multiple-connection program.

When you `DECLARE` a cursor or when you `PREPARE` a statement, you actually create an *SQL statement handle*; the runtime system allocates resources for that statement handle before sending the SQL text to the database server via the database driver.

The SQL statement handle is created in the context of the current connection, and must be used in that context, until it is freed or recreated with another `DECLARE` or `PREPARE` statement. Using an SQL statement handle in a different connection context than the one for which it was created will produce a runtime error.

The `SET CONNECTION` instruction changes the connection context. Connections are identified by a name. The `AS` clause of the `CONNECT TO` instruction allows to specify a connection name. If the `AS` clause is omitted, the connection gets a default name based on the data source name.

This small program example illustrates the use of two cursors with two different connections:

```
MAIN
```

```
   CONNECT TO "db1" AS "s1"
   CONNECT TO "db2" AS "s2"
   SET CONNECTION "s1"
   DECLARE c1 CURSOR FOR SELECT tab1.* FROM tab1
   SET CONNECTION "s2"
   DECLARE c2 CURSOR FOR SELECT tab1.* FROM tab1
   SET CONNECTION "s1"
   OPEN c1
   SET CONNECTION "s2"
   OPEN c2
   ...
END MAIN
```

The DECLARE and PREPARE instructions are a type of creator instructions; if an SQL statement handle is recreated in a connection other than the original connection for which it was created, old resources are freed and new resources are allocated in the current connection context.

This allows you to re-execute the same cursor code in different connection contexts, as in this example:

```
MAIN
   CONNECT TO "db1" AS "s1"
   CONNECT TO "db2" AS "s2"
   SET CONNECTION "s1"
   IF checkForOrders() > 0 ...
   SET CONNECTION "s2"
   IF checkForOrders() > 0 ...
   ...
END MAIN

FUNCTION checkForOrders(d)
   DEFINE d DATE, i INTEGER
   DECLARE c1 CURSOR FOR SELECT COUNT(*) FROM orders WHERE ord_date = d
   OPEN c1
   FETCH c1 INTO i
   CLOSE c1
   FREE c1
   RETURN i
END FUNCTION
```

If the SQL statement handle was created in a different connection, the resources used in the old connection context are freed automatically, and new statement handle resources are allocated in the current connection context.

Implicit database connection

An implicit database connection is made with the DATABASE instruction used before MAIN; use SCHEMA to avoid the implicit connection.

The DATABASE statement can be used in two distinct ways, depending on the context of the statement within its source module:

- To specify a default database.

  Typically used in a GLOBALS module, to define variables with the DEFINE ... LIKE, but it is also used for the INITIALIZE and VALIDATE statements. Using the DATABASE statement in this way results in that database being opened automatically at run time.
- To specify a current database.

  In MAIN or in a FUNCTION, used to connect to a database. A variable can be used in this context (DATABASE *varname*).

A default database is almost always used, because many programs contain `DEFINE ... LIKE` statements. A problem occurs when the production database name differs from the development database name, because the default database specification will result in an automatic connection (just after `MAIN`):

```
DATABASE stock_dev  -- Default database, used at compile time
DEFINE
   p_cust RECORD LIKE customer.*
MAIN -- Connection to default database occurs at MAIN
   DEFINE dbname CHAR(30)
   LET dbname = "stock1"
   DATABASE dbname -- Real database used in production
   ...
END MAIN
```

In order to avoid the implicit connection, you can use the `SCHEMA` instruction instead of `DATABASE`:

```
SCHEMA stock_dev -- Schema specification only
DEFINE
   p_cust RECORD LIKE customer.*
MAIN -- No default connection occurs...
   DEFINE dbname CHAR(30)
   LET dbname = "stock1"
   DATABASE dbname
END MAIN
```

This instruction will define the database schema for compilation only, and will not make an implicit connection at runtime.

The database utility library

The `fgldbutl.4gl` library provides several database-related utility functions.

You find this library in the `FGLDIR/src` directory.

The DB utility library implements helpers for the following areas:

- Database type identification
- Sequence number generation
- Nested transaction control

See the `fgldbutl.4gl` source file for more details.

Handling nested transactions

You can manage nested transactions in different parts of a program.

A program can become very complex if it contains a lot of nested functions calls, doing SQL processing within transactions. You may want to centralize transaction control commands in wrapper functions. The `fgldbutl.4gl` library contains special functions to manage the beginning and the end of a transaction with an internal counter, in order to implement nested function calls inside a unique transaction.

```
MAIN
  IF a() <> 0 THEN
       ERROR "..."
  END IF
  IF b() <> 0 THEN
       ERROR "..."
  END IF
END MAIN

FUNCTION a()
  DEFINE s INTEGER
  LET s = db_start_transaction()
  UPDATE ...
```

```
  LET s = SQLCA.SQLCODE
  IF s = 0 THEN
     LET s = b()
  END IF
  LET s = db_finish_transaction((s==0))
  RETURN s
END FUNCTION

FUNCTION b()
   DEFINE s INTEGER
   LET s = db_start_transaction()
   UPDATE ...
   LET s = SQLCA.SQLCODE
   LET s = db_finish_transaction((s==0))
   RETURN s
END FUNCTION
```

In this example, you see in the `MAIN` block that both functions `a()` and `b()` can be called separately. However, the transaction SQL commands will be used only if needed: When function `a()` is called, it starts the transaction, then calls `b()`, which does not start the transaction since it was already started by `a()`. When function `b()` is called directly, it starts the transaction.

The function `db_finish_transaction()` is called with the expression `(s==0)`, which is evaluated before the call. This allows you to write in one line the equivalent of the following `IF` statement:

```
IF s==0 THEN
   LET s = db_finish_transaction(1)
ELSE
   LET s = db_finish_transaction(0)
END IF
```

Transaction blocks across connections

Transaction blocks manage transactions when connected to several database servers.

In some cases, you need to copy data from a database to another. Database vendor export / import tools exist for this task and should be used when a large amount of data needs to be transferred. However, it is also possible to achieve this with a BDL program connected to both databases, reading data from the source database and inserting rows into the target database.

If the rows created in the target database need to be committed as a whole, you must open a transaction with the `BEGIN WORK` instruction, use `SET CONNECTION` to switch between the connections to read/write rows, and terminate the transaction with a `COMMIT WORK`.

In order to keep a transaction open when switching to another database connection, the connection must be initiated with the `WITH CONCURRENT TRANSACTION` clause. If this option is not used, databases servers might raise an error when changing the connection context. For example IBM® Informix® will return the SQL error -1801: Multiple-server transaction not supported.

The example below opens two database connections, reads rows from a table of the first database, and uses a transaction to insert rows in a table of the second database:

```
MAIN
   DEFINE rec RECORD
               pk INTEGER,
               name VARCHAR(50)
           END RECORD

   CONNECT TO "test1+driver='dbmifx'" AS "s1"
           USER "ifxuser" USING "fourjs"
           WITH CONCURRENT TRANSACTION
   CREATE TEMP TABLE tt1 ( pk INT, name VARCHAR(50) )
   INSERT INTO tt1 VALUES ( 1, "Item 1" )
```

```
    INSERT INTO tt1 VALUES ( 2, "Item 2" )

    CONNECT TO "test1+driver='dbmmys'" AS "s2"
            USER "mysuser" USING "fourjs"
            WITH CONCURRENT TRANSACTION
    CREATE TEMP TABLE tt2 ( pk INT, name VARCHAR(50) )

    SET CONNECTION "s1"
    DECLARE c1 CURSOR FOR SELECT * FROM tt1

    SET CONNECTION "s2"
    BEGIN WORK

    SET CONNECTION "s1"
    FOREACH c1 INTO rec.*
        SET CONNECTION "s2"
        INSERT INTO tt2 VALUES ( rec.* )
        SET CONNECTION "s1"
    END FOREACH

    SET CONNECTION "s2"
    COMMIT WORK

END MAIN
```

The base.SQLHandle built-in class
Handle SQL queries with a 3GL API.

Genero BDL provides a 3GL API to execute SQL queries and introspect result set column information with the `base.SqlHandle` built-in class.

The class implements typical SQL statement execution methods existing in well-known APIs, such as:

- `prepare("sql-text")`
- `setParameter()`
- `execute()`
- `open()`, `openScrollCursor()`
- `fetch()`, `fetchFirst()`, `fetchLast()`, ...

The class also implements introspection methods for the result set columns:

- `getResultCount()`
- `getResultType(`*`index`*`)`
- `getResultName(`*`index`*`)`
- `getResultValue(`*`index`*`)`

This class is provided to allow generic code implementation for specific needs. Consider using traditional static and dynamic SQL instruction for regular code implementing your business rules; the 3GL code based on the SqlHandle class is not as readable as static or dynamic SQL.

SQL security

- Database user authentication on page 410
- Avoiding SQL injection on page 411

Database user authentication
Understanding how users are authenticated to the database server.

When connecting to a database server, the user must be identified by the server. Once connected, the current user is authenticated and identified by the db server, and the database system can then apply specific privileges, audit user activity, and so on.

Database user authentication is typically achieved by specifying a login and password in the `CONNECT TO` instruction. However, most database servers support additional user authentication methods, such as OS user authentication, trusted connections, LDAP authentication, Single Sign-On authentication and even specific pluggable authentication methods.

Follow these simple security patterns to avoid basic user authentication problems:

- Make sure that application files installed on your production server have the appropriate file system permissions set. Regular users should have read-only access to program and resource files. If any OS user can replace a program file with another program, it could harm your database or retrieve sensitive private data.
- Each physical end user must have a specific database account. If several end users connect as the same db application account, they cannot be distinguished in the security and auditing system.
- For normal application users, always use database accounts with the minimum database privileges required to achieve the daily work (GRANT/REVOKE). For example, regular users should not be able to execute Data Definition Language statements (drop tables).
- Instead of asking a name and password in a login dialog when an application starts, some applications hard code the db user names and passwords in the program code, in scripts or configurations files such as FGLPROFILE. This is not a good practice and must be avoided. If a login dialog is not appropriate, you must set up another user authentication method supported by the database server, such as Single Sign-On.

Avoiding SQL injection
Prevent SQL injection attacks in your programs.

SQL injection is a well-known attack that started to appear with Web applications, where the end user enters SQL statement fragments in form fields that are normally designed to hold simple data. When the entered text is used to complete a SQL statement without further checking, there is a risk of SQL statements being injected by the user to intentionally harm the database.

To illustrate the problem, see the following code:

```
MAIN
   DEFINE sql CHAR(200), cn CHAR(50), n INTEGER
   OPEN FORM f FROM "custform"
   DISPLAY FORM f
   INPUT BY NAME cn
   LET sql = "SELECT COUNT(*) FROM customers WHERE custname = '", cn, "'"
   PREPARE stmt FROM sql
   EXECUTE stmt INTO n
   DISPLAY "Count = ", n
END MAIN
```

If the end user enters for example:

```
[xxx' ; delete from customers   ]
```

The resulting SQL statement will contain an additional `DELETE` command that will drop all rows of the *customers* table:

```
SELECT COUNT(*) FROM customers WHERE custname = 'xxx'; DELETE FROM customers
```

In some applications, you may also want to let the end user choose sort columns to be added in an `ORDER BY` clause. The code for such a feature should control the user input. For example, by providing a list of columns to choose from, instead of allowing free text input that will be added to the `ORDER BY` clause.

To avoid SQL injection attacks, do not build SQL instructions dynamically by concatenating user input that is not checked. Instead of basic concatenation, use static SQL statements with program variables (if dynamic SQL is not needed), use parameterized queries (with ? parameter placeholders), or use the `CONSTRUCT` instruction to implement a query by example form.

Simple static SQL example:

```
MAIN
   DEFINE cn CHAR(50), n INTEGER
   OPEN FORM f FROM "custform"
   DISPLAY FORM f
   INPUT BY NAME cn
   SELECT COUNT(*) INTO n FROM customers WHERE custname = cn
   DISPLAY "Count = ", n
END MAIN
```

Parameterized query example:

```
MAIN
   DEFINE sql CHAR(200), cn CHAR(50), n INTEGER
   OPEN FORM f FROM "custform"
   DISPLAY FORM f
   INPUT BY NAME cn
   LET sql = "SELECT COUNT(*) FROM customers WHERE custname = ?"
   PREPARE stmt FROM sql
   EXECUTE stmt USING cn INTO n
   DISPLAY "Count = ", n
END MAIN
```

`CONSTRUCT` example:

```
MAIN
   DEFINE sql CHAR(200), cond CHAR(50), n INTEGER
   OPEN FORM f FROM "custform"
   DISPLAY FORM f
   CONSTRUCT BY NAME cond ON custname
   LET sql = "SELECT COUNT(*) FROM customers WHERE ", cond
   PREPARE stmt FROM sql
   EXECUTE stmt INTO n
   DISPLAY "Count = ", n
END MAIN
```

SQL portability

Writing portable SQL is mandatory if you want to succeed with different kind of database servers. This section gives you some hints to solve SQL incompatibility problems in your programs. Read this section carefully and review your program source code if needed. You should also read carefully the ODI adaptation guides which contain database specific information about SQL compatibility issues.

To easily detect SQL statements with specific syntax, you can use the `-W stdsql` option of `fglcomp`:

```
$ fglcomp -W stdsql orders.4gl
module.4gl:15: SQL Statement or language instruction with specific SQL
 syntax.
```

This compiler option can only detect non-portable SQL syntax in static SQL statements.

- Database entities on page 413
- Database users and security on page 414
- Creating a database from programs on page 414
- Data definition statements on page 416
- Using portable data types on page 416
- Data manipulation statements on page 417
- CHAR and VARCHAR types on page 418
- Concurrent data access on page 422

- Scrollable cursors on page 422
- Optimistic locking on page 423
- Auto-incremented columns (serials) on page 424
- IBM Informix SQL ANSI Mode on page 428
- Positioned updates/deletes on page 428
- WITH HOLD and FOR UPDATE on page 429
- Insert cursors on page 430
- String literals in SQL statements on page 431
- Date and time in SQL statements on page 432
- Naming database objects on page 433
- Temporary tables on page 434
- Outer joins on page 435
- Substring expressions on page 436
- Using ROWIDs on page 437
- MATCHES and LIKE operators on page 438
- GROUP BY clause on page 439
- The LENGTH() function in SQL on page 439
- Transaction savepoints on page 440
- Stored procedures on page 441

Database entities
The database entity concept across different database engines.

Most database servers can handle multiple database entities (you can create multiple 'databases'), but this is not possible with all engines:

Table 139: Multiple database entities by Database server type

Database Server Type	Multiple Database Entities
IBM® DB2® UDB	Yes
IBM® Informix®	Yes
Microsoft™ SQL Server	Yes
MySQL	Yes
Oracle Database Server	No
PostgreSQL	Yes
Sybase ASE	Yes
SQLite	Yes

When using a database server that does not support multiple database entities, you can emulate different databases with schema entities, but this requires you to check for the database user definition. Each database user must have privileges to access any schema, and to see any table of any schema without needing to set a schema prefix before table names in SQL statements.

Some database drivers allow to select a specific schema at connection with the following FGLPROFILE entry:

```
dbi.database.dbname.dbtype.schema = "schema-name"
```

Some databases also allow you to define a default schema for each database user. When the user connects to the database, the default schema is automatically selected.

Database users and security

Properly identifying database users allows to use database security and audit features.

To get the benefit of the database server security features, you should identify each physical user as a database user.

Some applications use a single database user for different end users, to avoid user management and connection issues in the database. This is not good practice because all user-related features of the database are unusable. Further, the single db user often has all database privileges and thus can lead in security issues.

According to the type of server, you must do this steps to create a database user:

1. Define the user as an operating system user.
2. Declare the user in the database server.
3. Grant database access privileges.

Each database server has its specific users management and data access privilege mechanisms. Check the vendor documentation for security features and make sure you can define the users, groups, and privileges in all database servers you want to use.

Creating a database from programs

Creating a database from within a program requires special consideration.

Understanding database creation statements

The Genero language syntax supports database creation statements such as:

```
CREATE DATABASE mydb WITH BUFFERED LOG
```

Such instruction performs an implicit connection to the database server (i.e., no `CONNECT TO` or `DATABASE` is required before a `CREATE DATABASE`), and leads to a default connection.

Creating a database in a database server

When using a database server engine, the creation of a database entity is not a trivial operation. The process usually requires additional tasks such as data storage configuration, database user creation, data access policy, and so on. These tasks are typically left to the database administrator.

Database creation statements such as `CREATE DATABASE`, `CREATE DBSPACE`, and `DROP DATABASE` can be used in programs connected to an IBM® Informix® server, but these statements are not portable. Use database creation statements only for development or testing purpose.

Creating a database on mobile devices (SQLite)

Mobile applications usually create their database at first execution. Database creation on a mobile device is a much simpler operation than database creation on a database server. For example, with SQLite, creating a database only requires creating an empty file.

The SQLite database file must be created in the application sandbox, in a writable directory. This directory is specific to the type of mobile device, and can be found in programs with the `os.Path.pwd` on page `2004` method.

To build the full path to the database file, get the current working directory (`os.Path.pwd()`) and add this path to the database file name. This defines the `source` specification in the database connection parameters, to build the string used for the `CONNECT` instruction:

```
IMPORT os
...
DEFINE dbfile, source, connstr VARCHAR(256)
```

```
FUNCTION init_connection_strings()
   LET dbfile = "contacts.dbs"
   LET source = os.Path.join(os.Path.pwd(), dbfile)
   LET connstr = SFMT("contacts+source='%1'", source)
   IF NOT base.Application.isMobile() THEN
      -- Add db driver spec when in development mode
      LET connstr = connstr, ",driver='dbmsqt'"
   END IF
END FUNCTION
```

If not specified, the `source` connection parameter (i.e., the path to the database file) defaults to the database name specification in the `CONNECT` instruction. Thus, the `source='`*`dbpath`*`'` parameter is usually omitted, and *dbpath* is specified directly as the database name. In this case, however, the identifier of the database connection is the complete path to the SQLite database file. For more details about database connection parameters, see Database connections on page 457.

Before executing the `CONNECT` instruction, check if the database file already exists with `os.Path.exists(source)`. Create the database file and tables only if needed:

```
IMPORT os
...
   CALL init_connection_strings()
   IF os.Path.exists(source) THEN
      CONNECT TO connstr AS "c1"
   ELSE
      CALL create_empty_file(source)
      CONNECT TO connstr AS "c1"
      CALL create_database_tables()
   END IF
   ...

FUNCTION create_empty_file(fn)
   DEFINE fn STRING
   DEFINE ch base.Channel()
   LET ch = base.Channel.create()
   CALL ch.openFile(fn,"w")
   CALL ch.close()
END FUNCTION
```

Instead of creating an empty database file, it is also possible to prepare a template (pre-configured) SQLite database file on the development platform, deploy the template database with the other program files, and copy the template file from the program files directory (`base.Application.getProgramDir` on `page 1705`) into the working directory (`os.Path.pwd` on `page 2004`) on the first application execution (i.e. when the database file in the working directory does not yet exist):

```
IMPORT os
...
   CALL init_connection_strings()
   IF NOT prepare_database("template.dbs", source) THEN
      ERROR "Could not prepare database"
      EXIT PROGRAM 1
   END IF
   CONNECT TO connstr AS "c1"
   ...

FUNCTION prepare_database(template, target)
   DEFINE template, target STRING
   DEFINE tplpath STRING
   IF os.Path.exists(target) THEN
      RETURN TRUE
   END IF
   LET tplpath = os.Path.join(base.Application.getProgramDir(), template)
```

```
   IF NOT os.Path.exists(tplpath) THEN
      ERROR "Database template file not found"
      RETURN FALSE
   END IF
   RETURN os.Path.copy(tplpath, target)
END FUNCTION
```

> **Important:** When creating an initial database file into the working directory from a template file deployed in the program files directory, different file names should be used for the template and actual database file, as folders pointed by `base.Application.getProgramDir` on page 1705 and `os.Path.pwd` on page 2004 could be the same on some devices.

Data definition statements
DDL statements should be avoided in programs.

When using Data Definition Statements like `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, only a limited SQL syntax works on all database servers. Most databases support `NOT NULL`, `CHECK`, `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY` constraints, but the syntax for naming constraints is different.

The following statement works with most database servers and creates a table with equivalent properties in all cases:

```
 CREATE TABLE customer  (
     cust_id INTEGER NOT NULL,
     cust_name CHAR(50) NOT NULL,
     cust_lastorder DATE NOT NULL,
     cust_group INTEGER,
       PRIMARY KEY (cust_id),
       UNIQUE (cust_name),
       FOREIGN KEY (cust_group) REFERENCES group (group_id)
 )
```

Some engines like SQL Server have a different default behavior for `NULL` columns when you create a table. You may need to set up database properties to make sure that a column allows nulls if the `NOT NULL` constraint is not specified.

When you want to create tables in programs using non-standard clauses (for example to define storage options), you must use dynamic SQL and adapt the statement to the target database server.

Using portable data types
Only a limited set of data types are really portable across several database engines.

The ANSI SQL specification defines standard data types, but for historical reasons most databases vendors have implemented native (non-standard) data types. You can usually use a synonym for ANSI types, but the database server will uses the native types behind the scenes. For example, when you create a table with an `INTEGER` column in Oracle, the native `NUMBER` data type is used.

In your programs, avoid data types that do not have a native equivalent in the target database. This includes simple types like floating point numbers, as well as complex data types like `INTERVAL`. Numbers may cause rounding or overflow problems, because the values stored in the database have different limits. For the `DECIMAL` types, always use the same precision and scale for the program variables and the database columns.

To write portable applications, we strongly recommend using the following data types only:

- `CHAR(n)`
- `VARCHAR(n)`
- `BIGINT`
- `INTEGER`
- `SMALLINT`
- `DECIMAL(p,s)`

- DATE
- DATETIME HOUR TO MINUTE
- DATETIME HOUR TO SECOND
- DATETIME YEAR TO MINUTE
- DATETIME YEAR TO FRACTION(n)

Data manipulation statements

Make sure that SQL statement syntaxes are supported by all target database engines.

Several SQL syntaxes for the INSERT, UPDATE and DELETE statements are supported by the compiler. Some of the syntaxes are IBM® Informix® specific, but will be converted to standard SQL at compile time.

The following statements are standard SQL and work with all database servers:

```
(1) INSERT INTO table (column-list) VALUES (value-list)
(2) UPDATE table SET column = value, ... [WHERE condition]
(3) DELETE FROM table [WHERE condition]
```

The next statements are not standard SQL, but are converted by the compiler to standard SQL, working with all database servers:

```
(4) INSERT INTO table VALUES record.*
    -- where record is defined LIKE a table from db schema
(5) UPDATE table SET (column-list) = (value-list) [WHERE condition]
(6) UPDATE table SET {[table.]*|(column-list)}
 = record.* ... [WHERE condition]
   -- where record is defined LIKE a table from db schema
```

The next statement is not standard SQL and will not be converted by the compiler, so it must be reviewed:

```
(7) UPDATE table SET [table.]* = (value-list) [WHERE condition]
```

You can easily search for non-portable SQL statements in your sources by compiling with the `-W stdall` fglcomp option.

For maximum SQL portability, INSERT statements should be reviewed to avoid the SERIAL column from the value list.

For example, the following statement:

```
INSERT INTO tab (col1, col2, ...) VALUES ( 0, p_value2, ... )
```

should be converted to:

```
INSERT INTO tab (col2, ...) VALUES ( p_value2, ... )
```

A static SQL INSERT statement using records defined from the schema file should also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.* )   -- will use the serial column
```

should be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

For more details about supported SQL DML statements, see static SQL.

CHAR and VARCHAR types

Using the CHAR and VARCHAR data types with different sort of databases.

The CHAR and VARCHAR types are designed to store character strings, but all database servers do not have the same semantics for these types.

The behavior of database servers may defer in the following areas related to CHAR/VARCHAR types.

- Byte or Character Length semantics? on page 418
- SQL character type for Unicode/UTF-8 on page 418
- Empty strings and NULLs on page 419
- Trailing blanks in CHAR/VARCHAR on page 419
- What should you do? on page 421

Byte or Character Length semantics?

When defining a CHAR/VARCHAR database column or program variable, you must specify a size. When using a multibyte character set, the unit of this size matters: it can be specified in bytes or characters. In programs, the size unit of CHAR/VARCHAR variables depends on the length semantics defined by the FGL_LENGTH_SEMANTICS environment variable. In databases, the size unit of the CHAR/VARCHAR columns can be expressed in bytes or characters, depending on the database server and its configuration.

SQL character type for Unicode/UTF-8

This section explains database server specifics regarding Unicode / UTF-8 support with character string SQL types.

All database servers can store UNICODE data in character strings types, but there are some specifics you must be aware of. Genero BDL programs typically use the CHAR and VARCHAR types to store UTF-8 strings. But the correspond SQL type may have a different name, according to the database server type. Use the correct SQL type when creating you database tables. When the database uses a different UNICODE codeset as UTF-8 to store the character string data, the database client or the Genero database driver take care of the codeset conversion, as long as the runtime system and database client locale are properly defined.

Table 140: Database server character types for Unicode / UTF-8 data

Database Server Type	Char types to be used for Unicode/UTF-8
IBM® DB2® UDB	CHAR / VARCHAR if the database was created with UTF-8 codeset. Otherwise, you must use GRAPHIC, VARGRAPHIC types. For more details, see CHARACTER data types on page 547.
IBM® Informix®	CHAR / VARCHAR, the database must be created with UTF-8 locale.
IBM® Netezza	NCHAR / NVARCHAR (data always stored in UTF-8). For more details, see CHARACTER data types on page 577.
Microsoft™ SQL Server	NCHAR / NVARCHAR, to store UTF-16 data (drivers make the conversion for application codeset UTF-8) The CHAR/VARCHAR types can only store non-unicode data. For more details, see CHARACTER data types on page 600.

Database Server Type	Char types to be used for Unicode/UTF-8
Oracle MySQL	`CHAR` / `VARCHAR` if the database locale is UTF-8. `NCHAR` / `NVARCHAR` if you need to use the national character set. For more details, see CHARACTER data types on page 630.
Oracle Database Server	`CHAR` / `VARCHAR2` if the database locale is UTF-8. `NCHAR` / `NVARCHAR2` if you need to use the national character set. For more details, see CHARACTER data types on page 653.
PostgreSQL	`CHAR` / `VARCHAR`, the database locale must be UTF-8. For more details, see CHARACTER data types on page 689.
Sybase ASE	`CHAR` / `VARCHAR` if the database locale is UTF-8. `NCHAR` / `NVARCHAR` or `UNICHAR` / `UNIVARCHAR` if you need to use the national character set. For more details, see CHARACTER data types on page 729.
SQLite	`CHAR` / `VARCHAR` (data always stored in UTF-8). For more details, see CHARACTER data types on page 712.

Empty strings and NULLs

At the SQL level, most databases distinguish `''` empty strings from `NULL` (with some exceptions like Oracle DB). However, in programs, an empty string is the equivalent to `NULL` in program variables. As result, it is not possible to distinguish an empty string from a `NULL` when such values are fetched from the database. This limitation is only visible when fetching `VARCHAR` columns and expressions fetched into `VARCHAR` variables, because `CHAR` columns get filled with blanks if the database returns a `CHAR` column value that was filled with an empty string; `CHAR` columns get blanks up to the max size.

Trailing blanks in CHAR/VARCHAR

Trailing blanks in CHAR/VARCHAR database columns

With all kinds of databases servers, `CHAR` columns are always filled with blanks up to the size of the column (this is called blank padding). However, trailing blanks are not significant in comparisons:

```
CHAR('abc ') = CHAR('abc')
```

With all database servers except IBM® Informix®, trailing blanks are significant when comparing `VARCHAR` values:

```
VARCHAR('abc ') != VARCHAR('abc')
```

This is a major issue if you mix CHAR and VARCHAR columns and variables in your SQL statements, because the result of an SQL query can be different depending on whether you are using IBM® Informix® or another database server.

Further, the semantics of the SQL LIKE operator regarding trailing blanks and CHAR/VARCHAR types can differ from database to database. For example, try the following expressions with you database, with a CHAR(5) column containing a row with the value 'abc':

```
CREATE TABLE t1 ( k INT, c CHAR(5), vc VARCHAR(5) )
INSERT INTO t1 VALUES ( 1, 'abc', 'abc' )
SELECT * FROM t1 WHERE c LIKE 'ab_'
SELECT * FROM t1 WHERE vc LIKE 'ab_'
SELECT * FROM t1 WHERE RTRIM(c) LIKE 'ab_'
SELECT * FROM t1 WHERE c LIKE '%c'
SELECT * FROM t1 WHERE vc LIKE '%c'
SELECT * FROM t1 WHERE RTRIM(c) LIKE '%c'
```

See discussion about MATCHES and LIKE operators in adaption guides for more details.

Trailing blanks in CHAR/VARCHAR program variables

In programs, CHAR variables are filled with blanks, even if the value used does not contain all spaces.

The following example:

```
DEFINE c CHAR(5)
LET c = "abc"
DISPLAY c || "."
```

shows the value "abc ." (5 chars + dot).

VARCHAR variables are assigned with the exact value specified, with significant trailing blanks.

For example, this code:

```
DEFINE v VARCHAR(5)
LET v = "abc "
DISPLAY v || "."
```

shows the value "abc ." (4 chars + dot).

Assigning an empty string to a CHAR or VARCHAR variable will set the variable to NULL:

```
DEFINE v VARCHAR(5)
LET v = ""
IF v IS NULL THEN
   DISPLAY "is null"   -- will be displayed
END IF
```

When comparing CHAR or VARCHAR variables in an expression, the trailing blanks are not significant:

```
DEFINE c CHAR(5)
DEFINE v1, v2 VARCHAR(5)
LET c = "abc"
LET v1 = "abc "
LET v2 = "abc  "
IF c == v1 THEN
   DISPLAY "c==v1"
END IF
IF c == v2 THEN
   DISPLAY "c==v2"
END IF
```

```
IF v1 == v2 THEN
   DISPLAY "v1==v2"
END IF
```

All three messages are shown.

Additionally, when you assign a VARCHAR variable from a CHAR, the target variable gets the trailing blanks of the CHAR variable:

```
DEFINE pc CHAR(50)
DEFINE pv VARCHAR(50)
LET pc = "abc"
LET pv = pc
DISPLAY pv || "."
```

"abc <47 spaces>. " (50 chars + dot) is shown.

To avoid this, use the CLIPPED operator:

```
LET pv = pc CLIPPED
```

Trailing blanks in SQL statement parameters

When you insert a row containing a CHAR variable into a CHAR or VARCHAR column, the database interface removes the trailing blanks to avoid overflow problems, (insert CHAR(100) into CHAR(20) when value is "abc" must work).

In this example:

```
DEFINE c CHAR(5)
LET c = "abc"
CREATE TABLE t ( v1 CHAR(10), v2 VARCHAR(10) )
INSERT INTO tab VALUES ( c, c )
```

The value in column v1 and v2 would be "abc" (3 chars in both columns).

When you insert a row containing a VARCHAR variable into a VARCHAR column, the VARCHAR value in the database gets the trailing blanks as set in the variable. When the column is a CHAR(N), the database server fills the value with blanks so that the size of the string is N characters.

In this example:

```
DEFINE vc VARCHAR(5)
LET vc = "abc  "  -- note 2 spaces at end of string
CREATE TABLE t ( v1 CHAR(10), v2 VARCHAR(10) )
INSERT INTO tab VALUES ( vc, vc )
```

The value in column v1 would be "abc " (10 chars) and v2 would be "abc " (5 chars).

What should you do?

Make sure that you have correctly defined the locale and length semantics for your character string data types.

When designing your database tables, consider using CHAR(N) for fixed-length string data (such as codes) and VARCHAR(N) for variable-length string data, such as names, address and comments.

Use VARCHAR variables for VARCHAR columns, and CHAR variables for CHAR columns to achieve portability across all kinds of database servers.

Avoid storing empty strings in VARCHAR columns, or make sure that your program is prepared to get nulls while the database stores empty strings.

Using byte or character length semantics depends mainly on the character set of your application. When using a single-byte character set, keep the default byte length semantics. When using a multibyte character set such as UTF-8, use character length semantics in both the database and the programs. The database column definition and the program variable definition must match, this can be simplified by using a database schema.

Concurrent data access

Understanding concurrent data access and data consistency.

Data concurrency is the simultaneous access of the same data by many users. On the other hand, *data consistency* means that each user sees a consistent view of the database. Without adequate concurrency and consistency controls, data could be changed improperly, compromising data integrity. To write inter-operable applications, you must adapt the program logic to the behavior of the database server regarding concurrency and consistency management. This issue requires good knowledge of multiuser application programming, transactions, locking mechanisms, isolation levels and wait mode. If you are not familiar with these concepts, carefully read the documentation of each database server which covers this subject.

Processes accessing the database can change transaction parameters such as the isolation level. Existing programs might have to be adapted in order to work with this new behavior.

The following is the best configuration to get common behavior with all types of database engines:

- The database must support transactions; this is usually the case.
- Transactions must be as short as possible (under a second is fine, 3 or more seconds is a long transaction).
- The isolation Level should be set to `COMMITTED READ` or `CURSOR STABILITY`. IBM® Informix® IDS 11 has introduced the `LAST COMMITTED` option for the `COMMITTED READ` isolation level, which makes IDS behave like other database server using row-versioning, returning the most recently committed version of the row, rather than wait for a lock to be released. This option can also be turned on implicitly with the `USELASTCOMMITTED` configuration parameter, saving code changes.
- The wait mode for locks must be `WAIT` or `WAIT n` (timeout). Wait mode can be adapted to wait for the longest transaction.

Remarks: With this configuration, the locking granularity does not have to be at the row level. To improve performance with IBM® Informix® databases, you can use the `LOCK MODE PAGE` locking level, which is the default.

Scrollable cursors

How scrollable cursors can be supported on different databases.

Scrollable cursors can be used to go forward and backward in an SQL query result set:

```
DEFINE cust_rec RECORD LIKE customer.*
DECLARE sc SCROLL CURSOR
   FOR SELECT * FROM customer
OPEN sc
FETCH NEXT sc INTO cust_rec.*
FETCH LAST sc INTO cust_rec.*
FETCH FIRST sc INTO cust_rec.*
CLOSE sc
```

This is a useful feature, to implement record set navigation in applications. Scrollable cursors are typically implemented in the database server. But not all database servers support scrollable cursors.

When scrollable cursors are not supported by the target database server, the database driver will emulate it with temporary files.

The temporary files are create in a temporary directory, that can be defined with the DBTEMP environment variable. If DBTEMP is not defined, the default temporary directory dependents from the platform used.

You should consider to avoid scroll cursor usage if the target database does not support this feature:

With emulated scrollable cursors, when scrolling to the last row, all rows will be fetched into the temporary file. This can generate a lot of network traffic and can produce a large temporary file if the result-set contains a lot of rows. Additionally, programs are dependent on the file system resource allocated to the OS user (ulimit).

The following table lists the native scroll cursor availability for each supported database:

Table 141: Database server support for scrollable cursors

Database Server Type	Native scroll cursors?
IBM® DB2® UDB	Yes
IBM® Informix®	Yes
IBM® Netezza	No, emulated by the drivers.
Microsoft™ SQL Server	Yes
Oracle MySQL	No, emulated by the drivers.
Oracle Database Server	Yes
PostgreSQL	Yes
Sybase ASE	Yes
SQLite	No, emulated by the drivers.

Optimistic locking
Implementing optimistic locking to handle access concurrently to the same database records.

This section describes how to implement *optimistic locking* in applications. Optimistic locking is a portable solution to control simultaneous modification of the same record by multiple users.

Traditional IBM® Informix® applications use a `SELECT FOR UPDATE` to set a lock on the row to be edited by the user. This is called *pessimistic locking*. The `SELECT FOR UPDATE` is executed before the interactive part of the code, as described in here:

1. When the end user chooses to modify a record, the program declares and opens a cursor with a `SELECT FOR UPDATE`. At this point, an SQL error might be raised if the record is already locked by another process. Otherwise, the lock is acquired and user can modify the record.
2. The user edits the current record in the input form.
3. The user validates the dialog.
4. The `UPDATE` SQL instruction is executed.
5. The transaction is committed or the `SELECT FOR UPDATE` cursor is closed. The lock is released.

If the IBM® Informix® database was created with transaction logging, you must either start a transaction or define the `SELECT FOR UPDATE` cursor `WITH HOLD` option.

Unfortunately, this is not a portable solution. The lock wait mode should preferably be `WAIT` for portability reasons. Pessimistic locking is based on a `NOT WAIT` mode to return control to the program if a record is already locked by another process. Therefore, following the portable concurrency model, the pessimistic locking mechanisms must be replaced by the optimistic locking technique.

Basically, instead of locking the row before the user starts to modify the record data, the optimistic locking technique makes a copy of the current values (i.e. before modification values (BVM)), lets the user edit the record, and when it's time to write data into the database, checks if the BMVs still correspond to the current values in the database:

1. A `SELECT` is executed to fill the record variable used by the interactive instruction for modifications.
2. The record variable is copied into a backup record to keep Before Modification Values.
3. The user enters modifications in the input form; this updates the values in the modification record.

4. The user validates the dialog.
5. A transaction is started with `BEGIN WORK`.
6. Declare a cursor with a `SELECT FOR UPDATE`, to select the row to be updated.
7. Open the `SELECT FOR UPDATE` cursor and fetch the row into the temporary record.
8. If the SQL status is `NOTFOUND`, the row has been deleted by another process, and the transaction can stop with `ROLLBACK WORK`.
9. If the row was found, the program compares the temporary record values with the backup record values with the `(rec1.*==rec2.*)` notation.
10. If these values have changed, the row has been modified by another process, and the transaction can stop with `ROLLBACK WORK`.
11. If the values in the database have not changed, the `UPDATE` statement is executed to apply the last changes of the user.
12. The transaction is committed with a `COMMIT WORK`.

To compare 2 records (with `NULL` checking), simply write:

```
IF new_record.* != bmv_record.* THEN
   LET values_have_changed = TRUE
END IF
```

The optimistic locking technique could be implemented with a unique SQL instruction: an `UPDATE` could compare the column values to the BMVs directly (`UPDATE ... WHERE kcol = kvar AND col1 = bmv.var1 AND ...`). But, this is not possible when BMVs can be `NULL`. The database engine always evaluates conditional expressions such as "`col=NULL`" to `FALSE`. Therefore, you must use "`col IS NULL`" when the BMV is `NULL`. This means dynamic SQL statement generation based on the DMV values. Additionally, to use the same number of SQL parameters (? markers), you would have to use "`col=?`" when the BMV is not null and "`col IS NULL and ? IS NULL`" when the BMV is null. Unfortunately, the expression " `? IS [NOT] NULL` " is not supported by all database servers (DB2® raises error SQL0418N).

If you are designing a new database application from scratch, you can also use the row versioning method. Each tables of the database must have a column that identifies the current version of the row. The column can be a simple `INTEGER` (to hold a row version number) or it can be a timestamp (`DATETIME YEAR TO FRACTION(5)` for example). To guaranty that the version or timestamp column is updated each time the row is updated, you should implement a trigger to increment the version or set the timestamp when an `UPDATE` statement is issued. If this is in place, you just need to check that the row version or timestamp has not changed since the user modifications started, instead of testing all field of the BMV record. If you are only using one specific database type, you may check if the server supports a versioning column natively. For example, IBM® Informix® IDS 11.50.xC1 introduced the `ALTER TABLE ... ADD VERCOLS` option to get a version + checksum column to a table, you can then query the table with the `ifx_insert_checksum` and `ifx_row_version` columns.

Auto-incremented columns (serials)
How to implement automatic record keys?

IBM® Informix® provides the SERIAL, BIGSERIAL or SERIAL8 data types which can be emulated with database drivers for most non-Informix database engines by using native sequence generators (when "`ifxemul.serial`" FGLPROFILE setting is `true`). But, this requires additional configuration and maintenance tasks. If you plan to review the programming pattern of sequences, you should use a portable implementation instead of the serial emulation provided by the database drivers. This section describes different solutions to implement *auto-incremented fields*. The preferred implementation is the solution using SEQUENCES.

Solution 1: Use database specific serial generators

Principle

In accordance with the target database, you must use the appropriate native serial generation method. Get the database type with the `fgl_db_driver_type()` built-in function and use the appropriate SQL statements to insert rows with serial generation.

This solution uses the native auto-increment feature of the target database and is fast at execution, but is not very convenient as it requires to write different code for each database type. However, it is covered here to make you understand that each database vendor has it's own specific solution for auto-incremented columns. It is of course not realistic to use this solution in a large application with hundreds of tables.

Implementation

1. Create the database objects required for serial generation in the target database (for example, create tables with SERIAL columns in IBM® Informix®, tables with IDENTITY columns in SQL Server and SEQUENCE database objects in Oracle).
2. Adapt your programs to use the native sequence generators in accordance with the database type.

Example

```
DEFINE t1rec RECORD
          id    INTEGER,
          name  CHAR(50),
          cdate DATE
    END RECORD

CASE fgl_db_driver_type()
 WHEN "ifx"
   INSERT INTO t1 ( id, name, cdate )
          VALUES ( 0, t1rec.name, t1rec.cdate )
   LET t1rec.id = SQLCA.SQLERRD[2]
 WHEN "ora"
   INSERT INTO t1 ( id, name, cdate )
          VALUES ( t1seq.nextval, t1rec.name, t1rec.cdate )
   SELECT t1seq.currval INTO t1rec.id FROM dual
 WHEN "msv"
   INSERT INTO t1 ( name, cdate )
           VALUES ( t1rec.name, t1rec.cdate )
   PREPARE s FROM "SELECT @@IDENTITY"
   EXECUTE s INTO t1rec.id
END CASE
```

As you can see in this example, this solution requires database engine specific coding. Querying the last generated serial can be centralized in a function, but the insert statements would still need to be specific to the type of database.

Solution 2: Generate serial numbers from your own sequence table

Purpose

The goal is to generate unique `INTEGER` or `BIGINT` numbers. These numbers will usually be used for primary keys.

Prerequisites

1. The database must use transactions. This is usually the case with non-INFORMIX databases, but IBM® Informix® databases default to auto commit mode. Make sure your IBM® Informix® database allows transactions.
2. The sequence generation must be called inside a transaction (`BEGIN WORK` / `COMMIT WORK`).
3. The transaction isolation level must guarantee that a row UPDATEd in a transaction cannot be read or written by other db sessions until the transaction has ended (typically, `COMMITTED READ` is ok, but some db servers require a higher isolation level)
4. The lock wait mode must be `WAIT`. This is usually the case in non-INFORMIX databases, but INFORMIX defaults to `NOT WAIT`. You must change the lock wait mode with "`SET LOCK MODE TO WAIT`" or "`WAIT` *`seconds`*" when using IBM® Informix®.
5. Other applications or stored procedures must implement the same technique when inserting records in the table having auto-incremented columns.

Principle

A dedicated table named "`SEQREG`" is used to register sequence numbers. The key is the name of the sequence. This name will usually be the table name the sequence is generated for. In short, this table contains a primary key that identifies the sequence and a column containing the last generated number.

The uniqueness is granted by the concurrency management of the database server. The first executed instruction is an `UPDATE` that sets an exclusive lock on the `SEQREG` record. When two processes try to get a sequence at the same time, one will wait for the other until its transaction is finished.

Implementation

The "`fgldbutl.4gl`" utility library implements a function called "`db_get_sequence()`" which generates a new sequence. You must create the `SEQREG` table as described in the `fgldbutl.4gl` source found in `FGLDIR/src`, and make sure that every user has the privileges to access and modify this table.

In order to guarantee the uniqueness of the generated number, the call to `db_get_sequence()` must be done inside a transaction block that includes the `INSERT` statement. Concurrent db sessions must wait for each other in case of conflict and the transaction isolation level must be high enough to make sure that the row of the sequence table will not be read or written by other db sessions until the transaction end.

Example

```
IMPORT FGL fgldbutl
DEFINE rec RECORD
           id    INTEGER,
           name  CHAR(100)
     END RECORD
...
BEGIN WORK
LET rec.id = db_get_sequence( "CUSTID" )
INSERT INTO CUSTOMER ( CUSTID, CUSTNAME ) VALUES ( rec.* )
COMMIT WORK
```

Solution 3: Use native SEQUENCE database objects

Principle

Most recent database engines support `SEQUENCE` database objects; If all database server types you want to use do support sequences, you should use this solution.

Implementation

1. Create a `SEQUENCE` object for each table using previously a `SERIAL` column in the IBM® Informix® database.
2. In database creation scripts (`CREATE TABLE`), replace all `SERIAL` types by `INTEGER` (or `BIGINT` if you need large integers).
3. Adapt your programs to retrieve a new sequence before inserting a new row. Consider writing a function to retrieve a new sequence number, using dynamic SQL to pass the name of the sequence as parameter, and adapt to the target database specifics to retrieve a single row (see example below).

Example

```
MAIN
  DEFINE item_rec RECORD
           item_num BIGINT,
           item_name VARCHAR(40)
         END RECORD
  DEFINE i INT
  DATABASE test1
  CREATE TABLE item (
           item_num BIGINT NOT NULL PRIMARY KEY,
           item_name VARCHAR(50)
          )
  CALL sequence_create("item")
  LET item_rec.item_num = sequence_next("item")
  DISPLAY "New sequence: ", item_rec.item_num
  LET item_rec.item_name = "Item#" || item_rec.item_num
  INSERT INTO item VALUES ( item_rec.* )
  DROP TABLE item
  DROP SEQUENCE item_seq
END MAIN

PRIVATE FUNCTION is_sql_server()
  RETURN (fgl_db_driver_type()=="esm" OR
 fgl_db_driver_type()=="snc")
END FUNCTION

FUNCTION sequence_create(tabname)
  DEFINE tabname STRING
  IF is_sql_server() THEN
     EXECUTE IMMEDIATE "CREATE SEQUENCE item_seq START WITH 1"
  ELSE
     CREATE SEQUENCE item_seq
  END IF
END FUNCTION

FUNCTION sequence_next(tabname)
  DEFINE tabname STRING
  DEFINE sql STRING, newseq BIGINT
  CASE
     WHEN fgl_db_driver_type()=="pgs"
          LET sql = "SELECT nextval('"||tabname||"_seq')"||
unique_row_condition()
     WHEN is_sql_server()
          LET sql = "SELECT NEXT VALUE FOR "||tabname||"_seq"
     OTHERWISE
          LET sql = "SELECT "||tabname||"_seq.nextval "||
unique_row_condition()
  END CASE
  PREPARE seq FROM sql
  IF SQLCA.SQLCODE!=0 THEN RETURN -1 END IF
  EXECUTE seq INTO newseq
```

```
  IF SQLCA.SQLCODE!=0 THEN RETURN -1 END IF
  RETURN newseq
END FUNCTION

FUNCTION unique_row_condition()
    CASE fgl_db_driver_type()
        WHEN "ifx" RETURN " FROM systables WHERE tabid=1"
        WHEN "db2" RETURN " FROM sysibm.systables WHERE
 name='SYSTABLES'"
        WHEN "pgs" RETURN " FROM pg_class WHERE
 relname='pg_class'"
        WHEN "ora" RETURN " FROM dual"
        OTHERWISE RETURN " "
    END CASE
END FUNCTION
```

IBM® Informix® SQL ANSI Mode
Understanding the impact of the SQL ANSI mode of IBM® Informix®.

IBM® Informix® allows you to create databases in ANSI mode, which is supposed to be closer to ANSI standard behavior. Other databases like ORACLE and DB2® are 'ANSI' by default.

If you are not using the ANSI mode with IBM® Informix®, we suggest you keep the database as is, because turning an IBM® Informix® database into ANSI mode can result in unexpected behavior of the programs.

Here are some ANSI mode issues extracted from the IBM® Informix® books:

- Some actions, like `CREATE INDEX` will generate a warning but will not be forbidden.
- Buffered logging is not allowed to enforce data recovery. (Buffered logging provides better performance)
- The table-naming scheme allows different users to create tables without having to worry about name conflicts.
- Owner specification is required in database object names (`SELECT ... FROM "owner".table`). You must quote the owner name to prevent automatic translation of the owner name into uppercase: `SELECT ... FROM owner.table` becomes `SELECT .. FROM OWNER.table` and thus, the table is not found in the database.
- Default privileges differ: When creating a table, the server grants privileges to the table owner and the DBA only. The same thing happens for the 'Execute' privilege when creating stored procedures.
- Default isolation level is `REPEATABLE READ`.
- An error is generated if any character field is filled with a value that is longer than the field width.
- `DECIMAL(p)` (floating point decimals) are automatically converted to `DECIMAL(p,0)` (fixed point decimals).
- Closing a closed cursor generates an SQL error.

It will take more time to adapt the programs to the IBM® Informix® ANSI mode than using the database interface to simulate the native mode of IBM® Informix®.

Positioned updates/deletes
Using positioned updates/deletes with named database cursors.

The "`WHERE CURRENT OF` *`cursor-name`*" clause in `UPDATE` and `DELETE` statements is not supported by all database engines.

Table 142: Database server support of WHERE CURRENT OF

Database Server Type	WHERE CURRENT OF supported?
IBM® DB2® UDB	Yes

Database Server Type	WHERE CURRENT OF supported?
IBM® Informix®	Yes
Microsoft™ SQL Server	Yes
MySQL	Yes
Oracle Database Server	No, emulated by driver with ROWIDs
PostgreSQL	No, emulated by driver with OIDs
Sybase ASE	Yes
SQLite	No

Some database drivers can emulate `WHERE CURRENT OF` mechanisms by using rowids, but this requires additional processing. You should review the code to disable this option.

The standard SQL solution is to use primary keys in all tables and write `UPDATE` / `DELETE` statements with a `WHERE` clause based on the primary key:

```
DEFINE rec RECORD
           id    INTEGER,
           name  CHAR(100)
     END RECORD
BEGIN WORK
  SELECT CUSTID FROM CUSTOMER
         WHERE CUSTID=rec.id FOR UPDATE
  UPDATE CUSTOMER SET CUSTNAME = rec.name
        WHERE CUSTID = rec.id
COMMIT WORK
```

WITH HOLD and FOR UPDATE
Hold cursors and not portable.

IBM® Informix® supports `WITH HOLD` cursors using the `FOR UPDATE` clause. Such cursors can remain open across transactions (when using `FOR UPDATE`, locks are released at the end of a transaction, but the `WITH HOLD` cursor is not closed). This kind of cursor is IBM® Informix-specific and not portable. The SQL standards recommend closing `FOR UPDATE` cursors and release locks at the end of a transaction. Most database servers close `FOR UPDATE` cursors when a `COMMIT WORK` or `ROLLBACK WORK` is done. All database servers release locks when a transaction ends.

Table 143: Database server support of WITH HOLD FOR UPDATE

Database Server Type	WITH HOLD FOR UPDATE supported?
IBM® DB2® UDB	No
IBM® Informix®	Yes
Microsoft™ SQL Server	No
MySQL	No
Oracle Database Server	No
PostgreSQL	No
Sybase ASE	No
SQLite	No

It is mandatory to review code using `WITH HOLD` cursors with a `SELECT` statement having the `FOR UPDATE` clause.

The standard SQL solution is to declare a simple `FOR UPDATE` cursor outside the transaction and open the cursor inside the transaction:

```
DECLARE c1 CURSOR FOR SELECT ... FOR UPDATE
BEGIN WORK
  OPEN c1
  FETCH c1 INTO ...
  UPDATE ...
COMMIT WORK
```

If you need to process a complete result set with many rows including updates of master and detail rows, first fetch the primary keys of all master rows into a program array, declare a cursor with the `SELECT FOR UPDATE`, then for all rows in the array, start a transaction and perform the `UPDATE WHERE CURRENT OF` for the current master record and the `UPDATE` for detail rows, then commit the transaction and continue with the next master record:

```
DEFINE x, mkeys DYNAMIC ARRAY OF INTEGER
DECLARE c1 CURSOR FOR SELECT key FROM master ...
FOREACH c1 INTO x
    LET mkeys[mkeys.getLength()+1] = x
END FOREACH
DECLARE c2 CURSOR FOR SELECT * FROM master WHERE key=? FOR UPDATE
FOR x = 1 TO mkeys.getLength()
  BEGIN WORK
  OPEN c2 USING mkeys[x]
  FETCH c2 INTO mrec.*
  IF STATUS==NOTFOUND THEN
     ROLLBACK WORK
     CONTINUE FOREACH
  END IF
  UPDATE master SET ... WHERE CURRENT OF c2
  UPDATE detail SET ... WHERE master_key=mkeys[x]
  COMMIT WORK
END FOR
```

Insert cursors
Using insert cursors with non-Informix databases.

Database cursors defined with "`DECLARE` *`cursor-name`* `CURSOR FOR INSERT ...`" are designed for IBM® Informix® databases, to optimize row insertion when a lot of data must be loaded in the table.

This is an IBM® Informix® specifc feature. With non-Informix databases, insert cursors are emulated by executing the INSERT

```
DEFINE rec RECORD
           id    INTEGER,
           name  CHAR(100)
     END RECORD,
     i INTEGER
DECLARE c1 CURSOR FOR INSERT INTO customer VALUES (?,?)
BEGIN WORK
  OPEN c1
  FOR i=1 TO 100
      LET rec.id = i
      LET rec.name = "name"||i
      PUT c1 FROM rec.*
  END FOR
  FLUSH c1
  CLOSE c1
```

```
COMMIT WORK
```

Insert cursors are an IBM® Informix® specifc feature. The IBM® Informix® insert cursors buffers the provided rows and flushes blocks of rows into the database after a given number of rows, or when the program explicitly executes a `FLUSH` or `CLOSE`. In can of errors, for example when inserting a character string value for a numeric column, the SQL error is returned at "flush time" with Informix®.

With non-Informix databases, the rows are not buffered: insert cursors are emulated in db drivers by executing the `INSERT` statement on every `PUT` instruction. As result, this can lead to poor performances, and SQL errors can be returned earlier at `PUT` time.

Note that the `LOAD` instruction is based on an insert cursor. The same performance issue applies to the `LOAD` instruction when using a non-Informix database.

If you need to feed your database with a lot of data, coming for example from external sources, we recommend to use database vendor specific tools to load the data. This option is much more efficient as using a Genero program to load data.

String literals in SQL statements
Single quotes is the standard for delimiting string literals in SQL.

Some database servers like IBM® Informix® allow single and double quoted string literals in SQL statements, both are equivalent:

```
SELECT COUNT(*) FROM table
 WHERE col1 = "abc'def""ghi"
   AND col1 = 'abc''def"ghi'
```

Most database servers do not support this specific feature.

Table 144: Database servers support of double-quoted string literals

Database Server Type	Double quoted string literals
IBM® DB2® UDB	No
IBM® Informix®	Yes
Microsoft™ SQL Server	Yes
MySQL	No
Oracle Database Server	No
PostgreSQL	No
Sybase ASE	No
SQLite	Yes

The ANSI SQL standards define doubles quotes as database object names delimiters, while single quotes are dedicated to string literals:

```
CREATE TABLE "my table" ( "column 1" CHAR(10) )
SELECT COUNT(*) FROM "my table" WHERE "column 1" = 'abc'
```

If you want to write a single quote character inside a string literal, you must write 2 single quotes:

```
... WHERE comment = 'John''s house'
```

When writing static SQL in your programs, the double quoted string literals as converted to ANSI single quoted string literals by the fglcomp compiler. However, dynamic SQL statements are not parsed by the compiler and therefore need to use single quoted string literals.

We recommend that you always use single quotes for string literals and, if needed, double quotes for database object names.

Date and time in SQL statements
Good practices for date and time handling in SQL.

Date and time strings in SQL Statements

IBM® Informix® allows you to specify date and time values as a quoted character string in a specific format, depending upon DBDATE and GLS environment variables. For example, if `DBDATE=DMY4`, the following statement specifies a valid `DATE` represented by a string literal:

```
SELECT COUNT(*) FROM table WHERE date_col = '24/12/2005'
```

Other database servers do support date/time literals as quoted character strings, but the date/time format specification is quite different. The parameter to specify the date/time format can be a database parameter, an environment variable, or a session option.

In order to write portable SQL, use SQL parameters instead of string literals for date-time values:

```
DEFINE cnt INTEGER
DEFINE adate DATE
LET adate = MDY(12,24,2005)
SELECT COUNT(*) INTO cnt FROM table
    WHERE date_col = adate
```

Or, when using dynamic SQL:

```
DEFINE cnt INTEGER
DEFINE adate DATE
LET adate = MDY(12,24,2005)
PREPARE s1 FROM "SELECT COUNT(*) FROM table WHERE date_col = ?"
EXECUTE s1 USING adate INTO cnt
```

Date-time literals

IBM® Informix® `DATETIME` and `INTERVAL` literals are not converted automatically by the SQL translator of the database driver:

```
SELECT COUNT(*) FROM order WHERE ord_when > DATETIME (1999-10-12) YEAR TO
 DAY
```

Check your code, to detect where you are using such expressions in the SQL statements, and use an SQL parameter instead.

Informix-specific keywords

SQL Statements using expressions such as TODAY, CURRENT and EXTEND must be reviewed and adapted to the native syntax of the target database engine.

Check your code, to detect where you are using such expressions in the SQL statements.

Date-time expressions with parameters

Date-time arithmetic expressions using SQL parameters (USING variables) are not portable.

For example:

```
PREPARE s1 FROM "SELECT ... WHERE datecol < ? + 1"
```

Might generate an error with non-Informix databases.

DATEs as a number of days

IBM® Informix® can automatically convert integers to a DATE values, as a number of days since 12/31/1899 (1 = 01/01/1900). This is however not supported by other database engines.

Check your code, to detect where you are using integers with DATE columns.

Naming database objects

Name syntax

Database object naming conventions are different for each database engine.

The table below describes the naming conventions for database objects (i.e. tables, sequences, stored procedures):

Table 145: Database server naming conventions for database objects

Database Server Type	Naming Syntax
IBM® DB2® UDB	`[[database.]owner.]identifier`
IBM® Informix®	`[database[@dbservername]:][owner.]identifier`
Microsoft™ SQL Server	`[[[server.][database].][owner_name].]object_name`
MySQL	`[database.]identifier`
Oracle Database Server	`[schema.]identifier[@database-link]`
PostgreSQL	`[owner.]identifier`
Sybase ASE	`[database.]identifier`
SQLite	`[database.]identifier`

Case-sensitivity

Handling case-sensitivity with different database engines.

Most database engines have case-sensitive object identifiers. In most cases, when you do not specify identifiers in double quotes, the SQL parser automatically converts names to uppercase or lowercase, so that the identifiers match if the objects are also created without double quoted identifiers.

```
CREATE TABLE Customer ( cust_ID INTEGER )
```

In ORACLE, this statement would create a table named "`CUSTOMER`" with a "`CUST_ID`" column.

This table shows the behavior of each database engine regarding case sensitivity and double quoted identifiers:

Table 146: Database server support of case sensitivity and double-quoted identifiers

Database Server Type	Un-quoted names	Double-quoted names
IBM® DB2® UDB	Converts to uppercase	Case sensitive
IBM® Informix® (1)	Converts to lowercase	Syntax disallowed (non-ANSI mode)
Microsoft™ SQL Server (2)	Not converted, kept as is	Case sensitive
MySQL	Not converted, kept as is	Syntax disallowed
Oracle Database Server	Converts to uppercase	Uppercase
PostgreSQL	Converts to lowercase	Lowercase
Sybase ASE	Converts to lowercase	Lowercase
SQLite	Not converted, kept as is	Case insensitive

(1) If not ANSI database mode.

(2) When case-sensitive charset/collation used.

Take care with database servers marked in red, because object identifiers are case sensitive and are not converted to uppercase or lowercase if not delimited by double-quotes. This means that, by error, you can create two tables with a similar name:

```
CREATE TABLE customer ( cust_id INTEGER )  -- first table
CREATE TABLE Customer ( cust_id INTEGER )  -- second table
```

It is recommended to design databases with lowercase table and column names.

Size of identifiers
Avoid using long database object names.

The maximum size of a table or column name depends on the database server type. Some database engines allow very large names (256c), while others support only short names (30c max). Therefore, using short names is required for writing portable SQL. Short names also simplify SQL programs.

We recommend that you use simple and short (<30c) database object names, without double quotes and without a schema/owner prefix:

```
CREATE TABLE customer ( cust_id INTEGER )
SELECT customer.cust_id FROM table
```

You may need to set the database schema after connection, so that the current database user can see the application tables without specifying the owner/schema prefix each time.

> **Tip:** Even if all database engines do not required unique column names for all tables, we recommend that you define column names with a small table prefix (for example, CUST_ID in CUSTOMER table).

Temporary tables
Syntax for temporary table creation is not unique across all database engines.

Not all database servers support temporary tables. The engines supporting this feature often provide it with a specific table creation statement:

Table 147: Database server support of temporary tables

Database Server Type	Temp table creation syntax	Local to SQL session?
IBM® DB2® UDB	`DECLARE GLOBAL TEMPORARY TABLE tablename ( column-defs )` `DECLARE GLOBAL TEMPORARY TABLE tablename AS ( SELECT ... )`	Yes
IBM® Informix®	`CREATE TEMP TABLE tablename ( column-defs )` `SELECT ... INTO TEMP tablename`	Yes
Microsoft™ SQL Server	`CREATE TABLE #tablename ( column-defs )` `SELECT select-list INTO #tablename FROM ...`	Yes
MySQL	`CREATE TEMPORARY TABLE tablename ( column-defs )` `CREATE TEMPORARY TABLE tablename LIKE other-table`	Yes
Oracle Database Server	`CREATE GLOBAL TEMPORARY TABLE tablename ( column-defs )` `CREATE GLOBAL TEMPORARY TABLE tablename AS SELECT ...`	No: only data is local to session
PostgreSQL	`CREATE TEMP TABLE tablename ( column-defs )` `SELECT select-list INTO TEMP tablename FROM ...`	Yes
Sybase ASE	`CREATE TABLE #tablename ( column-defs )` `SELECT select-list INTO #tablename FROM ...`	Yes
SQLite	`CREATE TEMP TABLE tablename ( column-defs )`	Yes

Some databases even have a different behavior when using temporary tables. For example, ORACLE 9i supports a kind of temporary table, but it must be created as a permanent table. The table is not specific to an SQL session: it is shared by all processes - only the data is local to a database session.

You must review the programs using temporary tables, and adapt the code to use database-specific temporary tables.

Outer joins

Use standard ISO outer join syntax instead of the old IBM® Informix® `OUTER()` syntax.

Old IBM® Informix® SQL outer joins specified with the `OUTER` keyword in the `FROM` part are not standard:

```
SELECT * FROM master, OUTER ( detail )
  WHERE master.mid = detail.mid
    AND master.cdate IS NOT NULL
```

Table 148: Database server support of OUTER JOIN syntax

Database Server Type	Supports IBM® Informix® OUTER join syntax
IBM® DB2® UDB	No (but translated by driver)
IBM® Informix® (1)	Yes
Microsoft™ SQL Server (2)	No (but translated by driver)
MySQL	No (but translated by driver)
Oracle Database Server	No (but translated by driver)
PostgreSQL	No (but translated by driver)
Sybase ASE	No (but translated by driver)
SQLite	No (but translated by driver)

Most recent database servers now support the standard ANSI outer join specification:

```
SELECT * FROM master LEFT OUTER JOIN detail ON (master.mid = detail.mid)
  WHERE master.cdate IS NOT NULL
```

You should use recent database servers and use ANSI outer joins only.

Substring expressions
Handle substrings expressions with different database engines.

Only IBM® Informix® supports substring specification with square brackets:

```
SELECT * FROM item WHERE item_code[1,4] = "XBFG"
```

However, most database servers support a function that extracts substrings from a character string:

Table 149: Database server support of extraction of substrings

Database Server Type	Supports col[x,y] substrings?	Provides substring function?
IBM® DB2® UDB	No	`SUBSTR(expr,start,length)`
IBM® Informix® (1)	Yes	`SUBSTR(expr,start,length)`
Microsoft™ SQL Server (2)	No	`SUBSTRING(expr,start,length)`
MySQL	No	`SUBSTR(expr,start,length)`
Oracle Database Server	No	`SUBSTRING(expr,start,length)`
PostgreSQL	No	`SUBSTRING(expr FROM start FOR length )`
Sybase ASE	No	`SUBSTRING(expr,start,length)`
SQLite	No	`SUBSTR(expr,start,length)`

Informix® allows you to update some parts of a CHAR and VARCHAR column by using the substring specification (`UPDATE tab SET col[1,2] ='ab'`). This is not possible with other databases.

Review the SQL statements using substring expressions and use the database specific substring function.

You could also create your own SUBSTRING() user function in all databases that do not support this function, to have a common way to extract substrings. In Microsoft™ SQL Server, when you create a user function, you must specify the owner as prefix when using the function. Therefore, you should create a SUBSTRING() user function instead of SUBSTR().

Using ROWIDs

Automatic ROWIDs is not a common database feature.

Rowids are implicit primary keys generated by the database engine. Not all database servers support rowids:

Table 150: Database server support of rowid

Database Server Type	Rowid keyword?	Rowid type?
IBM® DB2® UDB	none	none
IBM® Informix® (1)	ROWID	INTEGER
Microsoft™ SQL Server (2)	none	none
MySQL	none	none
Oracle Database Server	ROWID	CHAR(18)

Database Server Type	Rowid keyword?	Rowid type?
PostgreSQL	OID	internal type
Sybase ASE	none	none
SQLite	ROWID	BIGINT

Informix® fills the `SQLCA.SQLERRD[6]` register with the ROWID of the last updated row. This register is an INTEGER and cannot be filled with rowids having CHAR(*) type.

Search for ROWID and SQLCA.SQLERRD[6] in your code and review the code to remove the usage of rowids.

MATCHES and LIKE operators
Use the standard LIKE operator instead of the MATCHES operator.

The MATCHES operator allows you to scan a string expression:

```
SELECT * FROM customer WHERE customer_name MATCHES "A*[0-9]"
```

Here is a table listing the database servers which support the MATCHES operator:

Table 151: Database server support for MATCHES operator

Database Server Type	Support for SQL MATCHES operator?
IBM® DB2® UDB	No
IBM® Informix® (1)	Yes
Microsoft™ SQL Server (2)	No
MySQL	No
Oracle Database Server	No
PostgreSQL	No
Sybase ASE	No
SQLite	No

The `MATCHES` operator is specific to IBM® Informix® SQL. The equivalent standard operator is `LIKE`. For maximum portability, replace `MATCHES` expressions in your SQL statements with a standard `LIKE` expression. `MATCHES` uses `*` and `?` as wildcards. The equivalent wildcards in the `LIKE` operator are `%` and `_`. Character ranges `[a-z]` are not supported by the `LIKE` operator.

Note that the Genero language includes a `MATCHES` operator. For example, in expressions such as: `IF custname MATCHES "S*"`. Do not confuse the language `MATCHES` operator with the SQL `MATCHES` operator, used in SQL statements. There is no problem in using the `MATCHES` operator of the language.

A program variable can be used as parameter for the `MATCHES` or `LIKE` operator, but you must pay attention to blank padding semantics of the target database. If the program variable is defined as a `CHAR(N)`, it is filled by the runtime system with trailing blanks, in order to have a size of N. For example, when a `CHAR(10)` variable is assigned with `"ABC%"`, it contains actually `"ABC% "` (with 6 additional blanks). If this variable is used in a `LIKE` expression in an SQL statement, the database server will search for column values matching "ABC"+ some characters + 6 blanks. To avoid automatic blanks, use a `VARCHAR(N)` data type instead of `CHAR(N)` to hold LIKE patterns.

Pay also attention to database specific semantics of the `LIKE` operation, especially when using CHAR(N) data types. For example, with Oracle DB, the expression `custname LIKE '%h'`, if custname is defined as CHAR(30), Oracle will only find the rows when the custname values end with a `'h'` at the last character

position (30), values such as `'Smith'` will not be found. Similarly, when doing `custname LIKE 'ab_'`, rows where the column type is CHAR(N>3), with values such as `'abc'` will not match in Oracle, IBM® DB2® and PostgreSQL, because of the significant trailing blanks.

As a general advice, use the VARCHAR type for variable string data, and leave CHAR usage for fixed-length character string data such as codes.

GROUP BY clause

Some databases allow you to specify a column index in the GROUP BY clause:

```
SELECT a, b, sum(c) FROM table GROUP BY 1,2
```

This is not possible with all database servers:

Table 152: Database server support of GROUP BY column index

Database Server Type	GROUP BY colindex, ... ?
IBM® DB2® UDB	No
IBM® Informix® (1)	Yes
Microsoft™ SQL Server (2)	No
MySQL	Yes
Oracle Database Server	No
PostgreSQL	Yes
Sybase ASE	No
SQLite	Yes

Search for GROUP BY in your SQL statements and use explicit column names.

The LENGTH() function in SQL

The semantics of the LENGTH() SQL function differs according to the database engine.

The SQL LENGTH() function must be used with care: Each database server has different semantics for this function, regarding length and trailing blanks handling.

> **Note:** The language provides a LENGTH built-in function which is part of the runtime system. Do not confuse this with the SQL LENGTH() function, used in SQL statements. The LENGTH() function of the language returns zero when the string expression is NULL.

Table 153: Database server support of LENGTH()

Database Server Type	Function name	Counting unit	Significant trailing blanks for CHAR() columns	Return value when NULL
IBM® DB2® UDB	`LENGTH(expr)`	Octets	Yes	NULL
IBM® Informix® (1)	`LENGTH(expr)`	Octets	No	NULL
Microsoft™ SQL Server (2)	`LEN(expr)`	Characters	No	NULL
MySQL	`LENGTH(expr)`	Characters	No	NULL
Oracle Database Server	`LENGTH(expr)`	Characters	Yes	NULL
PostgreSQL	`LENGTH(expr)`	Characters	Yes	NULL
Sybase ASE (2)	`LEN(expr)`	Characters	No	NULL
SQLite	`LENGTH(expr)`	Characters	Yes	NULL

Search for LENGTH() usage in your SQL statements and review the code of the database-specific function.

Transaction savepoints
Using transaction savepoints with different database engines.

IBM® Informix® IDS 11.50 introduced transaction savepoints, following the ANSI SQL standards. While most recent database severs support savepoints, you must pay attention and avoid Informix® specific features. For example, Oracle (11), SQL Server (2008 R2), Sybase ASE (15.5) do not support the `RELEASE SAVEPOINT` instruction. The `UNIQUE` clause of `SAVEPOINT` is only supported by IBM® Informix® and IBM® DB2® UDB.

Database Server Type	SAVEPOINT & ROLLBACK WORK TO SAVEPOINT	RELEASE SAVEPOINT	SAVEPOINT UNIQUE
IBM® DB2® UDB	Yes	Yes	Yes
IBM® Informix®	Yes	Yes	Yes
Microsoft™ SQL Server (Only 2005+ with SNC driver)	Yes	No	No
MySQL	Yes	Yes	No
Oracle Database Server	Yes	No	No
PostgreSQL	Yes	Yes	No

Database Server Type	SAVEPOINT & ROLLBACK WORK TO SAVEPOINT	RELEASE SAVEPOINT	SAVEPOINT UNIQUE
Sybase ASE	Yes	No	No
SQLite	Yes	Yes	No

Stored procedures
Executing stored procedures with different database engine types.

Stored procedures execution needs to be addressed specifically according to the database type. There are different ways to execute a stored procedure. This section describes how to execute stored procedures on the supported database engines.

> **Tip:** In order to write reusable code, you should encapsulate each stored procedure execution in a `FUNCTION` performing database-specific SQL based on a global database type variable. The program function would just take the input parameters and return the output parameters of the stored procedure, hiding database-specific execution steps from the caller.

Specifying input and output parameters

Input and ouput parameters can be specified in SQL statement execution to pass and return values to/from stored procedures, according to the database type:

```
EXECUTE stmt USING param1 IN, param2 INOUT, param3 INOUT
```

Stored procedures returning a result set

With some database servers it is possible to execute stored procedures that produce a result set, and fetch the rows as normal `SELECT` statements, by using `DECLARE`, `OPEN`, `FETCH`. Some databases can return multiple result sets and cursor handles declared in a stored procedure as output parameters, but Genero supports only unique and anonymous result sets. See the examples.

Calling stored procedures with supported databases

- Stored procedure call with IBM Informix on page 441
- Stored procedure call with Oracle DB on page 443
- Stored procedure call with IBM DB2 on page 444
- Stored procedure call with Microsoft SQL Server on page 445
- Stored procedure call with PostgreSQL on page 447
- Stored procedure call with Oracle MySQL on page 449

Stored procedure call with IBM® Informix®

IBM® Informix® stored procedures are written in the SPL, C or Java™ programming languages, also known as User Defined Routines.

See Informix IDS documentation for more details.

Stored functions returning values

To return values from an IBM® Informix® SPL routine, execute the routine and fetch the output values, as you would for a regular `SELECT` statement producing a result set.

> **Note:** Informix distinguishes between *stored functions* from *stored procedures*. Only stored functions (with a `RETURNING` clause) can return values. Stored procedures do not return values.

To execute an Informix stored function from a BDL program, use the `EXECUTE FUNCTION` SQL instruction:

```
PREPARE stmt FROM "execute function proc1(?)"
```

In order to retrieve returning values into program variables, use an `INTO` clause in the `EXECUTE` instruction.

This example shows how to call a stored function:

```
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE "create function proc1( p1 integer )"
               || " returning decimal(6,2), varchar(200);"
               || "  define p2 decimal(6,2);"
               || "  define p3 varchar(200);"
               || "  let p2 = p1 + 0.23;"
               || "  let p3 = 'Value = ' || p1;"
               || "  return p2, p3;"
               || " end function;"
   PREPARE stmt FROM "execute function proc1(?)"
   LET n = 111
   EXECUTE stmt USING n INTO d, c
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored functions defined with output parameters

Starting with IDS 10.00, IBM® Informix® introduced the concept of output parameters for stored functions.

To retrieve the output parameters, you must execute the routine in a `SELECT` statement defining *Statement Local Variables*. These variables will be listed in the select clause to be fetched as regular column values. See Informix documentation for more details.

In order to retrieve returning values into program variables, use an `INTO` clause in the `EXECUTE` instruction.

This example shows how to call a stored function with output parameters:

```
MAIN
    DEFINE pi, pr INTEGER
    DATABASE test1
    EXECUTE IMMEDIATE "create function proc2(i INT, OUT r INT)"
                    ||" returning int;"
                    ||" let r=i+10;"
                    ||" return 1;"
                    ||" end function"
    PREPARE s FROM "select r from systables where tabid=1 and
 proc2(?,r#int)==1"
    LET pi = 33
    EXECUTE s USING pi INTO pr
    DISPLAY "Output value: ", pr
    EXECUTE IMMEDIATE "drop function proc2"
END MAIN
```

Stored functions returning a result set

To retrieve the rows of a result set produced by an IBM® Informix® stored function, you must create a cursor, as you would for a regular `SELECT` statement.

This example shows how to execute a stored function producing a result set:

```
MAIN
    DEFINE m, p_pk INT, p_name VARCHAR(10)
    DATABASE test1
```

```
    CREATE TABLE t1 ( pk INT, name VARCHAR(10) )
    INSERT INTO t1 VALUES (1, 'aaaa')
    INSERT INTO t1 VALUES (2, 'bbbbbb')
    INSERT INTO t1 VALUES (3, 'cccc')
    EXECUTE IMMEDIATE "create function proc3(v_max INT)"
                    ||" returning int, lvarchar;"
                    ||" define r_pk integer;"
                    ||" define r_name lvarchar;"
                    ||" foreach c1 for select pk,name into r_pk, r_name from
 t1 where pk <= v_max"
                    ||" return r_pk,r_name with resume;"
                    ||" end foreach;"
                    ||" end function"
    DECLARE c CURSOR FROM "EXECUTE FUNCTION proc3(?)"
    LET m = 100
    FOREACH c USING m INTO p_pk, p_name
        DISPLAY p_pk, p_name
    END FOREACH
    EXECUTE IMMEDIATE "drop function proc3"
    DROP TABLE t1
END MAIN
```

Stored procedure call with Oracle DB

Oracle supports stored procedures and stored functions as a group of PL/SQL statements that you can call by name. Oracle stored functions are very similar to stored procedures, except that a function returns a value to the environment in which it is called. Functions can be used in SQL expressions.

Stored procedures with output parameters

Oracle stored procedures or stored functions must be called with the input and output parameters specification in the `USING` clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position, and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must include the procedure in an anonymous PL/SQL block with `BEGIN` and `END` keywords:

```
PREPARE stmt FROM "begin proc1(?,?,?); end;"
```

Remark: Oracle stored procedures do not specify the size of number and character parameters. The size of output values (especially character strings) are defined by the calling context (i.e. the data type of the variable used when calling the procedure). When you pass a CHAR(10) to the procedure, the returning value will be filled with blanks to reach a size of 10 bytes.

Note that for technical reasons, the Oracle driver uses dynamic binding with `OCIBindDynamic()`. The Oracle Call Interface does not support stored procedures parameters with the `CHAR` data type when using dynamic binding. You must use `VARCHAR2` instead of `CHAR` to define character string parameters for stored procedures.

Here is a complete example creating and calling a stored procedure with output parameters:

```
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
                   "create procedure proc1("
                || "         p1 in int,"
                || "         p2 in out number,"
                || "         p3 in out varchar2"
                || "      )"
```

```
                || " is begin"
                || "  p2:= p1 + 0.23;"
                || "  p3:= 'Value = ' || to_char(p1);"
                || "end;"
   PREPARE stmt FROM "begin proc1(?,?,?); end;"
   LET n = 111
   EXECUTE stmt USING n IN, d INOUT, c INOUT
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored functions with a return value

To execute the stored function returning a value, you must include the function in an anonymous PL/SQL block with `BEGIN` and `END` keywords, and use an assignment expression to specify the place holder for the returning value:

```
PREPARE stmt FROM "begin ?:= func1(?,?,?); end;"
```

Stored procedures producing a result set

Oracle supports result set generation from stored procedures with the concept of cursor variables (`REF CURSOR`).

Note that Genero does not support cursor references produced by Oracle stored procedures or functions.

Stored procedure call with IBM® DB2®

IBM® DB2® implements stored procedures as a saved collection of SQL statements, which can accept and return user-supplied parameters. IBM® DB2® stored procedures can also produce one or more result sets. Beside stored procedures, IBM® DB2® supports user defined functions, typically used to define scalar functions returning a simple value which can be part of SQL expressions.

Stored procedures with output parameters

IBM® DB2® stored procedures must be called with the input and output parameters specification in the `USING` clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use the `CALL` SQL instruction:

```
PREPARE stmt FROM "call proc1(?,?,?)"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
                   "create procedure proc1("
                || "          in p1 int,"
                || "          out p2 decimal(6,2),"
                || "         inout p3 varchar(20)"
                || "        )"
                || " language sql begin"
                || "  set p2 = p1 + 0.23;"
                || "  set p3 = 'Value = ' || char(p1);"
                || "end"
   PREPARE stmt FROM "call proc1(?,?,?)"
   LET n = 111
```

```
   EXECUTE stmt USING n IN, d OUT, c INOUT
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored procedures producing a result set

With DB2® UDB, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows:

```
MAIN
   DEFINE i, n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
   INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
   INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
   INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
   EXECUTE IMMEDIATE "create procedure proc2( in key integer )"
                || " result sets 1"
                || " language sql"
                || "  begin"
                || "  declare c1 cursor with return for"
                || "    select * from tab1 where c1 > key;"
                || "  open c1;"
                || "  end"
   DECLARE curs CURSOR FROM "call proc2(?)"
   LET i = 1
   FOREACH curs USING i INTO n, d, c
       DISPLAY n, d, c
   END FOREACH
END MAIN
```

Stored procedures with output parameters and result set

It is possible to execute DB2® UDB stored procedures with output parameters and a result set.

The output parameter values are available after the `OPEN` cursor instruction:

```
   OPEN curs USING n IN, d OUT, c INOUT
   FETCH curs INTO rec.*
```

Stored procedure call with Microsoft™ SQL Server

SQL Server implements stored procedures, which are a saved collection of Transact-SQL statements that can take and return user-supplied parameters.

SQL Server stored procedures can also produce one or more result sets.

Stored procedures with output parameters

SQL Server stored procedures must be called with the input and output parameters specification in the `USING` clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use an ODBC `call` escape sequence:

```
PREPARE stmt FROM "{ call proc1(?,?,?) }"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```
MAIN
```

```
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
                  "create procedure proc1"
               || "             @v1 integer,"
               || "             @v2 decimal(6,2) output,"
               || "             @v3 varchar(20) output"
               || " as begin"
               || "  set @v2 = @v1 + 0.23"
               || "  set @v3 = 'Value = ' || cast(@v1 as varchar)"
               || "end"
   PREPARE stmt FROM "{ call proc1(?,?,?) }"
   LET n = 111
   EXECUTE stmt USING n IN, d OUT, c OUT
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored procedures producing a result set

With SQL Server, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows.

The next example uses a stored procedure with a simple `SELECT` statement. If the stored procedure contains additional Transact-SQL statements such as `SET` or `IF` (which is the case in complex stored procedures), SQL Server generates multiple result sets. By default the Genero SQL Server driver uses "server cursors" to support multiple active SQL statements. But SQL Server stored procedures generating multiple result sets cannot be used with server cursors: The server cursor is silently converted to a "default result set" cursor by the ODBC driver. Since Default result set cursors do not support multiple active statements, you cannot use another SQL statement while processing the results of such stored procedure. You must `CLOSE` the cursor created for the stored procedure before continuing with other SQL statements.

```
MAIN
   DEFINE i, n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
   INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
   INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
   INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
   EXECUTE IMMEDIATE "create procedure proc2 @key integer"
               || "  as select * from tab1 where c1 > @key"
   DECLARE curs CURSOR FROM "{ call proc2(?) }"
   LET i = 1
   FOREACH curs USING i INTO n, d, c
       DISPLAY n, d, c
   END FOREACH
END MAIN
```

It is possible to fetch large objects (text/image) from stored procedure generating a result set, however, if the stored procedure executes other statements as the `SELECT` (like `SET`/`IF` commands), the SQL Server ODBC driver will convert the server cursor to a regular default result set cursor, requiring the LOB columns to appear at the end of the select list. Thus, in most cases (stored procedures typically use `SET` / `IF` statements), you will have to move the LOB columns and the end of the column list.

Stored procedures returning a cursor as output parameter

SQL Server supports "cursor output parameters": A stored procedure can declare/open a cursor and return a reference of the cursor to the caller.

SQL Server stored procedures returning a cursor as output parameter are not supported. There are two reasons for this: The language does not have a data type to store a server cursor reference, and the underlying ODBC driver does not support this anyway.

Stored procedures with return code

SQL Server stored procedures can return integer values. To get the return value of a stored procedure, you must use an assignment expression in the ODBC `call` escape sequence:

```
PREPARE stmt FROM "{ ? = call proc3(?,?,?) }"
```

Stored procedures with output parameters, return code and result set

With SQL Server you can call stored procedures with a return code, output parameters and producing a result set.

Return codes and output parameters are the last items returned to the application by SQL Server; they are not returned until the last row of the result set has been fetched, after the `SQLMoreResults()` ODBC function is called. If output parameters are used, the SQL Server driver executes a `SQLMoreResults()` call when closing the cursor instead of `SQLCloseCursor()`, to get the return code and output parameter values from SQL Server.

```
MAIN
   DEFINE r, i, n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
   INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
   INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
   INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
   EXECUTE IMMEDIATE "create procedure proc3 @key integer output"
                || "  as begin"
                || "     set @key = @key - 1"
                || "     select * from tab1 where c1 > @key"
                || "     return (@key * 3)"
                || "  end"
   DECLARE curs CURSOR FROM "{ ? = call proc3(?) }"
   LET i = 1
   OPEN curs USING r INOUT, i INOUT
   DISPLAY r, i
   FETCH curs INTO n, d, c
   FETCH curs INTO n, d, c
   FETCH curs INTO n, d, c
   DISPLAY r, i
   CLOSE curs
   DISPLAY r, i -- Now the returned values are available
END MAIN
```

The return code and output parameter variables must be defined as `INOUT` in the `OPEN` instruction.

Stored procedure call with PostgreSQL

PostgreSQL implements stored functions that can return values. If the function returns more that one value, you must specify the returning values as function parameters with the `OUT` keyword. If the function returns a unique value, you can use the `RETURNS` clause.

Pay attention to the function signature; PostgreSQL allows function overloading. For example, `func(int)` and `func(char)` are two different functions. To drop a function, you must specify the parameter type to identify the function signature properly.

Stored functions with output parameters

To execute a stored function with PostgreSQL, you must use `SELECT * FROM function`, as shown in this line:

```
PREPARE stmt FROM "select * from proc1(?)"
```

In order to retrieve returning values into program variables, you must use an `INTO` clause in the `EXECUTE` instruction.

The following example shows how to call a stored function with PostgreSQL:

```
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
                  "create function proc1("
               || "          p1 integer,"
               || "          out p2 numeric(6,2),"
               || "          out p3 varchar(200)"
               || "          )"
               || " as $$"
               || "  begin"
               || "    p2:= p1 + 0.23;"
               || "    p3:= 'Value = ' || cast(p1 as text);"
               || "  end;"
               || " $$ language plpgsql"
   PREPARE stmt FROM "select * from proc1(?)"
   LET n = 111
   EXECUTE stmt USING n INTO d, c
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored functions producing a result set

With PostgreSQL, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows:

```
MAIN
   DEFINE i, n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
   INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
   INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
   INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
   EXECUTE IMMEDIATE "create function proc2(integer)"
               || " returns setof tab1"
               || " as $$"
               || "  select * from tab1 where c1 > $1;"
               || " $$ language sql"
   DECLARE curs CURSOR FROM "select * from proc2(?)"
   LET i = 1
   FOREACH curs USING i INTO n, d, c
       DISPLAY n, d, c
   END FOREACH
END MAIN
```

Stored functions with output parameters and result set

With PostgreSQL you cannot return output parameters and a result set from the same stored procedure; both use the same technique to return values to the client, in the context of result columns to be fetched.

Stored procedure call with Oracle MySQL

MySQL implements stored procedures and stored functions as a collection of SQL statements that can take and return user-supplied parameters. Functions are very similar to procedures, except that they return a scalar value and can be used in SQL expressions.

Stored procedures with output parameters

Since MySQL C API (version 5.0) does not support an output parameter specification, the `IN`/`OUT`/`INOUT` technique cannot be used.

In order to return values from a MySQL stored procedure or stored function, you must use SQL variables. There are three steps to execute the procedure or function:

1. With the `SET` SQL statement, create and assign an SQL variables for each parameter.
2. `CALL` the stored procedure or stored function with the created SQL variables.
3. Perform a `SELECT` statement to return the SQL variables to the application.

In order to retrieve returning values into program variables, you must use an `INTO` clause in the `EXECUTE` instruction.

The following example shows how to call a stored procedure with output parameters:

MySQL version 5.0 does not allow you to prepare the `CREATE PROCEDURE` statement; you may need to execute this statement from the mysql command line tool.

MySQL version 5.0 cannot execute "`SELECT @variable`" with server-side cursors. Since the MySQL driver uses server-side cursors to support multiple active result sets, it is not possible to execute the `SELECT` statement to return output parameter values.

MySQL version >=5.0 evaluates "`@variable`" user variables assigned with a string as large `text` (CLOB) expressions. That type of values must normally be fetched into `TEXT` variable. To workaround this behavior, you can use the `substring(@var,1,255)` function to return a `VARCHAR()` expression from MySQL and fetch into a `VARCHAR()` variable.

```
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
                  "create procedure proc1("
               || "   p1 integer,"
               || "   out p2 numeric(6,2),"
               || "   out p3 varchar(200)
               || " )"
               || " no sql begin"
               || "    set p2 = p1 + 0.23;"
               || "    set p3 = concat( 'Value = ', p1 );"
               || "  end;"
   LET n = 111
   EXECUTE IMMEDIATE "set @p1 = ", n
   EXECUTE IMMEDIATE "set @p2 = NULL"
   EXECUTE IMMEDIATE "set @p3 = NULL"
   EXECUTE IMMEDIATE "call proc1(@p1, @p2, @p3)"
   PREPARE stmt FROM "select @p2, substring(@p3,1,200)"
   EXECUTE stmt INTO d, c
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored functions returning values

The following example shows how to retrieve the return value of a stored function with MySQL:

MySQL version 5.0 does not allow you to prepare the `CREATE FUNCTION` statement; you may need to execute this statement from the mysql command line tool.

```
MAIN
   DEFINE n INTEGER
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE "create function func1(p1 integer)"
                || " no sql begin"
                || "    return concat( 'Value = ', p1 );"
                || "  end;"
   PREPARE stmt FROM "select func1(?)"
   LET n = 111
   EXECUTE stmt USING n INTO c
   DISPLAY c
END MAIN
```

Stored procedures producing a result set

Note that MySQL version 5.0 stored procedures and stored functions cannot return a result set.

Stored procedure call with SAP Sybase ASE

Sybase ASE supports stored procedures, which can take and return user-supplied parameters.

Sybase ASE stored procedures can also produce one or more result sets.

Stored procedures with output parameters

Sybase ASE stored procedures must be called with the input and output parameters specification in the `USING` clause of the `EXECUTE`, `OPEN` or `FOREACH` instruction. As in normal dynamic SQL, parameters must correspond by position and the `IN/OUT/INOUT` options must match the parameter definition of the stored procedure.

To execute the stored procedure, you must use a specific syntax to have the database driver identify the statement as an RPC call. The syntax of an RPC call must be:

```
!rpc procedure-name ( [ @param-name [,...] ] )
```

The parameter names must be specified, with the same names as the arguments of the stored procedure, because the ODI driver must bind stored procedure parameters by name.

Example:

```
PREPARE stmt FROM "!rpc update_account ( @custid, @old, @new )"
```

Here is a complete example creating and calling a stored procedure with output parameters:

```
MAIN
   DEFINE n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   EXECUTE IMMEDIATE
                   "create procedure proc1"
                || "   @v1 integer,"
                || "   @v2 decimal(6,2) output,"
                || "   @v3 varchar(20) output"
                || " as begin"
                || "  set @v2 = @v1 + 0.23"
```

```
                || "  set @v3 = 'Value = ' || cast(@v1 as varchar)"
                || "end"
   PREPARE stmt FROM "!rpc proc1( @v1, @v2, @v3 )"
   LET n = 111
   EXECUTE stmt USING n IN, d OUT, c OUT
   DISPLAY d
   DISPLAY c
END MAIN
```

Stored procedures producing a result set

With Sybase, you can execute stored procedures returning a result set. To do so, you must declare a cursor and fetch the rows.

When the stored procedure generates multiple active statements, you cannot use another SQL statement while processing the results of such stored procedure. You must `CLOSE` the cursor created for the stored procedure before continuing with other SQL statements.

```
MAIN
   DEFINE i, n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
   INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
   INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
   INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
   EXECUTE IMMEDIATE "create procedure proc2 @key integer"
                || "  as select * from tab1 where c1 > @key"
   DECLARE curs CURSOR FROM "!rpc proc2( @key )"
   LET i = 1
   FOREACH curs USING i INTO n, d, c
       DISPLAY n, d, c
   END FOREACH
END MAIN
```

Stored procedures with output parameters, return code and result set

With Sybase ASE stored procedures, you call stored procedures with a return code, output parameters and producing a result set.

Return codes and output parameters are the last items returned to the application by Sybase; they are not returned until the last row of the result set has been fetched.

```
MAIN
   DEFINE r, i, n INTEGER
   DEFINE d DECIMAL(6,2)
   DEFINE c VARCHAR(200)
   DATABASE test1
   CREATE TABLE tab1 ( c1 INTEGER, c2 DECIMAL(6,2), c3 VARCHAR(200) )
   INSERT INTO tab1 VALUES ( 1, 123.45, 'aaaaaa' )
   INSERT INTO tab1 VALUES ( 2, 123.66, 'bbbbbbbbb' )
   INSERT INTO tab1 VALUES ( 3, 444.77, 'cccccc' )
   EXECUTE IMMEDIATE "create procedure proc3 @key integer output"
                || "  as begin"
                || "     set @key = @key - 1"
                || "     select * from tab1 where c1 > @key"
                || "     return (@key * 3)"
                || "  end"
   DECLARE curs CURSOR FROM "!rpc proc3( @key ) }"
   LET i = 1
   OPEN curs USING r OUT, i OUT
   DISPLAY r, i
```

```
   FETCH curs INTO n, d, c
   FETCH curs INTO n, d, c
   FETCH curs INTO n, d, c
   DISPLAY r, i
   CLOSE curs
   DISPLAY r, i -- Now the returned values are available
END MAIN
```

SQL performance

- Performance with dynamic SQL on page 452
- Performance with transactions on page 453
- Avoiding long transactions on page 454
- Declaring prepared statements on page 454
- Saving SQL resources on page 454
- Optimizing scrollable cursors on page 455

Performance with dynamic SQL

Comparing static SQL statements and dynamc SQL statements used in a loop.

Although SQL statements can be directly specified in the program source as a part of the language as static SQL, it is sometimes more efficient to use dynamic SQL instead, when you are executing SQL statements repeatedly.

Dynamic SQL allows you to PREPARE the SQL statements once and EXECUTE N times, improving performance.

Note however that implementing prepared statements with dynamic SQL has a cost in terms of database resources and code readability: When a simple static SQL statement is executed, database client and server resources are allocated for the statement and are reused for the next Static SQL statement. With dynamic SQL, you define a statement handle and allocate database resources that last until you FREE the handle. Regarding code readability, static SQL statements can be written directly in the source code (as another language statement), while Dynamic SQL uses several instructions and takes the SQL text as a string expression. Additionally, static SQL statements are parsed at compile time so you can detect syntax errors in the SQL text before executing the programs.

Therefore, dynamic SQL should only be used if the SQL statement is created at runtime (with a where part generated by a CONSTRUCT for example) or if the execution time is too long with static SQL (assuming that it's only a statement preparation issue).

To perform static SQL statement execution, the database interface must use the basic API functions provided by the database client. These are usually equivalent to the PREPARE and EXECUTE instructions. So when you write a static SQL statement in your program, it is actually converted to a PREPARE + EXECUTE behing the scene.

For example, the following code:

```
FOR n=1 TO 100
   INSERT INTO tab VALUES ( n, c )
END FOR
```

is actually equivalent to:

```
FOR n=1 TO 100
   PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
   EXECUTE s USING n, c
END FOR
```

To improve the performance of the preceding code, use a PREPARE instruction before the loop and put an EXECUTE instruction inside the loop:

```
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
FOR n=1 TO 100
   EXECUTE s USING n, c
END FOR
```

Performance with transactions

Commit database changes by blocks of transaction speeds performance with some database servers.

To mimic the IBM® Informix® auto-commit behavior with an ANSI compliant RDBMS like Oracle or DB2® UDB, the database driver must perform an implicit commit after each statement execution, if the SQL statement is not inside a transaction block. This generates unnecessary database operations and can slow down big loops. To avoid this implicit commit, you can control the transaction with BEGIN WORK / COMMIT WORK around the code containing a lot of SQL statement execution.

This technique is especially recommended with SQLite, because the SQLite database library performs a lot of operations during a commit.

For example, the following loop will generate 2000 basic SQL operations (1000 inserts plus 1000 commits):

```
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
FOR n=1 TO 100
   EXECUTE s USING n, c   -- Generates implicit COMMIT
END FOR
```

You can improve performance if you put a transaction block around the loop:

```
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
BEGIN WORK
FOR n=1 TO 100
   EXECUTE s USING n, c   -- In transaction -> no implicit COMMIT
END FOR
COMMIT WORK
```

With this code, only 1001 basic SQL operations will be executed (1000 inserts plus 1 commit).

However, you must take care when generating large transactions because all modifications are registered in transaction logs. This can result in a lack of database server resources ("transaction too long" errors, for example) when the number of operations is very big. If the SQL operation does not require a unique transaction for database consistency reasons, you can split the operation into several transactions, as in this example:

```
PREPARE s FROM "INSERT INTO tab VALUES ( ?, ? )"
BEGIN WORK
FOR n=1 TO 100
   IF n MOD 10 == 0 THEN
      COMMIT WORK
      BEGIN WORK
   END IF
   EXECUTE s USING n, c   -- In transaction -> no implicit COMMIT
END FOR
COMMIT WORK
```

Note that the LOAD instruction automatically starts a transaction, if not yet initiated. Therefore there is no need to enclose the LOAD statement within a BEGIN WORK / COMMIT WORK, except if other SQL statements are part of the transaction and need to be processed as a single atomic database change.

Avoiding long transactions

Long transactions consume resources and decrease concurrent data access.

Old applications based on IBM® Informix® database without transaction logging might perform long running SQL modifications.

With recent database engines, using huge transactions can lead to errors because of transaction log buffer overflow. For example, if a table holds many rows, a "`DELETE FROM table`" might produce a "snapshot too old" error in Oracle, if the rollback segments are too small.

Therefore, you must avoid long transactions when connected to a database using transactions:

- keep transactions as short as possible.
- access the least amount of data possible while in a transaction.
- split a long transaction into many short transactions. Use a loop to handle each block.
- to delete all rows from a table use the "`TRUNCATE TABLE`" instruction instead of "`DELETE FROM`" (Not for all vendors).
- In the end, increase the size of the transaction log to avoid filling it up.

Declaring prepared statements

Optimize prepared cursor statements by using the `FROM` clause of `DECLARE CURSOR`.

Line 2 of this example shows a cursor declared with a prepared statement:

```
PREPARE s FROM "SELECT * FROM table WHERE ", condition
DECLARE c CURSOR FOR s
```

While this has no performance impact with IBM® Informix® database drivers, it can become a bottleneck when using non-IBM Informix® databases:

Statement preparation consumes a lot of memory and processor resources. Declaring a cursor with a prepared statement is a native IBM® Informix® feature, which consumes only one real statement preparation. Non-IBM Informix® databases do not support this feature, so the statement is prepared twice (once for the `PREPARE`, and once for the `DECLARE`). When used in a big loop, this code can cause performance problems.

To optimize the code, use the `FROM` clause in the `DECLARE` statement:

```
DECLARE c CURSOR FROM "SELECT * FROM table WHERE " || condition
```

By using this solution only one statement preparation will be done by the database server.

> **Note:** This performance problem does not occur with `DECLARE` statements using static SQL.

Saving SQL resources

SQL cursors and prepared statement consume resources that should be freed when useless.

To write efficient SQL in your programs, you can use dynamic SQL. However, when using dynamic SQL, you allocate an SQL statement handle on the client and server side, consuming resources. According to the database type, this can be a few bytes or a significant amount of memory. When executing several static SQL statements, the same statement handle is reused and thus less memory is needed.

The language allows you to use either static SQL or dynamic SQL, so it's in your hands to choose memory or performance. However, in some cases the same code will be used by different kinds of programs, needing either low resource usage or good performance. In many OLTP applications you can actually distinguish two type of programs:

- Programs where memory usage is not a problem but good performance is needed (typically, batch programs executed as a unique instance during the night).
- Programs where performance is less important but memory usage must be limited (typically, interactive programs executed as multiple instances for each application user).

To reuse the same code for interactive programs and batch programs, you can do this:

1. Define a local module variable as an indicator for the prepared statement.
2. Write a function returning the type of program (for example, 'interactive' or 'batch' mode).
3. Then, in a reusable function using SQL statements, prepare and free the statement according to the indicators, as shown in the next example.

```
PRIVATE DEFINE up_prepared BOOLEAN

FUNCTION getUserPermissions( username )
  DEFINE username VARCHAR(20)
  DEFINE cre, upd, del CHAR(1)

  IF NOT up_prepared THEN
     PREPARE up_stmt FROM "SELECT can_create, can_update, cab_delete"
                          || " FROM user_perms WHERE name = ?"
     LET up_prepared = TRUE
  END IF

  EXECUTE up_stmt USING username INTO cre, upd, del

  IF isInteractive() THEN
     FREE up_stmt
     LET up_prepared = FALSE
  END IF

  RETURN cre, upd, del

END FUNCTION
```

The first time this function is called, the `up_prepared` value will be `FALSE`, so the statement will be prepared. The next time the function is called, the statement will be re-prepared only if `up_prepared` is `TRUE`. The statement is executed and values are fetched into the variables returned. If the program is interactive, the statement is freed and set the `up_prepared` module variable back to `FALSE`, forcing statement preparation in the next call of this function.

Optimizing scrollable cursors

A programming pattern to get fresh data from scrollable cursors.

Generally, when using scrollable cursors, the database server or the database client software (i.e. the application) will make a static copy of the result set produced by the `SELECT` statement. For example, when using an IBM® Informix® database engine, each scrollable cursor will create a temporary table to hold the result set. Thus, if the `SELECT` statement returns all columns of the table(s) in the `FROM` clause, the database software will make a copy of all these values. This practice has two disadvantages: A lot of resources are consumed, and the data is static.

A good programming pattern to save resources and always get fresh data from the database server is to declare two cursors based on the primary key usage, if the underlying database table has a primary key (or unique index constraint): The first cursor must be a scrollable cursor that executes the `SELECT` statement, but returns only the primary keys. The `SELECT` statement of this first cursor is typically assembled at runtime with the where-part produced by a `CONSTRUCT` interactive instruction, to give a subset of the rows stored in the database. The second cursor (actually, a `PREPARE`/`EXECUTE` statement handle) performs a single-row `SELECT` statement listing all columns to be fetched for a given record, based on the primary key value of the current row in the scrollable cursor list. The second statement must use a ? question mark place holder to execute the single-row `SELECT` with the current primary key as SQL parameter.

If the primary key `SELECT` statement needs to be ordered, check that the database engine allows that columns used in the `ORDER BY` clause do not need to appear in the `SELECT` list. For example, this was the case with IBM® Informix® servers prior to version 9.4. If needed, the `SELECT` list can be completed with

the columns used in `ORDER BY`, you can then just list the variable that holds the primary key in the `INTO` clause of `FETCH`.

Note also that the primary key result set is static. That is, if new rows are inserted in the database or if rows referenced by the scroll cursor are deleted after the scroll cursor was opened, the result set will be outdated. In this case, you can refresh the primary key result set by re-executing the scroll cursor with `CLOSE`/`OPEN` commands.

This code example illustrates this programming pattern:

```
MAIN
   DEFINE wp VARCHAR(500)
   DATABASE test1
   -- OPEN FORM / DISPLAY FORM with c_id and c_name fields
   ...
   -- CONSTRUCT generates wp string...
   ...
   LET wp = "c_name LIKE 'J%'"
   DECLARE clist SCROLL CURSOR FROM "SELECT c_id FROM customer WHERE " || wp
   PREPARE crec FROM "SELECT * FROM customer WHERE c_id = ?"
   OPEN clist
   MENU "Test"
        COMMAND "First"    CALL disp_cust("F")
        COMMAND "Next"     CALL disp_cust("N")
        COMMAND "Previous" CALL disp_cust("P")
        COMMAND "Last"     CALL disp_cust("L")
        COMMAND "Refresh"  CLOSE clist OPEN clist
        COMMAND "Quit" EXIT MENU
   END MENU
   FREE crec
   FREE clist

END MAIN

FUNCTION disp_cust(m)
   DEFINE m CHAR(1)
   DEFINE rec RECORD
          c_id INTEGER,
          c_name VARCHAR(50)
      END RECORD
   CASE m
       WHEN "F" FETCH FIRST clist INTO rec.c_id
       WHEN "N" FETCH NEXT clist INTO rec.c_id
       WHEN "P" FETCH PREVIOUS clist INTO rec.c_id
       WHEN "L" FETCH LAST clist INTO rec.c_id
   END CASE
   INITIALIZE rec.* TO NULL
   IF SQLCA.SQLCODE == NOTFOUND THEN
      ERROR "You reached to top or bottom of the result set."
   ELSE
      EXECUTE crec USING rec.c_id INTO rec.*
      IF SQLCA.SQLCODE == NOTFOUND THEN
         ERROR "Row was not found in the database, refresh the result set."
      END IF
   END IF
   DISPLAY BY NAME rec.*
END FUNCTION
```

Database connections

Explains how to manage database connections in a program.

- Understanding database connections on page 457
- Opening a database connection on page 458
- Database client environment on page 459
- Connection parameters on page 461
- Connection parameters in database specification on page 464
- Direct database specification method on page 465
- Indirect database specification method on page 466
- IBM Informix emulation parameters in FGLPROFILE on page 466
- Database vendor specific parameters in FGLPROFILE on page 469
- Database user authentication on page 473
- Unique session mode connection instructions on page 476
- Multi-session mode connection instructions on page 477
- Miscellaneous SQL statements on page 480

Understanding database connections

A *database connection* is a session of work, opened by the program to communicate with a specific database server, in order to execute SQL statements as a specific user.

Before working with database connections, make sure you have properly installed and configured all software, using the correct database client software/environment, and BDL database driver. It is very important to understand database client settings, regarding user authentication as well as database client character set configuration.

Note that on some platforms like on mobile devices, Genero BDL includes the SQLite lightweight database library, which is the default. Therefore, when executing programs on these platforms, there is no need to install a database client sowftare and configure the database driver for the runtime system.

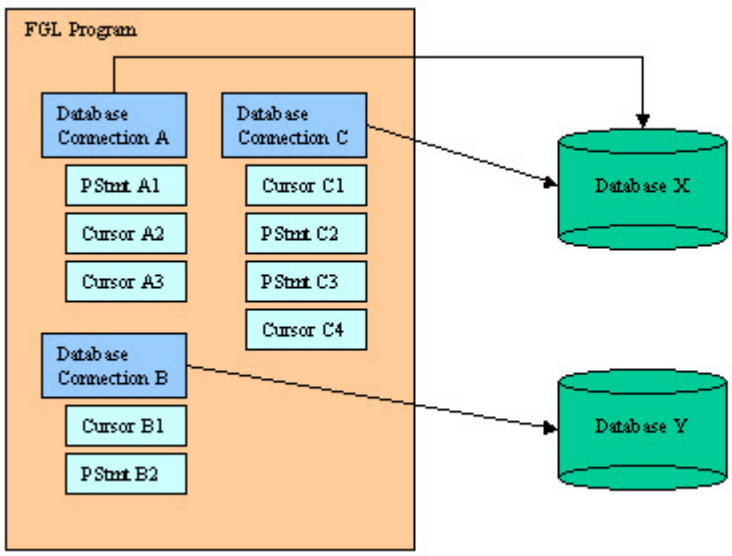


Figure 21: Schema example of a program using three database connections

The database user can be identified explicitly for each connection. Usually, the user is identified by a login and a password, or by using the authentication mechanism of the operating system (or even from a tier security system).

Database connection instructions DATABASE / CONNECT TO can not be prepared and executed as dynamic SQL statements.

There are two kind of connection modes: *unique-session* and *multi-session* mode. When using the DATABASE and CLOSE DATABASE instructions, the program is in unique-session mode. When using the CONNECT TO, SET CONNECTION and DISCONNECT instructions, the program is in *multi-session* mode. These connection modes are not compatible.

In unique-session mode, the DATABASE instruction initiates a connection the the database server and creates the current session. The database connection is terminated with the CLOSE DATABASE instruction, or when another DATABASE instruction is executed, or when the program ends.

In multi-session mode, open a database session with the CONNECT TO instruction. Other connections can be created with subsequent CONNECT TO instructions. To switch to a specific session, use the SET CONNECTION instruction; this suspends other opened connections. Disconnect from a specific or from all sessions with the DISCONNECT instruction. The end of the program disconnects all sessions automatically.

Once connected to a database server, the program uses the current session to execute SQL statements in that context.

Opening a database connection

A database connection identifies the SQL database server and the database entity the program connects to, in order to execute SQL statements.

To connect to a database server, the database driver needs to be loaded, and the SQL data source most be provided. Additionally, user authentication with user name / password may also be needed. All these parameters define connection information.

There are different ways to give connection information, and it is possible to mix the different methods to specify connection parameters. However, if provided, the database user name and password have to be specified together with the same method.

A database connection is performed in programs with the DATABASE or CONNECT TO instruction:

```
CONNECT TO dbspec [USER username USING password]
```

or

```
DATABASE dbspec
```

Prefer the CONNECT TO instruction, as it allows to specify a user name and password.

For portability reasons, it is not recommended that you use database vendor specific syntax (such as 'dbname@dbserver') in the DATABASE or CONNECT TO instructions: Connections must be identified in programs by a single name, while connection parameters are provided in external files.

Indirect database specification uses entries in the FGLPROFILE configuration file: When a DATABASE or CONNECT TO instruction is executed with the parameter *dbspec*, the runtime system first looks into FGLPROFILE for entries starting with dbi.database.*dbspec*, and uses these connection parameters if found. Otherwise, the runtime system will do direct database specification, by using the *dbspec* string to connect to the server.

> **Important:** When using FGLPROFILE entries for database connection parameters, keep in mind that entries must be written in lowercase.

Use a string variable with the DATABASE or CONNECT TO statement, in order to specify the database source at runtime. This solution gives you the best flexibility.

The string variable can be set from your own configuration file, from a program argument or from an environment variable.

Example

```
MAIN
  DEFINE db, us, pwd CHAR(50)
  LET db = fgl_getenv("MYDBSOURCE")
  LET us = arg_val(2)
  LET pwd = arg_val(3)
  CONNECT TO db USER us USING pwd
  ...
END MAIN
```

Database client environment

To connect to a database server, the programs must be executed in the correct database client environment. The database client software is usually included in the database server software, so you do not need to install it when your programs are executed on the same machine as the database server. However, you must install the database client software in three-tier configurations, when applications and database servers run on different systems.

This section describes basic configuration elements of the database client environment for some well-known database servers.

IBM® DB2 Universal Database™

1. The DB2DIR environment variable must define the DB2® software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to `$DB2DIR/lib`.
4. The DB2® client library `'DB2DIR/lib/libdb2*'` must be available.
5. The remote server node and the remote database must be declared locally with the CATALOG db2 command.
6. Make sure the database client locale is properly defined.
7. You can make a connection test with the IBM® db2 tool.

IBM® Informix® Dynamic Server

1. The INFORMIXDIR environment variable must define the IBM® Informix® software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to `$INFORMIXDIR/lib:$INFORMIXDIR/lib/esql`.
4. The IBM® Informix® client libraries `'INFORMIXDIR/lib/*'` must be available.
5. The INFORMIXSERVER environment variable can be used to define the name of the database server.
6. The sqlhost file must define the database server identified by INFORMIXSERVER.
7. Make sure the database client locale is properly defined.
8. You can make a connection test with the IBM® Informix® dbaccess tool.

Oracle MySQL

1. The MYSQL_HOME environment variable must define the MySQL software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to `$MYSQL_HOME/lib`.
4. Make sure the database client locale is properly defined.
5. You can make a connection test with the MySQL tool.

Oracle database

1. The ORACLE_HOME environment variable must define the Oracle software installation path.

2. The ORACLE_SID environment variable can be used to define the name of the local database instance.
3. The PATH environment variable must define the access path to database client programs.
4. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to `$ORACLE_HOME/lib`.
5. The Oracle client library `'ORACLE_HOME/lib/libclntsh*'` must be available.
6. The TNSNAMES.ORA file must define the database server identifiers for remote connections (the Oracle Listener must be started on the database server to allow remote connections).
7. The SQLNET.ORA file must define network settings for remote connections.
8. Make sure the database client locale is properly defined.
9. You can make a connection test with the Oracle sqlplus tool.

PostgreSQL

1. The PGDIR environment variable must define the PostgreSQL software installation path.
2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to `$PGDIR/lib`.
4. The PostgreSQL client library `'PGDIR/lib/libpq*'` must be available.
5. On the database server, the `pg_hba.conf` file must define security policies.
6. Make sure the database client locale is properly defined.
7. You can make a connection test with the PostgreSQL psql tool.

Microsoft™ SQL Server

1. Make sure that ODBC data source is defined on database client and database server systems, with the correct ODBC driver. Note that Genero FGL provides different sort of SQL Server drivers:
   - The SNC driver is based on the SQL Native Client ODBC driver (SQLNCLI*.DLL). The version of the SQL Native Client must match the ODI driver.
   - The FTM driver is based on the FreeTDS ODBC driver (libtdsodbc.so). This driver can be used if you want to connect to SQL Server from a UNIX™ machine.
   - The ESM driver is based on the EasySoft ODBC driver (libessqlsrv.so). This driver can be used if you want to connect to SQL Server from a UNIX™ machine.
2. On Windows™ platforms, the PATH environment variable must define the access path to database client programs (ODBC32.DLL). On UNIX/Linux platforms, check database client software documentation for environment settings (LD_LIBRARY_PATH, ldconfig).
3. On Windows™, Check the SQL Server Client configuration with the Client Network Utility tool: Verify that the ANSI to OEM conversion corresponds to the execution of applications in a CONSOLE environment.
4. Make sure the database client locale is properly defined. On UNIX/Linux platforms, check that the client character set parameter of the ODBC data source corresponds the the locale used by the application (LANG/LC_ALL).
5. On Windows™, you can make a connection test with the Microsoft™ Query Analyzer tool. On UNIX/Linux, see client software documentation for available SQL command tools (isql for example).

SQLite

1. The SQLite database driver includes the SQLite library, except on systems where that library is commonly available, like Linux distributions, Mac OS X and mobile devices.
2. Make sure the database client locale is properly defined. The SQLite library uses UTF-8. If the current character set (LANG/LC_ALL) is not UTF-8, like plain ASCII or UTF-8, the database driver will make appropriate character set conversions.
3. You can make a connection test with the `sqlite3` command line tool.

Sybase Adaptive Server Enterprise (ASE)

1. The SYBASE environment variable must define the Sybase ASE software installation path.

2. The PATH environment variable must define the access path to database client programs.
3. On UNIX™, LD_LIBRARY_PATH (or equivalent) must hold the path to the client libraries `libsybct.so` and `libsybcs.so`. On Windows™, the path to the DLLs must be defined in PATH.
4. Check the Sybase Client configuration, especially server name definition in connection's directory source, see DSQUERY environment variable.
5. Make sure the database client locale is properly defined.
6. You can make a connection test with the Sybase ISQL tool.

Connection parameters

This section describes the different parameters which need to be specified in order to connect to a database. The parameters can be provided with different methods (in the connection string or in FGLPROFILE settings). Some of these parameters are optional. For example, if the database user is authenticated by the operating system, username/password parameters are not needed.

Database source specification (source)

In database connection parameters, the `source` parameter identifies the data source name.

If the `source` parameter is defined with an empty value (""), the database interface connects to the default database server, which is usually the local server.

If the `source` entry is not present in FGLPROFILE, direct database specification method takes place.

Table 154: Meaning of the `source` connection parameter for supported databases

Database Type	Value of "source" entry	Description
Generic ODBC	*datasource*	ODBC Data Source
IBM® Informix®	*dbname*[@*dbserver*]	IBM® Informix® database specification
IBM® DB2®	*dsname*	DB2® Catalogued Database
Oracle MySQL / MariaDB	*dbname*[@*host*[:*port*]] or *dbname*[@*localhost~socket*]	Database Name @ Host Name: TCP Port or Database Name @ Local host ~ UNIX™ socket file
Oracle Database	*tnsname*	Oracle TNS Service name
PostgreSQL	*dbname*[@*host*[:*port*]][?*options*]	Database Name @ Host Name : TCP Port ? PostgreSQL URI-style query string options
SQL Server	*datasource*	ODBC Data Source
SQLite	*filename* or `:memory:`	Database file path, or simple file name to be found with DBPATH, or `:memory:` to create a database in memory.
Sybase Adaptive Server Enterprise (ASE)	*dbname*[@*engine*]	Database Name @ Engine Name

Database driver specification (driver)

In database connection parameters, the `driver` parameter identifies the type of database driver to be used.

The driver must correspond to the database client software.

> **Important:** Pay attention to the binary architecture of the database client software: Genero runtime system and database client binaries must match. For example, a 32 bit Oracle client can not be used with a Genero 64 bit runtime system.

We distinguish two sort of database driver names:

- Generic driver names ("`dbmora`", "`dbmsnc`"), and aliases ("`oracle`", "`sqlserver`")
- Version-stamped driver names ("`dbmora_12`", "`dbmsnc_10`", "`dbmsnc_11`")

A driver name "`dbm`*`xxx`*" identifies a generic driver name for the database server identified by the code `xxx`.

For example, in FGLPROFILE, to define the database driver for the Oracle OCI client (code "`ora`"), use the name "`dbmora`":

```
dbi.database.stores.driver = "dbmora"
```

For convenience, it is also possible to specify a long name (alias) such as "`oracle`" or "`sqlserver`", as defined in the database driver table below.

Use generic named drivers (with the latest database client software available on the platform), instead of version-stamped driver names. Use the version-stamped driver name only if the most recent database client software is not available on the platform.

Check for library dependency on your system, to identify the database client library required by the driver with the generic name. The driver definition table below lists the driver names for each supported database client types and versions. For example, on Linux platform, use the ldd command:

```
$ ldd $FGLDIR/dbdrivers/dbmmys.so
      ...
      libmysqlclient.so.18 => ...
      ...
```

Drivers with generic name are compatible with the latest database client version available on the platform. Thus, according to the platform, the same generic driver name can refer for different database client software. For example, on a platform where only MySQL 5.1 is available, `dbmmys` will match the MySQL 5.1 client, while on a more recent platform, `dbmmys` will match the MySQL 5.5 client.

To limit the number of drivers, if the database client software allows it, the drivers are build with the oldest database client version that is compatible with the latest available database client versions. For example, the `dbmmys_5_5` driver is build with the MySQL Client 5.5, but is compatible with MySQL 5.6 and 5.7 clients.

Note that given driver (combined with the corresponding database client software) can connect to a database server of an older version, if the database vendor client/server protocol supports the combination. For example, you can use an Oracle client version 12c to connect to an Oracle 11g server.

A default driver can be specified with the `dbi.default.driver` FGLPROFILE entry. This driver will be used for all database connections that do not specify the driver explicitely:

```
dbi.default.driver = "dbmora"
```

If this entry is not defined, and if no driver parameter is specified for the data source, the driver name defaults to `dbmdefault`. This default driver is a copy of the database driver that was chosen during installation.

Table 155: Database driver names according to database client type

Name with db client version	Generic name / alias	Code	Database client software version	UNIX™ shared objects	Microsoft™ Windows™ DLLs	Mac OS X™ dynamic libraries
dbmase_16	dbmase / sybase_ase	ase	Sybase ASE Open Client Library 16.x	libsybct[64].so, libsybcs[64].so	libsybct[64].dll, libsybcs[64].dll	N/A
dbmdb2_10	dbmdb2 / db2	db2	IBM® DB2® UDB Client 10.x	libdb2.so.1	db2cli.dll	N/A
dbmesm_1	dbmesm / easysoft_sqlserver	esm	EasySoft ODBC for SQL Server	libessqlsrv.so	N/A	N/A
dbmifx_9	dbmifx / informix	ifx	IBM® Informix® CSDK 2.80 and higher	libifsql.so, libifasf.so, libifgen.so, libifos.so, libifgls.so, libifglx.so	isqlt09a.dll	libifsql.dylib, libifasf.dylib, libifgen.dylib, libifos.dylib, libifgls.dylib, libifglx.dylib
dbmmys_5_1		mys	MySQL Client 5.1.x	libmysqlclient.so.16	libmysql.dll	libmysqlclient.16.dylib
dbmmys_5_5	dbmmys / mysql	mys	MySQL Client 5.5.x and higher / MariaDB 10.x and higher	libmysqlclient.so.18	libmysql.dll	libmysqlclient.18.dylib
dbmntz_6	dbmntz / netezza	ntz	IBM® Netezza® (6.x)	libnzodbc.so	odbc32.dll	N/A
dbmodc_3	dbmodc / odbc	odc	Generic ODBC (ODBC 3.x)	libodbc.so	odbc32.dll	libodbc.dylib
dbmora_11		ora	OCI Client V11	libclntsh.so.11.1	oci.dll	N/A
dbmora_12	dbmora / oracle	ora	OCI Client V12	libclntsh.so.12.1	oci.dll	N/A
dbmpgs_9	dbmpgs / postgresql	pgs	PostgreSQL Client 9.x	libpq.so.5	libpq.dll	libpq.5.dylib
dbmsnc_9		snc	SQL Native client 2005 (V9)	N/A	odbc32.dll / SQLNCLI.DLL	N/A
dbmsnc_10		snc	SQL Native client 2008 (V10)	N/A	odbc32.dll / SQLNCLI10.DLL	N/A

Name with db client version	Generic name / alias	Code	Database client software version	UNIX™ shared objects	Microsoft™ Windows™ DLLs	Mac OS X™ dynamic libraries
dbmsnc_11	dbmsnc / sqlserver	snc	SQL Native Client 2012 (V11)	N/A	odbc32.dll / SQLNCLI11.DLL	N/A
dbmftm_0	dbmftm / freetds_sqlserver	ftm	FreeTDS ODBC version 0.82 to 0.95	libtdsodbc.so.0	N/A	N/A
dbmsqt_3	dbmsqt / sqlite	sqt	SQLite 3.x	libsqlite3.so.0	N/A (statically linked)	libsqlite3.dylib

Default database driver

The dbi.default.driver FGLPROFILE entry defines a default database driver to be loaded, if the driver is not specified by the connection parameters.

```
dbi.default.driver = "driver-name"
```

The driver name must be specified without the .so or .DLL extension.

If this configuration entry is not defined, the driver name defaults to dbmdefault.

User name and password (username/password)

In database connection parameters, the username and password parameters define the default database user, when the program uses the DATABASE instruction or the CONNECT TO instruction without the USER/USING clause.

The username and password FGLPROFILE entries are not encrypted. These parameters are provided to simplify migration and should not be used in production. You better use CONNECT TO with a USER / USING clause to avoid any security hole, setup OS user authentication or use the connection callback method. Example of database servers supporting OS user authentication: IBM® Informix®, Oracle and SQL Server.

> **Important: Do not write clear user passwords in your sources! The username and password parameters should be set from a variables.**

For backward compatibility reasons, when using the IBM® Informix® driver, the username / password specification is ignored by the DATABASE instruction, only the CONNECT TO instruction takes external (or callback) login parameters into account.

Connection parameters in database specification

For development or testing purpose, connection parameters can be provided in the database specification string passed to the DATABASE and CONNECT TO instructions. Do not hard code connection specification parameters in programs to be installed on a production site, use the indirect database specification method instead, or build the connection string at runtime, to keep the database connection flexible.

The connection specification parameters override the dbi.database connection parameters defined in FGLPROFILE.

A + plus sign in the database specification starts the list of connection parameters. Each parameter is defined with a name followed by an equal sign an a value enclosed in single quotes. Connection specification parameters must be separated by a comma:

```
dbname+parameter='value'[,...]
```

In this syntax, *parameter* can be one of the following:

Table 156: Connection parameters in the database specification string

Parameter	Description
`resource`	Specifies which '`dbi.database`' entries have to be read from the FGLPROFILE configuration file. When this property is set, the database interface reads `dbi.database.`*`name`*`.*` entries, where *name* is the value specified for the *resource* parameter.
`driver`	Defines the database driver library to be loaded (filename without extension).
`source`	Specifies the data source of the database.
`username`	Defines the name of the database user.
`password`	Defines the password of the database user. **Important: Do not write clear user passwords in your sources! This parameter should be set from a variable value.**

In the next example, `driver`, `source` and `resource` are specified in the connection string:

```
MAIN
  DEFINE db CHAR(50)
  LET db = "stores+driver='dbmora',source='orcl',resource='myconfig'"
  DATABASE db
  ...
END MAIN
```

Direct database specification method

Direct database specification method takes place when the database name used in a `DATABASE` or `CONNECT TO` instruction is not defined in FGLPROFILE with a '`dbi.database.`*`dbname`*`.source`' entry. In this case, the database specification used in the connection instruction is used as the data source.

This method is well known for IBM® Informix® databases, for example to specify the IBM® Informix® server:

```
MAIN
  DATABASE stores@orion
  ...
END MAIN
```

In the next example, the database server is PostgreSQL. The string used in the connection instruction defines the PostgreSQL database (stock), the host (localhost), and the TCP service (5432) the postmaster is listening to. As PostgreSQL syntax is not allowed in the language, a CHAR variable must be used:

```
MAIN
  DEFINE db CHAR(50)
  LET db = "stock@localhost:5432"
  DATABASE db
  ...
```

```
END MAIN
```

This method tieds the compiler programs to a given database server configuration. Prefer indirect database specification method instead of direct database specification.

Indirect database specification method

Indirect database specification method takes place when the database name used in the `DATABASE` or `CONNECT TO` instruction corresponds to a '`dbi.database.`*`dbname`*`.source`' entry defined in the FGLPROFILE configuration file. In this case, the *dbname* database specification is used as a key to read the connection information from the configuration file.

In FGLPROFILE, the entries starting with '`dbi.database`' group information defining connection parameters for indirect database specification:

```
dbi.database.dbname.source   = "value"
dbi.database.dbname.driver   = "value"
dbi.database.dbname.username = "value"
dbi.database.dbname.password = "value"
-- Warning: Password is not encrypted, do not use in production!
```

Keep in mind that FGLPROFILE entry names are converted to lower case when loaded by the runtime system. In order to avoid any mistakes, it is recommended to write FGLPROFILE entry names and program database names in lower case.

In the next example, the program specifies a data source with the name `stores`, and FGLPROFILE defines the *source* and *driver* parameters for the `stores` data source:

Program:

```
MAIN
  DATABASE stores
  ...
END MAIN
```

FGLPROFILE:

```
dbi.database.stores.source   = "stock@localhost:5432"
dbi.database.stores.driver   = "dbmpgs"
```

The indirect database specification technique is a flexible technique to define the database source: The database name in programs is a kind of alias for the real data source, which is defined in an external configuration file (i.e. FGLPROFILE), where entries can be easily changed on production sites without needing program recompilation.

IBM® Informix® emulation parameters in FGLPROFILE

What are Informix SQL emulation settings used for?

To simplify the migration process to other database servers as IBM® Informix®, the database drivers can emulate some IBM® Informix-specific features like SERIAL columns and temporary tables; the drivers can also do some SQL syntax translation.

Avoid using IBM® Informix® emulations; write portable SQL code instead. IBM® Informix® emulations are only provided to help you in the migration process. Disabling IBM® Informix® emulations improves performance, because SQL statements do not have to be parsed to search for IBM® Informix-specific syntax.

Emulations can be controlled with FGLPROFILE parameters. You can disable all possible switches step-by-step, in order to test your programs for SQL compatibility.

`dbi.database.dsname.ifxemul`

This is a global switch to enable or disable IBM® Informix® emulations.

Values can be true or false. Default is true.

```
dbi.database.stores.ifxemul = false
```

`dbi.database.dsname.ifxemul.datatype.type`

The '`ifxemul.datatype`' switches define whether the specified data type must be converted to a native type (for example, when creating a table with the CREATE TABLE statement).

Where *type* can be one of `char, varchar, datetime, decimal, money, float, real, integer, smallint, serial, text, byte, bigint, bigserial, int8, serial8, boolean`.

Default is true for all types.

```
dbi.database.stores.ifxemul.datatype.serial = false
```

`dbi.database.dsname.ifxemul.datatype.serial.emulation`

This parameter can be used to control the SERIAL generation technique used by the driver to generate auto-incremented values.

The value can be one of following:

- `native` uses database's native sequence generator directly in the table definitions (depends on the db type).
- `native2` uses a secondary native sequence generator directly in the table definitions (depends on the db type).
- `regtable` uses the SERIALREG table with triggers. It is slower than the `native` emulation.
- `trigseq`", uses database sequence generator with triggers (not supported by all drivers).

Default is "`native`".

```
dbi.database.stores.ifxemul.datatype.serial.emulation = "native"
```

SERIAL emulations depend on the type of database server used. See SQL adaptation guides on page 529 for more details.

`dbi.database.dsname.ifxemul.temptables`

This switch can be used to control temporary table emulation.

Defaults is true.

```
dbi.database.stores.ifxemul.temptables = false
```

`dbi.database.dsname.ifxemul.temptables.emulation`

This parameter can be used to specify what technique must be used to emulate temporary tables in the database server.

Possible values are "`default`" and "`global`".

```
dbi.database.stores.ifxemul.temptables.emulation = "global"
```

See SQL adaptation guides on page 529 for more details.

dbi.database.*dsname*.ifxemul.dblquotes

This switch can be used to define whether double quoted strings must be converted to single quoted strings.

Default is true.

```
dbi.database.stores.ifxemul.dblquotes = false
```

If this emulation is enabled, all double quoted strings are converted, including database object names.

dbi.database.*dsname*.ifxemul.outers

This switch can be used to control IBM® Informix® `OUTER` translation to native SQL outer join syntax.

Default is true.

```
dbi.database.stores.ifxemul.outers = false
```

Note: Consider using standard ISO outer joins in your SQL statements (LEFT OUTER).

dbi.database.*dsname*.ifxemul.today

This switch can be used to convert the `TODAY` keyword to a native expression returning the current date.

Default is true.

```
dbi.database.stores.ifxemul.today = false
```

dbi.database.*dsname*.ifxemul.current

This switch can be used to convert the `CURRENT X TO Y` expressions to a native expression returning the current time.

Default is true.

```
dbi.database.stores.ifxemul.current = false
```

dbi.database.*dsname*.ifxemul.selectunique

This switch can be used to convert the `SELECT UNIQUE` to `SELECT DISTINCT`.

Default is true.

```
dbi.database.stores.ifxemul.selectunique = false
```

Note: Consider replacing all `UNIQUE` keywords by `DISTINCT`.

dbi.database.*dsname*.ifxemul.colsubs

This switch can be used to control column substrings expressions (col[x,y]) to native substring expressions.

Default is true.

```
dbi.database.stores.ifxemul.colsubs = false
```

Note: Consider using substring SQL functions instead of [x,y] expressions in SQL.

dbi.database.*dsname*.ifxemul.matches

This switch can be used to define whether MATCHES expressions must be converted to LIKE expressions.

Default is true.

```
dbi.database.stores.ifxemul.matches = false
```

Note: Consider using LIKE expressions instead of MATCHES in SQL.

dbi.database.*dsname*.ifxemul.length

This switch can be used to define whether LENGTH( ) function names have to be converted to the native equivalent.

Default is true.

```
dbi.database.stores.ifxemul.length = true
```

dbi.database.*dsname*.ifxemul.rowid

This switch can be used to define whether ROWID keywords have to be converted to native equivalent (for example, OID in PostgreSQL).

Default is true.

```
dbi.database.stores.ifxemul.rowid = false
```

Note: Consider using primary keys instead of ROWIDs.

dbi.database.*dsname*.ifxemul.listupdate

This switch can be used to convert the UPDATE statements using non-ANSI syntax.

Default is true.

```
dbi.database.stores.ifxemul.listupdate = false
```

dbi.database.*dsname*.ifxemul.extend

This switch can be used to convert simple EXTEND( ) expressions to native date/time expressions.

Default is true.

```
dbi.database.stores.ifxemul.extend = true
```

Database vendor specific parameters in FGLPROFILE

Database vendor specific connection parameters can be configured by using FGLPROFILE entries with the following syntax:

```
dbi.database.dsname.dbtype.param.[.subparam] = "value"
```

Where *dbtype* identifies the database vendor type, such as "ifx", "ora", "db2".

- IBM DB2 specific FGLPROFILE parameters on page 470
- Oracle DB specific FGLPROFILE parameters on page 470
- Oracle MySQL / MariaDB specific FGLPROFILE parameters on page 471

- SQL Server (Native Client driver) specific FGLPROFILE parameters on page 471
- SQL Server (Native Client driver) specific FGLPROFILE parameters on page 471
- SQL Server (EasySoft driver) specific FGLPROFILE parameters on page 473
- SQL Server (FreeTDS driver) specific FGLPROFILE parameters on page 472
- Sybase ASE specific FGLPROFILE parameters on page 473

IBM® DB2® specific FGLPROFILE parameters

`dbi.database.`*`dsname`*`.db2.schema`

Name of the database schema to be selected after connection is established.

```
dbi.database.stores.db2.schema = "store2"
```

Set this parameter to a specific schema in order to share the same table with all users.

`dbi.database.`*`dsname`*`.db2.prepare.deferred`

True/False boolean to enable/disable deferred prepare.

```
dbi.database.stores.db2.prepare.deferred = true
```

Set this parameter to true if you do not need to get SQL errors during PREPARE statements: SQL statements will be sent to the server when executing the statement (OPEN or EXECUTE). The default is false (SQL statements are sent to the server when doing the PREPARE).

Default is `false` .

Oracle DB specific FGLPROFILE parameters

`dbi.database.`*`dsname`*`.ora.schema`

Name of the database schema to be selected after connection is established.

```
dbi.database.stores.ora.schema = "store2"
```

Set this parameter to a specific schema in order to share the same table with all users.

`dbi.database.`*`dsname`*`.ora.prefetch.rows`

Maximum number of rows to be pre-fetched.

```
dbi.database.stores.ora.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program. This parameter applies to all cursors in the program.

The default is 10 rows.

`dbi.database.`*`dsname`*`.ora.prefetch.memory`

Maximum buffer size for pre-fetching (in bytes).

```
dbi.database.stores.ora.prefetch.memory = 4096
```

This parameter is equivalent to prefetch.rows, but here you can specify the memory size instead of the number of rows. Like `prefetch.rows`, this parameter applies to all cursors in the program.

The default is 0, which means that memory size is not included in computing the number of rows to pre-fetch.

dbi.database.*dsname*.ora.sid.command

SQL command (SELECT) to generate a unique session id (used for temp table names).

```
dbi.database.stores.ora.sid.command =
 "SELECT TO_CHAR(SID)||'_'||TO_CHAR(SERIAL#)
      FROM V$SESSION WHERE AUDSID=USERENV('SESSIONID')"
```

By default the driver uses "`SELECT USERENV('SESSIONID') FROM DUAL`". This is the standard session identifier in Oracle, but it can become a very large number and can't be reset.

This parameter gives you the freedom to provide your own way to generate a session id.

The SELECT statement must return a single row with one single column.

Value can be an integer or an identifier.

dbi.database.*dsname*.ora.date.ifxfetch

Controls the way an Oracle DATE is fetched into program variables, especially CHAR/VARCHAR targets.

```
dbi.database.stores.ora.date.ifxfetch = true
```

By default, since ORACLE DATE type is equivalent to `DATETIME YEAR TO SECOND`, values are fetched into `CHAR/VARCHAR` with time information and are formatted with the style `YYYY-MM-DD hh:mm:ss` If you need to get the IBM® Informix® behavior, to fetch DATEs only with the YMD part following the DBDATE environment variable, set this parameter to true. Note however that this parameter is useless when fetching ORACLE DATEs into `DATE` or `DATETIME` variables, which is the recommended way to hold date and time values in programs.

Default is `false` (with time information, using normalized format).

Oracle MySQL / MariaDB specific FGLPROFILE parameters

dbi.database.*dsname*.mys.config

Defines an explicit configration to read MySQL options from

```
dbi.database.stores.mys.config = "/opt/myapp/etc/my.cnf"
```

Set this parameter will be passed to the MySQL API function `mysql_options((MYSQL*), MYSQL_READ_DEFAULT_FILE, *filename* )`.

It can be used to bypass the default MySQL configuration files reading, to define database client settings in the `[client]` group, such as the client character set with the `default-character-set` option.

> **Note:**
>
> On Microsoft™ Windows™ platforms, the configuration file must be in DOS format.

SQL Server (Native Client driver) specific FGLPROFILE parameters

dbi.database.*dsname*.snc.logintime

Connection timeout (in seconds).

```
dbi.database.stores.snc.logintime = 5
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.

The default is 5 seconds.

dbi.database.*dsname*.snc.prefetch.rows

Maximum number of rows to be pre-fetched.

```
dbi.database.stores.snc.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program.

The default is 10 rows.

dbi.database.*dsname*.snc.widechar

Control wide char usage for character string data.

Set this parameter to `false` if you use char/varchar columns in the SQL Server database.

```
dbi.database.stores.snc.widechar = false
```

By default the SNC driver uses wide char ODBC functions, by converting the character data from the current locale to UCS/2, by adding the N prefix before string literals and by binding SQL parameters with SQL_C_WCHAR and SQL_WCHAR/SQL_WVARCHAR types.

If you set this parameter to `false`, the driver will pass the character strings as is without character set conversion, leave the string literals without N prefix and bind character string parameters with SQL_C_CHAR and SQL_CHAR/SQL_VARCHAR.

The default is `true` (use wide chars).

SQL Server (FreeTDS driver) specific FGLPROFILE parameters

dbi.database.*dsname*.ftm.logintime

Connection timeout (in seconds).

```
dbi.database.stores.ftm.logintime = 5
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.

The default is 5 seconds.

dbi.database.*dsname*.ftm.prefetch.rows

Maximum number of rows to be pre-fetched.

```
dbi.database.stores.ftm.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program.

The default is 10 rows.

SQL Server (EasySoft driver) specific FGLPROFILE parameters

dbi.database.*dsname*.esm.logintime

Connection timeout (in seconds).

```
dbi.database.stores.esm.logintime = 5
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.

The default is 5 seconds.

dbi.database.*dsname*.esm.prefetch.rows

Maximum number of rows to be pre-fetched.

```
dbi.database.stores.esm.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program.

The default is 10 rows.

Sybase ASE specific FGLPROFILE parameters

dbi.database.*dsname*.ase.logintime

Connection timeout (in seconds).

```
dbi.database.stores.ase.logintime = 10
```

Set this parameter to raise an SQL error if the connection can not be established after the given number of seconds.

The default is 5 seconds.

dbi.database.*dsname*.ase.prefetch.rows

Maximum number of rows to be pre-fetched.

```
dbi.database.stores.ase.prefetch.rows = 50
```

Use this parameter to increase performance by defining the maximum number of rows to be fetched into the db client buffer. However, the bigger this parameter is, the more memory is used by each program.

The default is 10 rows.

Database user authentication

Connecting to a database server is not just specifying a database name: The current user must be identified by the database server. Database users must be declared in the database server and must be authenticated.

The typical user authentication is done by passing a login name and password at connection time. Some database servers support external authentication methods, that do not require login/password information (for example when db users as based on operating system users), as well as delegated user authentication via credential tokens (for example, when using an LDAP dinstinguished name). See database vendor specific documentation for more details.

Additional user authentication solutions are provided to simplify migration from IBM® Informix®databases, but should not be used in production for security reasons.

See also SQL adaptation guides for database vendor specific notes regarding user authentication.

Specifying a user name and password with CONNECT

In order to specify a user name and password, use the `CONNECT` instruction with the `USER/USING` clause:

```
MAIN
  DEFINE uname, upswd STRING
  CALL login_dialog() RETURNING uname, upswd
  CONNECT TO "stock" USER uname USING upswd
  ...
END MAIN
```

This is the recommended way to connect to a database server.

With some database types, it is possible to use an external user authentication service, such as Kerberos / SSL / LDAP-based directory services. To connect as an external user, configure database client settings to authenticate the external user and perform the CONNECT TO instruction without specifying a login/ password:

```
CONNECT TO "stock"
```

For more details, see for example database user handling in the Oracle SQL Adaptation Guide.

Specifying a user name and password with DATABASE

The `DATABASE` instruction does not support the `USER/USING` clause as `CONNECT TO` does. If you don't use an automatic user authentication method of the database server, you must provide a user name and password in some way.

The best way to identify database users is to replace every `DATABASE` instruction by a `CONNECT TO` with `USER/USING` clause. However, it is also possible to provide the user name and password with the user authentication callback function, by defining a global FGLPROFILE entry.

In a development environment, a default login and password can be specified with the `dbi.database.`*`dbname`*`.username` and `dbi.database.`*`dbname`*`.password` FGLPROFILE entries. This solution must not be used in a production environment because the password is not encrypted. For backward compatibility reasons, when using the IBM® Informix® driver, these FGLPROFILE entries are ignored by the `DATABASE` instruction, only the `CONNECT TO` instruction takes external (or callback) login parameters into account.

Login parameters can also be provided in the connection string used in the database name specification in `DATABASE` instruction.

User authentication callback function

When using the `DATABASE` connection instruction, you can define an FGLPROFILE entry with the name of a function to be called when the `DATABASE` instruction is executed, in order to provide a user name and password dynamically.

```
dbi.default.userauth.callback = "[module-name.]function-name"
```

This callback method is not a password encryption solution, it is only provided as workaround to provide a user credentials for programs using the `DATABASE` instructions. If possible, use the `CONNECT TO` instruction with the `USER/USING` clause instead. This callback method is provided to connect to databases different from IBM® Informix®, when a lot of existing code uses the `DATABASE` instruction. With the IBM® Informix® driver, the callback method is also called, but the user name and password are ignored by the `DATABASE` instruction: Only `CONNECT TO` will take the login parameters into account for IBM® Informix®.

The callback function must have the following signature:

```
CALL function-name(dbspec STRING)
      RETURNING STRING (username), STRING (password)
```

If you do not specify the module name, the callback function must be linked to the 42r program. By using the "*module-name.function-name*" syntax in the FGLPROFILE entry, the runtime system will automatically load the module. In both cases, the module must be located in a directory where the runtime system can find it, defined by the FGLLDPATH environment variable.

In the callback function body, the value of *dbspec* can be used to identify the database source, read user name and encrypted password from FGLPROFILE entries with the `fgl_getResource()` function, then decrypt password with the algorithm of your choice and return user name and decrypted password.

User authentication callback function for DATABASE:

```
FUNCTION getUserAuth(dbspec)
  DEFINE dbspec STRING
  DEFINE un, ep STRING
  LET un = fgl_getResource("dbi.database."||
dbspec||".username")
  LET ep = fgl_getResource("dbi.database."||
dbspec||".password.encrypted")
  RETURN un, decrypt_user_password(dbspec, un, ep)
END FUNCTION
```

Order of precedence for database user specification

Database user login can be specified with different methods, as show in this table. Precedence order if defined from top to bottom:

Table 157: Database user login methods

Connection Instruction	FGLPROFILE	Effect

`CONNECT TO "dbname" USER "user" USING "pswd"`
or
`DEFINE db VARCHAR(200)`
`LET db = "dbname +username='username', password='pswd'" DATABASE db`
	N/A (ignored)	The user information in the `USER`/`USING` clause of the `CONNECT TO` instruction or in the connection string of the `DATABASE` instruction are used to identify the actual user. are used to identify the actual user.
		Connection string can also be used with `CONNECT TO`.

`DATABASE dbname`
or
`CONNECT TO "dbname"`
	No specific `dbi.*` entry	No user login and password is provided to the database server. Usually, the Operating System authentication takes place.

`DATABASE dbname` or	`dbi.default.userauth.callback = "fx"`	Callback function `fx` is called to get user name and password when connection instruction is executed.

Connection Instruction	FGLPROFILE	Effect
`CONNECT TO "dbname"`		

`DATABASE dbname`
or
`CONNECT TO "dbname"`
	`dbi.database.dbname.username = ...`
	`dbi.database.dbname.password = ...`
		The FGLPROFILE default user name and password are used to connect to the database server.
		Important: NOT RECOMMENDED IN PRODUCTION!

Unique session mode connection instructions

Opening and closing a database for a unique session.

- DATABASE on page 476
- CLOSE DATABASE on page 477

DATABASE

Opens a new database connection in unique-session mode.

Syntax

```
DATABASE { dbname[@dbserver] | variable | string } [EXCLUSIVE]
```

1. *dbname* identifies the database name.
2. *dbserver* identifies the IBM® Informix® database server (INFORMIXSERVER).
3. *variable* can be any character string defined variable containing the database specification.
4. *string* can be a string literal containing the database specification.

Usage

The DATABASE instruction opens a connection to the database server, like `CONNECT TO`, but without user and password specification.

```
MAIN
  DATABASE stores
  ...
END MAIN
```

It is possible to use a program variable containing the database specification.

```
MAIN
  DEFINE dbname VARCHAR(100)
  LET dbname = arg_val(1)
  DATABASE dbname
  ...
END MAIN
```

If a current connection exists, it is automatically closed before connecting to the new database.

The connection is closed with the `CLOSE DATABASE` instruction, or when the program ends.

The DATABASE instruction raises an exception if the connection could not be established, for example, if you specify a database that the runtime system cannot locate, or cannot open, or for which the user of your program does not have access privileges.

The EXCLUSIVE keyword can be used to open an IBM® Informix® database in exclusive mode to prevent access by anyone but the current user. This keyword is IBM® Informix® specific and should be avoided when writing a portable SQL application.

The CONNECT TO instructions allow better control over database connections; you should use these instructions instead of DATABASE and CLOSE DATABASE.

When used outside a program block, the DATABASE instruction defines the database schema for compilation. See SCHEMA on page 356 for more details.

CLOSE DATABASE

Closes the current database connection created by a DATABASE instruction.

Syntax

```
CLOSE DATABASE
```

Usage

The CLOSE DATABASE instruction closes the current database connection opened by a DATABASE instruction.

The current connection is automatically closed when the program ends.

Example

```
MAIN
  DATABASE stores1
  CLOSE DATABASE
  DATABASE stores2
  CLOSE DATABASE
END MAINs
```

Multi-session mode connection instructions

Opening and closing a database for a unique session.

- CONNECT TO on page 477
- SET CONNECTION on page 478
- DISCONNECT on page 479

CONNECT TO

Opens a new database session in multi-session mode.

Syntax

```
CONNECT TO { dbname | DEFAULT } [ AS session ]
    [ USER login USING auth ]
    [ WITH CONCURRENT TRANSACTION ]
```

1. *dbname* is the database specification.
2. *session* identifies the database session. By default, it is *dbname*.
3. *login* is the name of the database user.
4. *auth* is a string to authenticate the database user, like a password.

Usage

The CONNECT TO instruction opens a database connection. If the instruction successfully connects to the database environment, the connection becomes the current database session for the program.

The session name is case-sensitive.

A program can connect to several database environments at the same time (using different database drivers), and it can establish multiple connections to the same database environment, provided each connection has a unique connection name.

The connection is closed with the DISCONNECT instruction, or when the program ends.

When the USER *login* USING *auth* clause is specified, the database user is identified by *login* and *auth*, ignoring all other user settings defined in FGLPROFILE or as connection string parameters.

The *auth* parameter can be a simple password for internal dabase users, but for some type of database engines, it can be used to specify an external authentication token, such as a distinguished name (DN). For more details, see the SQL adaptation guide of you database type.

The WITH CONCURRENT TRANSACTION clause allows a program to open several transactions concurrently in different database sessions: The transaction can be started with the BEGIN WORK statement in a given connection context, then the program can switch to another connection with SET CONNECTION, and when done, switch back to the first connection to issue a COMMIT WORK or ROLLBACK WORK. This is supported for IBM® Informix® database servers. The option is ignored with other database server types, but it can be used in the CONNECT statement for consistency with Informix.

A CONNECT TO statement cannot be executed with dynamic SQL (i.e. PREPARE + EXECUTE).

With IBM® Informix® database servers, when using the CONNECT TO DEFAULT, you connect to the default IBM® Informix® database server, identified by the INFORMIXSERVER environment variable, without any database selection.

When using IBM® Informix® databases on UNIX™, the only restriction on establishing multiple connections to the same database environment is that an program can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared-memory connection mechanism or the local-loopback connection mechanism, examine the $INFORMIXDIR/etc/sqlhosts file.

Example

```
MAIN
  DEFINE uname, upswd VARCHAR(50)
  CONNECT TO "stores1" -- Session name is "stores1"
  CONNECT TO "stores1" AS "SA" -- Session name is "SA"
  CALL login_dialog() RETURNING uname, upswd
  CONNECT TO "stores2" AS "SB" USER uname USING upswd
END MAIN
```

SET CONNECTION

Selects the current session when in multi-session mode.

Syntax

```
SET CONNECTION {
    { session | DEFAULT } [DORMANT]
    | CURRENT DORMANT }
```

1. *session* is a string expression identifying the name of the database session to be set as current.

Usage

The SET CONNECTION instruction makes a given connection current.

The session name is case-sensitive.

When using the DEFAULT keyword, it identifies the default database server connection established with a CONNECT TO DEFAULT or a DATABASE instruction. This clause is specific to IBM® Informix® databases.

To make the current connection dormant, use CURRENT DORMANT keyword. This clause is specific to IBM® Informix® databases.

A SET CONNECTION statement cannot be executed with dynamic SQL (i.e. PREPARE + EXECUTE).

Example

```
MAIN
  DEFINE c1, c2, c3 INT
  CONNECT TO "stores1"
  CONNECT TO "stores2" AS "SA"
  CONNECT TO "stores3" AS "SB"
  SET CONNECTION "stores1"    -- Select first session
  SELECT COUNT(*) INTO c1 FROM customers
  SET CONNECTION "SA"         -- Select second session
  SELECT COUNT(*) INTO c2 FROM customers
  SET CONNECTION "SB"         -- Select third session
  SELECT COUNT(*) INTO c3 FROM customers
  SET CONNECTION "stores1"    -- Select first session again
END MAIN
```

DISCONNECT

Terminates database sessions when in multi-session mode.

Syntax

```
DISCONNECT { ALL | CURRENT | session }
```

1. *session* is a string expression identifying the name of the database session to be terminated.

Usage

The DISCONNECT instruction closes a given database connection.

The session name is case-sensitive.

When using the DEFAULT keyword, it identifies the default database server connection established with a CONNECT TO DEFAULT or a DATABASE instruction. This clause is specific to IBM® Informix® databases.

Use the ALL keyword to terminate all opened connections. From that point, you must establish a new connection to execute SQL statements.

Use the CURRENT keyword to terminate the current connection only. From that point, in order to execute SQL statements, you must select another connection with SET CONNECTION, or establish a new connection with CONNECT TO.

A DISCONNECT statement cannot be executed with dynamic SQL (i.e. PREPARE + EXECUTE).

If a DISCONNECT statement is used while a database transaction is active, the transaction is automatically rolled back.

Example

```
MAIN
  CONNECT TO  "stores1"  -- Will be identified by "stores1"
  CONNECT TO  "stores1" AS "SA"
  CONNECT TO  "stores2" AS "SB" USER "scott" USING "tiger"
  DISCONNECT "stores1"
  DISCONNECT "SB"
  SET CONNECTION "SA"
END MAIN
```

Miscellaneous SQL statements

These are particular SQL statements supported in the static SQL syntax.

- SET EXPLAIN on page 480
- UPDATE STATISTICS on page 480

SET EXPLAIN

Turns on/off SQL report of the optimizer plan.

Syntax:

```
SET EXPLAIN { ON | OFF }
```

Usage:

Important: This SQL instruction is specific to IBM® Informix® databases.

UPDATE STATISTICS

Updates the statistics for all or for the specified table in the database.

Syntax:

```
UPDATE STATISTICS [ FOR TABLE table-specification ]
```

Usage:

Important: This SQL instruction is specific to IBM® Informix® databases.

Database transactions

Database transaction concepts and handling.

- Understanding database transactions on page 481
- BEGIN WORK on page 482
- SAVEPOINT on page 483
- COMMIT WORK on page 483
- ROLLBACK WORK on page 484
- RELEASE SAVEPOINT on page 484
- SET ISOLATION on page 485
- SET LOCK MODE on page 486

Understanding database transactions

A *database transaction* delimits a set of database operations (i.e. SQL statements), that are processed as a whole.

Database operations included inside a transaction are validated or canceled as a unique operation.

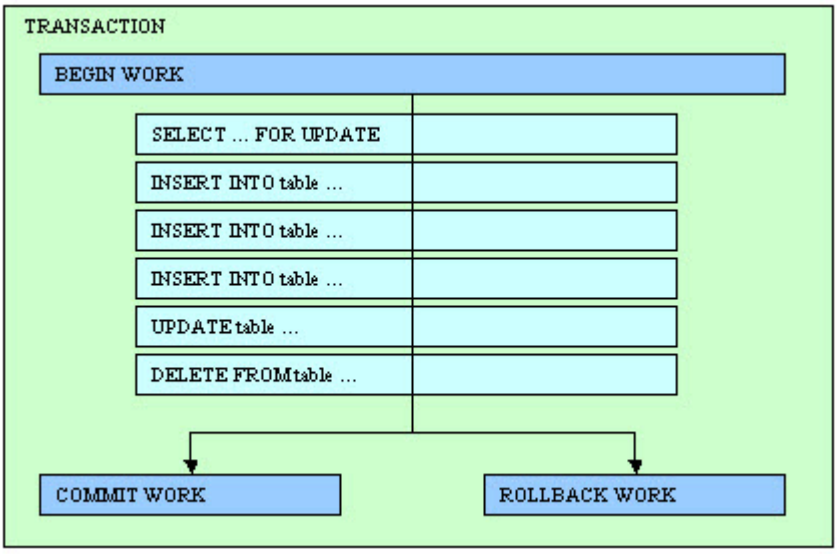


Figure 22: Database transaction

The database server is in charge of *data concurrency* and *data consistency*. Data concurrency allows the simultaneous access of the same data by many users, while data consistency gives each user a consistent view of the database.

Without adequate concurrency and consistency control, data can be changed improperly, compromising integrity of your database. If you want to write applications that can work with different kinds of database servers, you must adapt the program logic to the behavior of the database servers, regarding concurrency and consistency management. This requires good knowledge of multiuser database application programming, transactions, locking mechanisms, isolation levels and wait mode. If you are not familiar with these concepts, carefully read the documentation of each database server that covers this subject.

Usually, database servers set exclusive locks on rows that are modified or deleted inside a transaction. These locks are held until the end of the transaction to control concurrent access to that data. Some database servers implement row versioning (before modifying a row, the server makes a copy of the original row). This technique allows readers to see a consistent copy of the rows that are updated during a transaction not yet committed. When the isolation level is high (REPEATABLE READ) or when using a `SELECT FOR UPDATE` statement, the database server sets shared locks on fetched rows, to prevent other users from changing the rows fetched by the reader. These locks are held until the end of the transaction. Some database servers allow read locks to be held regardless of the transactions (`WITH HOLD` cursor option), but this is not a standard.

Programs accessing the database can change transaction parameters such as the isolation level or lock wait mode. To write portable applications, you must use a configuration that produces the same behavior on every database engine.

The recommended programming pattern regarding transactions is following:

- The database must support transactions; this is usually the case.
- Transactions must be as short as possible (a few seconds).
- The isolation level must be at least `COMMITTED READ`.
- The wait mode for locks must be `WAIT` or `WAIT` n (lock timeout).

To write portable SQL applications, programmers use the `BEGIN WORK`, `COMMIT WORK` and `ROLLBACK WORK` instructions described in this section to delimit transaction blocks and define concurrency parameters with `SET ISOLATION` and `SET LOCK MODE`. These instructions are part of the language syntax. At runtime, the database driver generates the appropriate SQL commands to be used with the target database server. This allows you to use the same source code for different kinds of database servers.

If you initiate a transaction with a `BEGIN WORK` statement, you must issue a `COMMIT WORK` at the end of the transaction. If one of the SQL statement fails in the transaction, you typically issue a `ROLLBACK WORK` to force the database server to cancel any modifications that the transaction made to the database. If you do not issue a `BEGIN WORK` statement to start a transaction, each statement executes within its own transaction. These single-statement transactions do not require either a `BEGIN WORK` statement or a `COMMIT WORK` statement.

Recent database engines support transaction savepoints, which allowing to set markers in the current transaction, in order to rollback to a specific point without canceling the complete transaction. The transaction savepoint instructions `SAVEPOINT`, `ROLLBACK TO SAVEPOINT` and `RELEASE SAVEPOINT` are part of the language syntax and can be directly used in the code.

Some database servers do not support a Data Definition Language (DDL) statements (like `CREATE TABLE`) inside transactions, and some commit automatically the transaction when such a statement is executed. Therefore, it is strongly recommended that you avoid DDL statements inside transactions.

A transaction that processes many rows can exceed the limits that your operating system or the database server configuration imposes on the maximum number of simultaneous locks. Include a limited number of SQL operations in a transaction blocks.

When a program is using several database connections, and if transactions are not terminated before switching to another connection (`SET CONNECTION`), it is mandatory to use the `WITH CONCURRENT TRANSACTION` option in the `CONNECT` instruction.

BEGIN WORK

Starts a database transaction in the current connection.

Syntax

```
BEGIN WORK
```

Usage

Use the `BEGIN WORK` instruction to indicate where the database transaction starts in your program. Each row that an `UPDATE`, `DELETE`, or `INSERT` statement affects during a transaction is locked and remains locked throughout the transaction.

`BEGIN WORK` is part of the language syntax, the underlying database driver executes the native SQL statement corresponding to this SQL instruction.

In order the

Example

The next code example starts a transaction block, inserts a row and updates the row, then commits the transaction. To other users, the `INSERT` and `UPDATE` instruction will be seen as an single atomic database modification:

```
MAIN
  DATABASE stock
  BEGIN WORK
  INSERT INTO items VALUES ( ... )
  UPDATE items SET ...
  COMMIT WORK
```

```
END MAIN
```

SAVEPOINT

Defines or resets the position of a rollback point in the current transaction.

Syntax

```
SAVEPOINT spVname [UNIQUE]
```

1. *spname* is the savepoint identifier.

Usage

The SAVEPOINT instruction declares a new rollback label at the current position in the lexical order within the current transaction. After defining a savepoint, you can rollback to the specified point in the transaction by using the ROLLBACK WORK TO SAVEPOINT instruction.

If the same savepoint name was used in a prior SAVEPOINT instruction, the previous savepoint is destroyed and the name is reused to flag the new rollback position. The optional UNIQUE keyword specifies that you do not want to reuse the same savepoint name in a subsequent SAVEPOINT instruction. Reusing the same name after a SAVEPOINT *spname* UNIQUE will raise an SQL error.

Example

In this example, a first savepoint is defined before the INSERT statement, then reset before the UPDATE statement. The ROLLBACK TO SAVEPOINT instruction will cancel the UPDATE statement only:

```
MAIN
  DATABASE stock
  BEGIN WORK
  DELETE FROM items
  SAVEPOINT sp1
  INSERT INTO items VALUES ( ... )
  SAVEPOINT sp1  -- releases previous savepoint named sp1
  UPDATE items SET ...
  ROLLBACK WORK TO SAVEPOINT sp1
  COMMIT WORK
END MAIN
```

COMMIT WORK

Validates and terminates a database transaction in the current connection.

Syntax

```
COMMIT WORK
```

Usage

Use the COMMIT WORK instruction to commit all modifications made to the database from the beginning of a transaction. The database server takes the required steps to make sure that all modifications that the transaction makes are completed correctly and saved to disk.

COMMIT WORK is part of the language syntax, the underlying database driver executes the native SQL statement corresponding to this SQL instruction.

The COMMIT WORK statement releases all exclusive locks that have been set during the transaction. With some databases, shared locks are not released if the FOR UPDATE cursor is declared WITH HOLD option. However, the COMMIT WORK statement closes all cursors not declared with the WITH HOLD option.

ROLLBACK WORK

Cancels and terminates a database transaction in the current connection.

Syntax

```
ROLLBACK WORK [TO SAVEPOINT [spname]]
```

- *spname* is the savepoint identifier.

Usage

Use ROLLBACK WORK to cancel the current transaction and invalidate all changes since the beginning of the transaction. After the execution of this instruction, the database is restored to the state that it was in before the transaction began. All row and table locks that the canceled transaction holds are released. If you issue this statement when no transaction is pending, an error occurs.

ROLLBACK WORK is part of the language syntax, the underlying database driver executes the native SQL statement corresponding to this SQL instruction.

When specifying a savepoint with the TO SAVEPOINT clause, all SQL statements executed since the specified savepoint will be canceled. The transaction is not canceled, however, and you can continue to execute other SQL statements.

Example

This example checks for a potential SQL error after the DELETE statement and cancels the complete transaction with a ROLLBACK instruction:

```
MAIN
  DATABASE stock
  WHENEVER ERROR CONTINUE
  BEGIN WORK
  INSERT INTO orders_hist VALUES ( ... )
  DELETE FROM orders WHERE ...
  IF SQLCA.SQLCODE < 0 THEN
    ROLLBACK WORK
  ELSE
    COMMIT WORK
  END IF
END MAIN
```

RELEASE SAVEPOINT

Destroys the specified savepoint in the current transaction.

Syntax

```
RELEASE SAVEPOINT spname
```

- *spname* is the savepoint identifier.

Usage

Use the `RELEASE SAVEPOINT` instruction to delete a savepoint defined by the `SAVEPOINT` instruction. See database documentation for more details about the behavior of this SQL statement. Note for example that IBM® Informix® IDS will also release any savepoint that has been declared between the specified savepoint and the `RELEASE SAVEPOINT` instruction.

Example

In the next example, the `RELEASE SAVEPOINT` instruction cancels the `UPDATE` and `INSERT` statements and destroys the sp1 and sp2 savepoints. Only the `DELETE` statement will take effect at the end of the transaction:

```
MAIN
  DATABASE stock
  BEGIN WORK
  DELETE FROM items
  SAVEPOINT sp1
  INSERT INTO items VALUES ( ... )
  SAVEPOINT sp2
  UPDATE items SET ...
  RELEASE SAVEPOINT sp1
  ROLLBACK WORK TO SAVEPOINT
  COMMIT WORK
END MAIN
```

SET ISOLATION

Defines the transaction isolation level for the current connection.

Syntax

```
SET ISOLATION TO
  { DIRTY READ
  | COMMITTED READ [LAST COMMITTED] [RETAIN UPDATE LOCKS]
  | CURSOR STABILITY
  | REPEATABLE READ }
```

Usage

The `SET ISOLATION` instruction sets the transaction isolation level for the current connection. See database concepts in your database server documentation for more details about isolation levels and concurrency management.

When possible, the underlying database driver sets the corresponding transaction isolation level. If the isolation level cannot be set, the runtime system generates an exception.

When using the `DIRTY READ` isolation level, the database server might return a phantom row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. No other isolation level allows access to a phantom row.

On most database servers, the default isolation level is `COMMITTED READ`, which is appropriate to portable database programming.

The `LAST COMMITTED` and `RETAIN UPDATE LOCKS` options have been added to the language syntax for conformance with IBM® Informix® IDS 11. The `LAST COMMITTED` option can be turned on implicitly with a server configuration parameter, saving unnecessary code changes.

Example

```
MAIN
  DATABASE stock
  SET ISOLATION TO COMMITTED READ
  ...
END MAIN
```

SET LOCK MODE

Defines the behavior of the program that tries to access a locked row or table.

Syntax

```
SET LOCK MODE TO { NOT WAIT | WAIT[ seconds ] }
```

Usage

The `SET LOCK MODE` instruction defines the timeout for lock acquisition for the current connection.

When possible, the underlying database driver sets the corresponding connection parameter to define the timeout for lock acquisition. But some database servers may not support setting the lock timeout parameter. In this case, the runtime system generates an exception.

When using the `NOT WAIT` clause, the timeout is set to zero. If the resource is locked, the database server ends the operation immediately and raises an exception with the SQL error.

seconds defines the number of seconds to wait for lock acquisition. If the resource is locked, the database server ends the operation after the elapsed time and raises an exception with the SQL error.

When using the `WAIT` clause without a number of seconds, the database server waits for lock acquisition for an infinite time.

With most database servers, the default is to wait for locks to be released.

Make sure that the database server and corresponding database driver both support a lock acquisition timeout option, otherwise the program will raise an exception.

Example

```
MAIN
  DATABASE stock
  SET LOCK MODE TO WAIT 20
  ...
END MAIN
```

Static SQL statements

Describes static SQL statements supported in the language.

- Understanding static SQL statements on page 487
- Using program variables in static SQL on page 487
- Table and column names in static SQL on page 488
- SQL texts generated by the compiler on page 488
- INSERT on page 489
- DELETE on page 492

- UPDATE on page 490
- SELECT on page 493
- SQL ... END SQL on page 495
- CREATE SEQUENCE on page 496
- ALTER SEQUENCE on page 496
- DROP SEQUENCE on page 497
- CREATE TABLE on page 497
- ALTER TABLE on page 497
- DROP TABLE on page 498
- CREATE INDEX on page 498
- ALTER INDEX on page 499
- DROP INDEX on page 499
- CREATE VIEW on page 499
- DROP VIEW on page 499
- CREATE SYNONYM on page 499
- DROP SYNONYM on page 499
- RENAME on page 500

Understanding static SQL statements

Static SQL statements are SQL instructions that are a part of the language syntax. Static SQL statements can be used directly in the source code as a normal procedural instruction. The static SQL statements are parsed and validated at compile time. At runtime, these SQL statements are automatically prepared and executed by the runtime system.

Program variables can be used inside static SQL statements; Variabales are detected by the compiler and handled as SQL parameters at runtime.

The following example defines two variables that are directly used in an `INSERT` statement:

```
MAIN
  DEFINE iref INTEGER, name CHAR(10)
  DATABASE stock
  LET iref = 65345
  LET name = "Kartopia"
  INSERT INTO item ( item_ref, item_name ) VALUES ( iref, name )
  SELECT item_name INTO name
    FROM item WHERE item_ref = iref
END MAIN
```

Become it is integrated in the language syntax, static SQL statement usage clarifies the source code, but the SQL text is hard-coded and cannot be modified at runtime as it is possible with `PREPARE` / `EXECUTE` instructions.

Limited SQL syntax is part of the language, only common SQL statements such as `INSERT`, `UPDATE`, `DELETE`, `SELECT` are supported.

The compiler supports also `SQL ... END SQL` blocks to write free SQL text in your programs.

Using program variables in static SQL

The syntax of static SQL statements supports the usage of program variables directly as SQL parameters. This gives a better understanding of the source code and requires less lines as when using SQL parameters in dynamic SQL statements.

```
MAIN
  DEFINE c_num INTEGER
  DEFINE c_name CHAR(10)
```

```
  DATABASE stock
  SELECT cust_name INTO c_name FROM customer WHERE cust_num = c_num
END MAIN
```

If a database column name conflicts with a program variable, you can use the @ sign as the column prefix. The compiler will treat the identifier following the @ as a table column:

```
MAIN
  DEFINE cust_name CHAR(10)
  DEFINE cnt INTEGER
  DATABASE stock
  SELECT COUNT(*) INTO cnt FROM customer WHERE @cust_name = cust_name
END MAIN
```

The @ sign will not figure in the resulting SQL statement stored in the .42m compiled module.

Table and column names in static SQL

In static SQL statements, table and column names will be converted to lowercase by the fglcomp compiler. The SQL keywords are always converted to uppercase.

For example:

```
UPDATE CUSTOMER set CUST_name = 'undef' WHERE cust_name is null
```

Will be converted to:

```
UPDATE customer SET cust_name = 'undef' WHERE cust_name IS NULL
```

While SQL keywords are not case sensitive for database servers, table names and column names can be case-sensitive.

You can dump the static SQL statement texts with the `-S` option of fglcomp.

SQL texts generated by the compiler

The fglcomp compiler parses the static SQL statements and modifies them before writing the resulting SQL text to the .42m module.

You can extract all static SQL statements from the source by using the `-S` option of fglcomp:

Example

```
MAIN
  DEFINE c_name CHAR(10)
  DEFINE cnt INTEGER
  DATABASE stock
  SELECT COUNT(*) INTO cnt FROM customer WHERE
 customer.cust_name = c_name
END MAIN
```

```
$ fglcomp -S test.4gl
test.4gl^5^SELECT COUNT(*) FROM customer WHERE cust_name = ?
```

INSERT

Creates a new row in a database table.

Syntax 1:

This is the most standard syntax, working with all type of database engines.

```
INSERT INTO table-specification [ ( column [,...] ) ]
{
 VALUES ( { variable | sql-expression } [,...] )
|
 select-statement
}
```

Syntax 2:

The fglcomp compiler will automatically generate a standard INSERT statement with the complete list of members of the record. The generated SQL will depend from the definition of the record.

```
INSERT INTO table-specification VALUES ( record.* )
```

Syntax 3:

This syntax requires a database schema specification with the SCHEMA instruction, and the corresponding database schema file.

```
INSERT INTO table-specification VALUES record.*
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

1. *dbname* identifies the database name.
2. *dbserver* identifies the database server (INFORMIXSERVER).
3. *owner* identifies the owner of the table, with optional double quotes.
4. *table* is the name of the database table.
5. *column* is a name of a table column.
6. *variable* is a program variable, a record member or an array member used as a parameter buffer to provide values.
7. *sql-expression* is an expression supported by the database server, this can be a literal or NULL for example.
8. *select-statement* is a static SELECT statement with or without parameters as variables.
9. *record* is the name of a record (followed by dot star in this syntax).

Usage

The INSERT SQL statement can be used to create a row i specified database table.

The *dbname*, *dbserver* and *owner* prefix of the table name should be avoided for maximum SQL portability.

When using the VALUES clause, the statement inserts a row in the table with the values specified in variables, as literals, or with NULL. If a record is available, you can specify all record members with the star notation (record.*).

The third syntax can be used to avoid serial column usage in the value list: The record member corresponding to a column defined as SERIAL, SERIAL8 or BIGSERIAL in the schema file will be removed

by the compiler. This is useful when using databases like Microsoft™ SQL Server, where IDENTITY columns must be omitted in `INSERT` statements.

When using a *select-statement*, the statement insert all rows returned in the result set of the `SELECT` statement. The columns returned by the result set must match the column number and data types of the target table. For SQL portability, it is not recommended that you use this syntax.

Example

```
MAIN
   DEFINE myrec RECORD
             key INTEGER,
             name CHAR(10),
             cdate DATE,
             comment VARCHAR(50)
          END RECORD
   DATABASE stock
   LET myrec.key     = 123
   LET myrec.name    = "Katos"
   LET myrec.cdate   = TODAY
   LET myrec.comment = "xxxxxx"
   INSERT INTO items VALUES ( 123, 'Practal', NULL,
 myrec.comment )
   INSERT INTO items VALUES ( myrec.* )
   INSERT INTO items VALUES myrec.*  -- without serial (if one
 is used)
   INSERT INTO items SELECT * FROM histitems WHERE name =
 myrec.name
END MAIN
```

UPDATE

Modifies rows of a database table.

Syntax 1:

This is the most standard syntax, working with all type of database engines.

```
UPDATE table-specification
   SET
       column = { variable | sql-expression }
       [,...]
   [  sql-condition ]
```

Syntax 2:

This syntax is not standard, but will be converted by compiler to a portable UPDATE syntax.

```
UPDATE table-specification
   SET ( column [,...] )
     = ( { variable | sql-expression } [,...] )
   [  sql-condition ]
```

Syntax 3:

This syntax is not portable, and is not converted by the compiler.

```
UPDATE table-specification
   SET [table.]*
```

```
      = ( { variable | sql-expression } [,...] )
   [  sql-condition ]
```

Syntax 4:

The last syntax requires a database schema specification with SCHEMA instruction, and the corresponding database schema file.

```
UPDATE table-specification
   SET { [table.]* | ( column [,...] ) }
     = record.*
   [  sql-condition ]
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

And *sql-condition* is:

```
WHERE { condition | CURRENT OF cursor }
```

1. *dbname* identifies the database name.
2. *dbserver* identifies the database server (INFORMIXSERVER).
3. *owner* identifies the owner of the table, with optional double quotes.
4. *table* is the name of the database table.
5. *column* is a name of a table column.
6. *variable* is a program variable, a record member or an array member used as a parameter buffer to provide values.
7. *sql-expression* is an expression supported by the database server, this can be a literal or NULL for example.
8. *record* is the name of a record (followed by dot star in this syntax).
9. *condition* is an SQL expression to select the rows to be updated.
10. *cursor* is the identifier of a database cursor.

Usage

The UPDATE SQL statement can be used to modify one or more rows in the specified database table.

The *dbname*, *dbserver* and *owner* prefix of the table name should be avoided for maximum SQL portability.

The third syntax should be avoided, this syntax is not standard and will not work will all database types.

The fourth syntax can be used if the database schema file has been generated with the correct data types. This is especially important when using SERIAL columns or equivalent auto-incremented columns. The fglcomp compiler will automatically extend the SQL text with the columns identified by the record variable. The columns defined in the database schema file as SERIAL (code 262) will be omitted in the generated column list.

column with a subscript expression (`column[a,b]`) is not recommended because most database servers do not support this notation.

For more details about the `WHERE CURRENT OF` clause, see Positioned updates/deletes on page 514.

Example

```
MAIN
   DEFINE myrec RECORD
             key INTEGER,
             name CHAR(10),
```

```
            cdate DATE,
            comment VARCHAR(50)
         END RECORD
   DATABASE stock
   LET myrec.key     = 123
   LET myrec.name    = "Katos"
   LET myrec.cdate   = TODAY
   LET myrec.comment = "xxxxxx"
   UPDATE items SET
        name    = myrec.name,
        cdate   = myrec.cdate,
        comment = myrec.comment
    WHERE key = myrec.key
END MAIN
```

DELETE

Removes rows from a database table.

Syntax

```
DELETE FROM table-specification
   [ WHERE { condition | CURRENT OF cursor } ]
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

1. *dbname* identifies the database name.
2. *dbserver* identifies the database server (INFORMIXSERVER).
3. *owner* identifies the owner of the table, with optional double quotes.
4. *table* is the name of the database table.
5. *condition* is an SQL expression to select the rows to be deleted.
6. *cursor* is the identifier of a database cursor.

Usage

The `DELETE` SQL statement can be used to delete one or more rows from the specified database table.

The *dbname*, *dbserver* and *owner* prefix of the table name should be avoided for maximum SQL portability.

If you do not specify the `WHERE` clause, all rows in the table will be deleted. No warning will be generated by the compiler.

For more details about the `WHERE CURRENT OF` clause, see Positioned updates/deletes on page 514.

Example

```
MAIN
   DATABASE stock
   DELETE FROM items WHERE name LIKE 'A%'
END MAIN
```

SELECT

Produces a result set from a query on database tables.

Syntax

```
select-statement [ UNION [ALL] select-statement
] [...]
```

where *select-statement* is:

```
SELECT [subset-clause] [duplicates-option] { * | select-list }
  [ INTO variable [,...] ]
  FROM table-list [,...]
  [ WHERE condition ]
  [ GROUP BY column-list [ HAVING condition ]]
  [ ORDER BY column [{ASC|DESC}] [,...] ]
```

where *subset-clause* is:

```
[ SKIP { integer | variable }]
[ {FIRST|MIDDLE|LIMIT} { integer | variable ]
```

where *duplicates-option* is:

```
{ ALL
| DISTINCT
| UNIQUE
}
```

where *select-list* is:

```
{ [@]table-specification.*
| [table-specification.]column
| literal
} [ [AS] column-alias ]
[,...]
```

where *table-list* is:

```
{ table-name
| OUTER table-name
| OUTER ( table-name [,...] )
}
[,...]
```

where *table-name* is:

```
table-specification [ [AS] table-alias]
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

where *column-list* is:

```
column-name [,...]
```

where *column-name* is:

```
[table.]column
```

1. *dbname* identifies the database name.
2. *dbserver* identifies the database server (INFORMIXSERVER).
3. *owner* identifies the owner of the table, with optional double quotes.
4. *table* is the name of the database table.
5. *table-alias* defines a new name to reference the *table* in the rest of the statement.
6. *integer* is an integer constant.
7. *variable* is a program variable.
8. *column* is a name of a table column.
9. *column-alias* defines a new name to reference the *column* in the rest of the statement.
10. *condition* is an SQL expression to select the rows to be deleted.

Usage

The *dbname*, *dbserver* and *owner* prefix of the table name should be avoided for maximum SQL portability.

If the SELECT statement returns only one row of data, you can write it directly as a procedural instruction. However, you must use the INTO clause to provide the list of variables where column values will be fetched. The INTO clause provides the list of fetch buffers. This clause is not part of the SQL language sent to the database server; it is extracted from the statement by the compiler.

```
MAIN
   DEFINE myrec RECORD
            key INTEGER,
            name CHAR(10),
            cdate DATE,
            comment VARCHAR(50)
         END RECORD
   DATABASE stock
   LET myrec.key = 123
   SELECT name, cdate
     INTO myrec.name, myrec.cdate
     FROM items
     WHERE key=myrec.key
END MAIN
```

If the SELECT statement returns more than one row of data, you must declare a database cursor to process the result set.

```
MAIN
   DEFINE myrec RECORD
            key INTEGER,
            name CHAR(10),
            cdate DATE,
            comment VARCHAR(50)
         END RECORD
   DATABASE stock
   LET myrec.key = 123
   DECLARE c1 CURSOR FOR
     SELECT name, cdate
       FROM items
       WHERE key=myrec.key
   OPEN c1
   FETCH c1 INTO myrec.name, myrec.cdate
   CLOSE c1
END MAIN
```

The SELECT statement can include the INTO clause, but it is strongly recommended that you use that clause in the FETCH instruction only.

The SELECT INTO TEMP statement creates temporary tables. Such statement does not return a result set.

SQL ... END SQL

Performs an SQL that is not part of the static SQL syntax.

Syntax

```
SQL
 sql-statement
END SQL
```

where *sql-statement* is:

```
  sql-keyword
| identifier
| INTO $host-variable [,...]
| $host-variable
| {+ sql-directive }
| --+ sql-directive
| --# fgl-comment
[...]
```

1. *sql-keyword* is any keyword of the SQL language.
2. *identifier* is a regular SQL identifier such as a table or column name.
3. *host-variable* is a program variable defined in the current scope.
4. *sql-directive* is a special comment to be kept in the SQL statement.
5. *fgl-comment* defines a comment that will be interpreted as a regular syntax element.

Usage

SQL blocks provide a convenient way to execute specific SQL statements that are not supported in the language as static SQL statements.

SQL blocks start with the SQL keyword and end with the END SQL keywords. The content of the SQL block is parsed by the fglcomp compiler to extract host variables, but the SQL statement syntax is not checked. This is actually the main purpose of SQL blocks, compared to regular static SQL statements; with SQL blocks, you can use any recent SQL statement introduced by the latest version of your database server. Note however, that you can achieve the same result using dynamic SQL instructions.

Only one SQL statement can be included in an SQL block. Using the ; semicolon statement separator is forbidden.

Program variables can be used inside the SQL statement. However, unlike static SQL statements, each host variable must be identified with a $ dollar prefix. The list of fetch targets must be preceded by the INTO keyword, as in static SELECT statements. Complete records can be used in SQL blocks by using the dot star notation ($record.*), you can also use the THROUGH or THRU keywords), as well as array elements.

SQL blocks can also be used to declare a cursor with the DECLARE mycursor CURSOR FOR SQL ... END SQL syntax.

SQL directives can be used inside SQL blocks as special comments with the {+} or --+ syntax. The SQL directives will be kept in the SQL text that will be executed by the database server. You typically write optimizer hints with the SQL directives syntax.

The `--#` specific comment is supported for backward compatibility. The SQL text following this marker will be parsed as regular SQL text, but will be ignored by other compilers. It is not recommended to use this feature.

You can check the resulting SQL statement after parsing by using the `-S` option of fglcomp.

Example

```
MAIN
   DEFINE myrec RECORD
             key INTEGER,
             name CHAR(10)
          END RECORD
   DATABASE stock
   LET myrec.key = 123
   SQL
     SELECT (+EXPLAIN) items.* INTO $myrec.*
        FROM items WHERE key=$myrec.key
   END SQL
END MAIN
```

CREATE SEQUENCE

Creates a new sequence object in the database.

Syntax:

```
CREATE SEQUENCE [ IF NOT EXISTS ] sequence-name
[ INCREMENT BY integer
| START WITH integer
| NOMAXVALUE
| MAXVALUE integer
| NOMINVALUE
| MINVALUE integer
| CYCLE
| NOCYCLE
| CACHE integer
| NOCACHE
| ORDER
| NOORDER
]
```

ALTER SEQUENCE

Modifies the definition of an existing sequence in the database.

Syntax:

```
ALTER SEQUENCE sequence-name
[ INCREMENT BY integer
| RESTART WITH integer
| NOMAXVALUE
| MAXVALUE integer
| NOMINVALUE
| MINVALUE integer
| CYCLE
| NOCYCLE
| CACHE integer
| NOCACHE
| ORDER
```

```
| NOORDER
]
```

DROP SEQUENCE

Drops a sequence object from the database.

Syntax:

```
DROP SEQUENCE [ IF EXISTS ] sequence-name
```

CREATE TABLE

Creates a new table object in the database.

Syntax:

```
CREATE [TEMP] TABLE [ IF NOT EXISTS ] table-specification
(
[ column-name data-type
       [ DEFAULT default-value ] [ NOT NULL ]
       [ PRIMARY KEY [ contraint-name ]
       | UNIQUE [ contraint-name ]
       | CHECK ( sql-condition ) [ contraint-name ]
       | REFERENCES table-name
             [ ( column-name [,...] ) ]
             [ ON DELETE CASCADE ]
             [ contraint-name ]
       ]
| PRIMARY KEY ( column-name [,...] ) [ contraint-name ]
| UNIQUE ( column-name [,...] ) [ contraint-name ]
| CHECK ( sql-condition ) [ contraint-name ]
| FOREIGN KEY ( column-name [,...] )
    REFERENCES table-name
             [ ( column-name [,...] ) ]
             [ ON DELETE CASCADE ]
             [ contraint-name ]
] [,...]
)
[ WITH NO LOG ]
[ IN tablespace-name ]
[ EXTENT SIZE integer ]
[ NEXT SIZE integer ]
[ LOCK MODE { PAGE | ROW } ]
```

ALTER TABLE

Modifies the definition of an existing table in the database.

Syntax:

```
ALTER TABLE table-specification
(
[ DROP ( column-name [,...] )
| ADD ( column-name data-type
          [ DEFAULT default-value ] [ NOT NULL ]
          [ PRIMARY KEY [ contraint-name ]
          | UNIQUE [ contraint-name ]
          | CHECK ( sql-condition ) [ contraint-name ]
          | REFERENCES table-name
```

```
                [ ( column-name [,...] ) ]
                [ ON DELETE CASCADE ]
                [ contraint-name ]
          ]
          [ BEFORE column-name
          [,...]
        )
| MODIFY ( column-name data-type
          [ DEFAULT default-value ] [ NOT NULL ]
          [ PRIMARY KEY [ contraint-name ]
          | UNIQUE [ contraint-name ]
          | CHECK ( sql-condition ) [ contraint-name ]
          | REFERENCES table-name
                [ ( column-name [,...] ) ]
                [ ON DELETE CASCADE ]
                [ contraint-name ]
          ]
          [,...]
        )
| DROP CONSTRAINT constraint-name
| ADD CONSTRAINT
    { PRIMARY KEY ( column-name [,...] ) [ contraint-name ]
    | UNIQUE ( column-name [,...] ) [ contraint-name ]
    | CHECK ( sql-condition ) [ contraint-name ]
    | FOREIGN KEY ( column-name [,...] )
         REFERENCES table-name
             [ ( column-name [,...] ) ]
             [ ON DELETE CASCADE ]
             [ contraint-name ]
    }
| LOCK MODE ( { PAGE | ROW } )
| MODIFY NEXT SIZE integer
] [,...]
)
```

DROP TABLE

Drops a table object from the database.

Syntax:

```
DROP TABLE [ IF EXISTS ] table-specification
```

CREATE INDEX

Creates a new index object in the database.

Syntax:

```
CREATE [ UNIQUE | CLUSTER | UNIQUE CLUSTER ] INDEX
       [ IF NOT EXISTS ] index-name
 ON table-specification
    ( column-name [ ASCENCDING | DESCENDING ]  [,...] )
```

ALTER INDEX

Modifies the definition of an existing index in the database.

Syntax:

```
ALTER INDEX index-name TO [ NOT ] CLUSTER
```

DROP INDEX

Drops an index object from the database.

Syntax:

```
DROP INDEX [ IF EXISTS ] index-name
```

CREATE VIEW

Creates a new view object in the database.

Syntax:

```
CREATE VIEW [ IF NOT EXISTS ] view-name
[ ( column-alias-name [,...] ) ]
    AS sub-query
[ WITH CHECK OPTION ]
```

where *sub-query* is a limited syntax of the `SELECT` statement.

DROP VIEW

Drops a view object from the database.

Syntax:

```
DROP VIEW [ IF EXISTS ] view-name
```

CREATE SYNONYM

Creates a new synonym object in the database.

Syntax:

```
CREATE SYNONYM [ IF NOT EXISTS ] synonym-name
   FOR table-specification
```

DROP SYNONYM

Drops a synonym object from the database.

Syntax:

```
DROP SYNONYM [ IF EXISTS ] synonym-name
```

RENAME

Renames an object in the database.

Syntax:

```
RENAME { TABLE | COLUMN | INDEX | SEQUENCE }
    old-name TO new-name
```

Dynamic SQL management

Explains how to execute and manage SQL statements at runtime.

- Understanding dynamic SQL on page 500
- PREPARE (SQL statement) on page 501
- EXECUTE (SQL statement) on page 502
- FREE (SQL statement) on page 503
- EXECUTE IMMEDIATE on page 504

Understanding dynamic SQL

Basic SQL instructions are part of the language syntax as static SQL statements, but only a limited number of SQL instructions are supported this way. *Dynamic SQL* management allows you to execute any kind of SQL statement, hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.

In order to execute an SQL statement dynamically, you must first prepare the SQL statement to initialize a *statement handle*, then you execute the prepared statement one or more times:

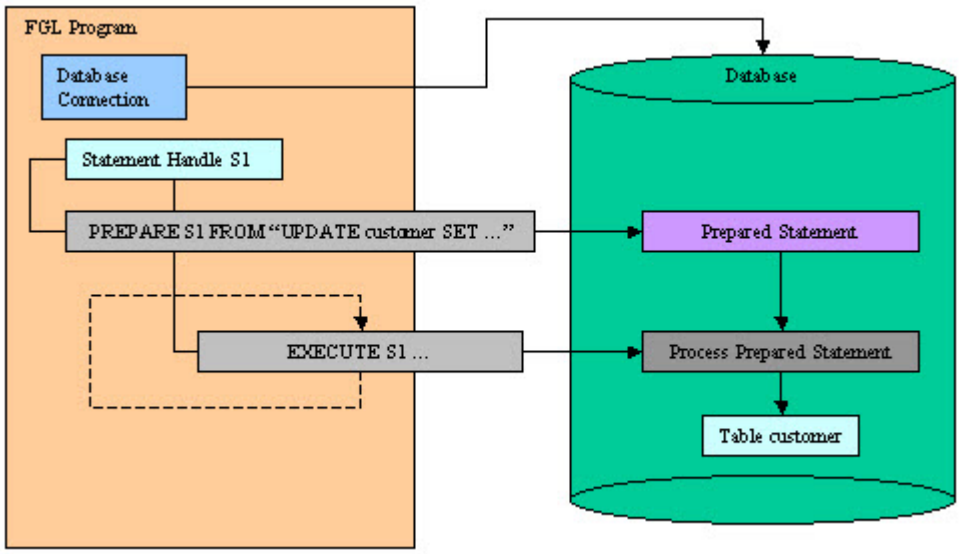


Figure 23: Dynamic SQL management diagram

When you no longer need the prepared statement, you can free the statement handle to release allocated resources:

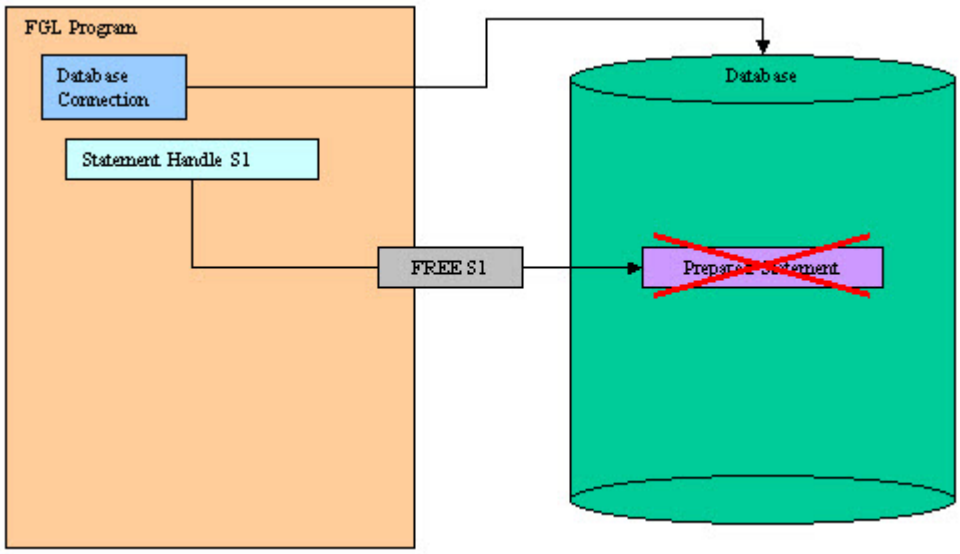


Figure 24: FREE statement diagram

When using insert cursors or SQL statements that produce a result set (like `SELECT`), you must declare a cursor with a prepared statement handle.

Prepared SQL statements can contain SQL parameters by using ? placeholders in the SQL text. In this case, the `EXECUTE` or `OPEN` instruction supplies input values in the `USING` clause.

To increase performance efficiency, you can use the `PREPARE` instruction, together with an `EXECUTE` instruction in a loop, to eliminate overhead caused by redundant parsing and optimizing. For example, an `UPDATE` statement located within a `WHILE` loop is parsed each time the loop runs. If you prepare the `UPDATE` statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution.

PREPARE (SQL statement)

Prepares an SQL statement for execution.

Syntax

```
PREPARE sid FROM sqltext
```

1. *sid* is an identifier to handle the prepared SQL statement.
2. *sqltext* is a string expression containing the SQL statement to be prepared.

Usage

The `PREPARE` instruction allocates resources for an SQL statement handle, in the context of the current database connection. The SQL text is sent to the database server for parsing, validation and to generate the execution plan.

Prepared SQL statements can be executed with the `EXECUTE` instruction, or, when the SQL statement generates a result set, the prepared statement can be used to declare cursors with the `DECLARE` instruction.

A statement identifier (*sid*) can represent only one SQL statement at a time. You can execute a new `PREPARE` instruction with an existing statement identifier if you wish to assign the text of a different SQL statement to the statement identifier. The scope of reference of the *sid* statement identifier is local to the module where it is declared. That is, the identifier of a statement that was prepared in one module cannot be referenced from another module.

The SQL statement can have parameter placeholders, identified by the question mark (?) character. You cannot directly reference a variable in the text of a prepared SQL statement. You cannot use question mark (?) placeholders for SQL identifiers such as a table name or a column name; you must specify these identifiers in the statement text when you prepare it.

Resources allocated by PREPARE can be released later by the FREE instruction.

The number of prepared statements in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the prepared statement.

Some database servers support multiple SQL statement preparation in a unique PREPARE instruction, but most database servers deny multiple statements. You should only prepare one SQL statement at a time.

Example

```
FUNCTION deleteOrder(n)
  DEFINE n INTEGER
  PREPARE s1 FROM "DELETE FROM order WHERE key=?"
  EXECUTE s1 USING n
  FREE s1
END FUNCTION
```

See EXECUTE for more code examples.

EXECUTE (SQL statement)

This instruction runs an SQL statement previously prepared.

Syntax

```
EXECUTE sid
     [ USING pvar {IN|OUT|INOUT} [,...] ]
     [ INTO fvar [,...] ]
```

1. *sid* is an identifier to handle the prepared SQL statement.
2. *pvar* is a variable containing an input value for an SQL parameter.
3. *fvar* is a variable used as fetch buffer.

Usage

The EXECUTE instruction performs the execution of a prepared SQL statement. Once prepared, an SQL statement can be executed as often as needed.

If the SQL statement has (?) parameter placeholders, you must specify the USING clause to provide a list of variables as parameter buffers. Parameter values are assigned by position.

If the SQL statement returns a result set with one row, you can specify the INTO clause to provide a list of variables to receive the result set column values. Fetched values are assigned by position. If the SQL statement returns a result set with more than one row, the instruction raises an exception.

The IN, OUT or INOUT options can only be used for simple variables, you cannot specify those options for a complete record with the record.* notation.

The IN, OUT or INOUT options can be used to call stored procedures having input / output parameters. Use the IN, OUT or INOUT options to indicate if a parameter is respectively for input, output or both.

You cannot execute a prepared SQL statement based on database tables if the table structure has changed (ALTER TABLE) since the PREPARE instruction; you must re-prepare the SQL statement.

Example

```
MAIN
  DEFINE var1 CHAR(20)
  DEFINE var2 INTEGER

  DATABASE stores

  PREPARE s1 FROM "UPDATE tab SET col=? WHERE key=?"
  LET var1 = "aaaa"
  LET var2 = 345
  EXECUTE s1 USING var1, var2

  PREPARE s2 FROM "SELECT col FROM tab WHERE key=?"
  LET var2 = 564
  EXECUTE s2 USING var2 INTO var1

  PREPARE s3 FROM "CALL myproc(?,?)"
  LET var1 = 'abc'
  EXECUTE s3 USING var1 IN, var2 OUT

END MAIN
```

FREE (SQL statement)

Releases the resources allocated to a prepared statement.

Syntax

```
FREE sid
```

1. *sid* is the identifier of the prepared SQL statement.

Usage

The FREE instruction takes the name of a statement as parameter.

All resources allocated to the SQL statement handle are released.

After resources are released, the statement identifier cannot be referenced by a cursor, or by the EXECUTE statement, until you prepare the statement again.

Free the statement if it is not needed anymore, this saves resources on the database client and database server side.

Example

```
FUNCTION update_customer_name( key, name )
  DEFINE key INTEGER
  DEFINE name CHAR(10)
  PREPARE s1 FROM "UPDATE customer SET name=? WHERE
 customer_num=?"
  EXECUTE s1 USING name, key
  FREE s1
END FUNCTION
```

EXECUTE IMMEDIATE

Performs a simple SQL execution without SQL parameters or result set.

Syntax

```
EXECUTE IMMEDIATE sqltext
```

1. *sqltext* is a string expression containing the SQL statement to be executed.

Usage

The `EXECUTE IMMEDIATE` instruction passes an SQL statement to the database server for execution in the current database connection.

The SQL statement used by `EXECUTE IMMEDIATE` must be a single statement without SQL parameters and must not produce a result set.

This instruction is equivalent to `PREPARE`, `EXECUTE` and `FREE` in one step.

Example

```
MAIN
  DATABASE stores
  EXECUTE IMMEDIATE "UPDATE tab SET col='aaa' WHERE key=345"
END MAIN
```

Result set processing

Shows how to fetch rows from a database query.

- Understanding database result sets on page 504
- DECLARE (result set cursor) on page 506
- OPEN (result set cursor) on page 509
- FETCH (result set cursor) on page 510
- CLOSE (result set cursor) on page 511
- FREE (result set cursor) on page 512
- FOREACH (result set cursor) on page 512

Understanding database result sets

A *database result set* is a group of rows produced by an SQL statement such as `SELECT`. The result set is maintained by the database server. In a program, you handle a result set with a *database cursor*.

First you must declare the database cursor with the `DECLARE` instruction. This instruction sends the SQL statement to the database server for parsing, validation and to generate the execution plan.

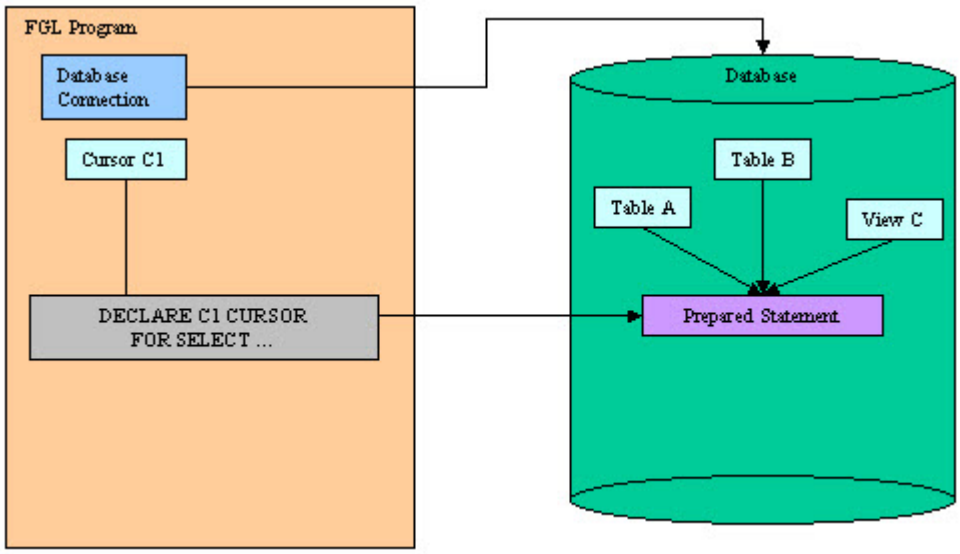


Figure 25: Database result set

The result set is produced after execution of the SQL statement, when the database cursor is associated with the result set by the OPEN instruction. At this point, no data rows are transmitted to the program. You must use the FETCH instruction to retrieve data rows from the database server.

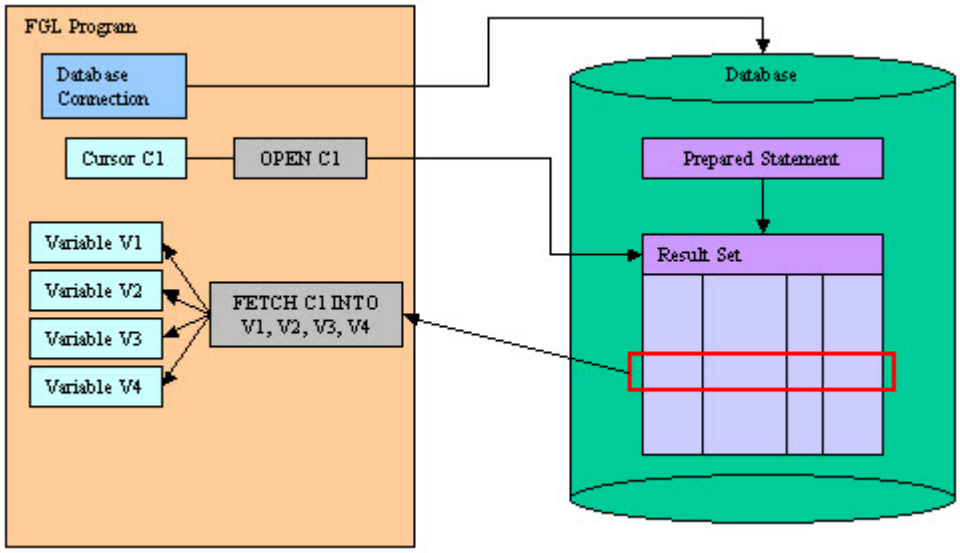


Figure 26: FETCH instruction

When finished with the result set processing, you must CLOSE the cursor to release the resources allocated for the result set on the database server. The cursor can be reopened if needed. If the SQL statement is no longer needed, you can free the resources allocated to statement execution with the FREE instruction.

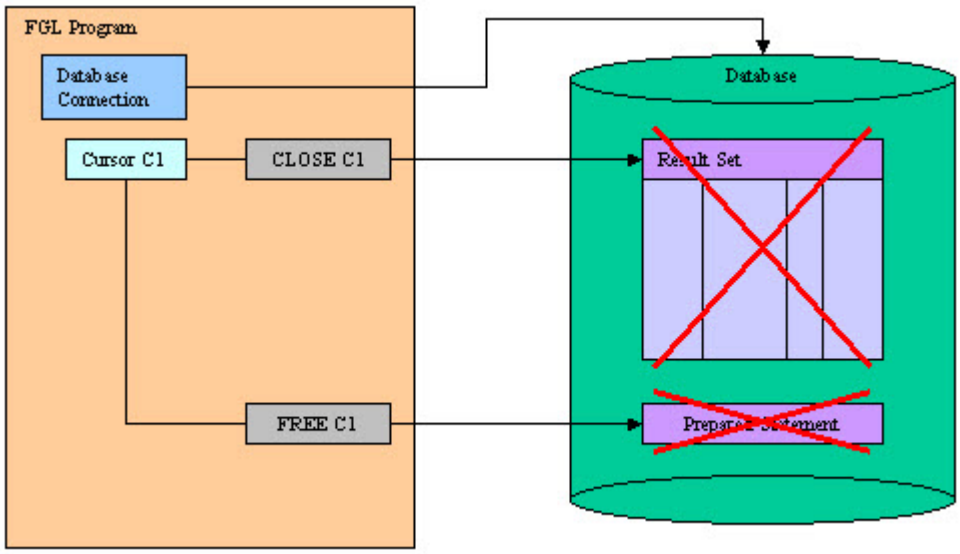


Figure 27: FREE instruction

The scope of reference of a database cursor is local to a module, so a cursor that was declared in one source file cannot be referenced in a statement in another file.

The language supports *sequential cursors* and *scrollable cursors*. Sequential cursors, which are unidirectional, are used to retrieve rows for a `REPORT`, for example. Scrollable cursors allow you to move backwards or to an absolute or relative position in the result set. Specify whether a cursor is scrollable with the `SCROLL` option of the `DECLARE` instruction.

DECLARE (result set cursor)

Associates a database cursor with an SQL statement producing a result set.

Syntax 1: Cursor declared with a static SQL statement.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR select-statement
```

Syntax 2: Cursor declared with a prepared statement.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR sid
```

Syntax 3: Cursor declared with a string expression.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FROM expr
```

Syntax 4: Cursor declared with an SQL Block.

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD] FOR SQL sql-statement END SQL
```

1. *cid* is the identifier of the database cursor.
2. *select-statement* is a `SELECT` statement defined in static SQL.
3. *sid* is the identifier of a prepared SQL statement.
4. *expr* is any expression that evaluates to a string.
5. *sql-statement* is a statement defined in an SQL block.

Usage

The DECLARE instruction allocates resources for an SQL statement handle, in the context of the current connection. The SQL text is sent to the database server for parsing, validation and to generate the execution plan.

After declaring the cursor, you can use the OPEN instruction to execute the SQL statement and produce the result set.

DECLARE must precede any other statement that refers to the cursor during program execution.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared.

Resources allocated by the DECLARE *cursor-name* can be released later by the FREE *cursor-name* instruction.

The static *select-statement* used in the DECLARE can contain ? (question mark) parameter placeholders, that can be bound to program variables with the USING clause of the OPEN instruction.

The maximum number of declared cursors in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the declared cursor.

When declaring a cursor with a static *select-statement*, the statement can include an INTO clause. However, to be consistent with prepared statements you better omit the INTO clause in the SQL text and use the INTO clause of the FETCH statement to retrieve the values from the result set.

You can add the FOR UPDATE clause in the SELECT statement to declare an update cursor. You can use the update cursor to modify (update or delete) the current row.

You should use the WITH HOLD option with care, because this feature is specific to IBM® Informix® servers. Other database servers do not behave as Informix® does with this type of cursor. For example, if the SELECT is not declared FOR UPDATE, most database servers keep cursors open after the end of a transaction, but IBM® DB2® automatically closes all cursors when the transaction is rolled back.

Forward only cursors

If you use only the DECLARE CURSOR keywords, you create a *sequential cursor*, which can fetch only the next row in sequence from the result set. The sequential cursor can read through the result set only once each time it is opened. If you are using a sequential cursor for a select cursor, on each execution of the FETCH statement, the database server returns the contents of the current row and locates the next row in the result set.

Cursors can be declare with a static SELECT statement:

```
MAIN
   DATABASE stores
   DECLARE c1 CURSOR FOR SELECT * FROM customer
END MAIN
```

Cursors can also be declared with a SELECT statement defined in a character string:

```
MAIN
  DEFINE key INTEGER
  DEFINE cust RECORD
            num INTEGER,
            name CHAR(50)
         END RECORD
  DATABASE stores
  PREPARE s1
     FROM "SELECT customer_num, cust_name FROM customer WHERE
 customer_num>?"
  DECLARE c1 CURSOR FOR s1
  LET key=101
  FOREACH c1 USING key INTO cust.*
```

```
     DISPLAY cust.*
  END FOREACH
END MAIN
```

Scrollable cursors

Use the `DECLARE SCROLL CURSOR` keywords to create a *scrollable cursor*, which can fetch rows of the result set in any sequence. Until the cursor is closed, the database server retains the result set of the cursor in a static data set (for example, in a temporary table like Informix®). You can fetch the first, last, or any intermediate rows of the result set as well as fetch rows repeatedly without having to close and reopen the cursor. On a multiuser system, the rows in the tables from which the result set rows were derived might change after the cursor is opened and a copy of the row is made in the static data set. If you use a scroll cursor within a transaction, you can prevent copied rows from changing, either by setting the isolation level to `REPEATABLE READ` or by locking the entire table in share mode during the transaction. Scrollable cursors cannot be declared `FOR UPDATE`.

With most database servers, scrollable cursors take quite a few resources to hold a static copy of the result set. Therefore you should consider optimizing scrollable cursor usage by fetching only the primary keys of rows, and execute a secondary `SELECT` statement to fetch other fields for each row that must be displayed.

The `DECLARE [SCROLL] CURSOR FROM` syntax allows you to declare a cursor directly with a string expression, so that you do not have to use the `PREPARE` instruction. This simplifies the source code and speeds up the execution time for non-Informix databases, because the SQL statement is not parsed twice.

```
MAIN
  DEFINE key INTEGER
  DEFINE cust RECORD
            num INTEGER,
            name CHAR(50)
        END RECORD
  DATABASE stores
  DECLARE c1 SCROLL CURSOR
     FROM "SELECT customer_num, cust_name FROM customer WHERE
 customer_num>?"
  LET key=101
  FOREACH c1 USING key INTO cust.*
     DISPLAY cust.*
  END FOREACH
END MAIN
```

Hold cursors

Use the `WITH HOLD` option with Informix® databases to create a *hold cursor*. A hold cursor allows uninterrupted access to a set of rows across multiple transactions. Ordinarily, all cursors close at the end of a transaction. A hold cursor does not close; it remains open after a transaction ends. A hold cursor can be either a sequential cursor or a scrollable cursor. Hold cursors are only supported by Informix® database engines.

You can use the ? question mark place holders with prepared or static SQL statements, and provide the parameters at execution time with the `USING` clause of the `OPEN` or `FOREACH` instructions.

```
MAIN
  DEFINE key INTEGER
  DEFINE cust RECORD
            num INTEGER,
            name CHAR(50)
        END RECORD
  DATABASE stores
  DECLARE c1 CURSOR WITH HOLD
```

```
     FOR SELECT customer_num, cust_name FROM customer WHERE customer_num > ?
  LET key=101
  FOREACH c1 USING key INTO cust.*
     BEGIN WORK
     UPDATE cust2 SET name=cust.cust_name WHERE num=cust.num
     COMMIT WORK
  END FOREACH
END MAIN
```

OPEN (result set cursor)

Executes the SQL statement with result set associated to the specified database cursor

Syntax

```
OPEN cid
   [ USING pvar {IN|OUT|INOUT} [,...] ]
   [ WITH REOPTIMIZATION ]
```

1. *cid* is the identifier of the database cursor.
2. *pvar* is a variable containing an input value for an SQL parameter.

Usage:

The `OPEN` instruction executes the SQL statement of a declared cursor. The result set is produced on the server side and rows can be fetched.

The `USING` clause is required to provide the SQL parameters as program variables, if the cursor was declared with a prepared statement that includes (?) question mark placeholders.

A subsequent `OPEN` statement closes the cursor and then reopens it. When the database server reopens the cursor, it creates a new result set, based on the current values of the variables in the `USING` clause. If the variables have changed since the previous `OPEN` statement, reopening the cursor can generate an entirely different result set.

The `IN`, `OUT` or `INOUT` options can be used to call stored procedures having input / output parameters and generating a result set. Use the `IN`, `OUT` or `INOUT` options to indicate if a parameter is respectively for input, output or both.

Sometimes, query execution plans need to be re-optimized when SQL parameter values change. Use the `WITH REOPTIMIZATION` clause to indicate that the query execution plan has to be re-optimized on the database server (this operation is normally done during the `DECLARE` instruction). If this option is not supported by the database server, it is ignored.

In a IBM® Informix® database that is ANSI-compliant, you receive an error code if you try to open a cursor that is already open. **Informix® only!**

A cursor is closed with the `CLOSE` instruction, or when the parent connection is terminated (typically, when the program ends). By using the `CLOSE` instruction explicitly, you release resources allocated for the result set in the db client library and on the database server.

The database server evaluates the values that are named in the `USING` clause of the `OPEN` statement only when it opens the cursor. While the cursor is open, subsequent changes to program variables in the `OPEN` clause do not change the result set of the cursor; you must re-open the cursor to re-execute the statement.

If you release cursor resources with a `FREE` instruction, you cannot use the cursor unless you declare the cursor again.

The `IN`, `OUT` or `INOUT` options can only be used for simple variables, you cannot specify those options for a complete record with the record.* notation.

Example

```
MAIN
   DEFINE k INTEGER
   DEFINE n VARCHAR(50)
   DATABASE stores
   DECLARE c1 CURSOR FROM "SELECT cust_name FROM customer WHERE
 cust_id > ?"
   LET k = 102
   OPEN c1 USING k
   FETCH c1 INTO n
   LET k = 103
   OPEN c1 USING k
   FETCH c1 INTO n
END MAIN
```

FETCH (result set cursor)

Moves a cursor to a new row in the corresponding result set and retrieves the row values into fetch buffers.

Syntax

```
FETCH [ direction ] cid
   [ INTO fvar [,...] ]
```

where *direction* is one of:

```
{
  NEXT
| { PREVIOUS | PRIOR }
| CURRENT
| FIRST
| LAST
| ABSOLUTE position
| RELATIVE offset
}
```

1. *cid* is the identifier of the database cursor.
2. *fvar* is a variable used as fetch buffer.
3. *direction* options different from `NEXT` can only be used with scrollable cursors.
4. *position* is an positive integer expression.
5. *offset* is a positive or negative integer expression.

Usage

The `FETCH` instruction retrieves a row from a result set of an opened cursor. The cursor must be opened before using the `FETCH` instruction.

The `INTO` clause can be used to provide the fetch buffers that receive the result set column values.

A sequential cursor can fetch only the next row in sequence from the result set.

The `NEXT` clause (the default) retrieves the next row in the result set. If the row pointer was on the last row before executing the instruction, the SQL code is set to 100 (`NOTFOUND`), and the row pointer remains on the last row. (if you issue a `FETCH PREVIOUS` at this time, you get the next-to-last row).

The `PREVIOUS` clause retrieves the previous row in the result set. If the row pointer was on the first row before executing the instruction, the SQL code is set to 100 (`NOTFOUND`), and the row pointer remains on the first row. (if you issue a `FETCH NEXT` at this time, you get the second row).

The CURRENT clause retrieves the current row in the result set.

The FIRST clause retrieves the first row in the result set.

The LAST clause retrieves the last row in the result set.

The ABSOLUTE clause retrieves the row at *position* in the result set. If the *position* is not correct, the SQL code is set to 100 (NOTFOUND). Absolute row positions are numbered from 1.

The RELATIVE clause moves *offset* rows in the result set and returns the row at the current position. The offset can be a negative value. If the *offset* is not correct, the SQL code is set to 100 (NOTFOUND). If *offset* is zero, the current row is fetched.

Fetching rows can have specific behavior when the cursor was declared FOR UPDATE to perform a positioned update or delete.

Example

```
MAIN
   DEFINE cust_rec RECORD
            cnum INTEGER,
            cname CHAR(20)
        END RECORD
   DATABASE stores
   DECLARE c1 SCROLL CURSOR FOR SELECT customer_num, cust_name
 FROM customer
   OPEN c1
   FETCH c1 INTO cust_rec.*
   FETCH LAST c1 INTO cust_rec.*
   FETCH PREVIOUS c1 INTO  cust_rec.*
   FETCH FIRST c1 INTO cust_rec.*
   FETCH LAST c1 -- INTO clause is optional
   FETCH FIRST c1 -- INTO clause is optional
END MAIN
```

CLOSE (result set cursor)

Closes a database cursor and frees resources allocated on the database server for the result set.

Syntax

```
CLOSE cid
```

1. *cid* is the identifier of the database cursor.

Usage

The CLOSE instruction releases the resources allocated for the result set on the database server.

After using the CLOSE instruction, you must reopen the cursor with OPEN before retrieving values with FETCH.

You should close the cursor when the result set is no longer used, this saves resources on the database client and database server side.

Example

```
MAIN
   DATABASE stores
   DECLARE c1 CURSOR FOR SELECT * FROM customer
   OPEN c1
```

```
   CLOSE c1
   OPEN c1
   CLOSE c1
END MAIN
```

FREE (result set cursor)

Releases SQL cursor resources allocated by the DECLARE instruction.

Syntax

```
FREE cid
```

1. *cid* is the identifier of the database cursor.

Usage

The FREE instruction takes the name of a cursor as parameter.

All resources allocated to the database cursor are released.

If not done, the cursor is automatically closed when doing a FREE.

When cursor resources are released with FREE, the cursor must be declared again before usage.

Free the cursor when the result set is no longer used by the program; this saves resources on the database client and database server side.

Example

```
MAIN
   DEFINE i, j INTEGER
   DATABASE stores
   FOR i=1 TO 10
       DECLARE c1 CURSOR FOR SELECT * FROM customer
       FOR j=1 TO 10
           OPEN c1
           FETCH c1
           CLOSE c1
       END FOR
       FREE c1
   END FOR
END MAIN
```

FOREACH (result set cursor)

Processes a series data rows returned from a database cursor.

Syntax

```
FOREACH cid
   [ USING pvar {IN|OUT|INOUT} [,...] ]
   [ INTO fvar [,...] ]
   [ WITH REOPTIMIZATION ]
       {
         statement
       | CONTINUE FOREACH
       | EXIT FOREACH
       }
```

```
        [...]
END FOREACH
```

1. *cid* is the identifier of the database cursor.
2. *pvar* is a variable containing an input value for an SQL parameter.
3. *fvar* is a variable used as fetch buffer.

Usage

Use the FOREACH instruction to retrieve and process database rows that were selected by a query. This instruction is equivalent to using the OPEN, FETCH and CLOSE cursor instructions:

1. Open the specified cursor
2. Fetch the rows selected
3. Close the cursor (after the last row has been fetched)

You must declare the cursor (by using the DECLARE instruction) before the FOREACH instruction can retrieve the rows. A compile-time error occurs unless the cursor was declared prior to this point in the source module. You can reference a sequential cursor, a scroll cursor, a hold cursor, or an update cursor, but FOREACH only processes rows in sequential order.

The FOREACH statement performs successive fetches until all rows specified by the SELECT statement are retrieved. Then the cursor is automatically closed. It is also closed if a WHENEVER NOT FOUND exception handler within the FOREACH loop detects a NOTFOUND condition.

After a FOREACH loop, STATUS and SQLCA.SQLCODE will not be set to NOTFOUND(100) if no rows are returned by the query: If no error occurred, these registers will hold the value zero.

The USING clause is required to provide the SQL parameter buffers, if the cursor was declared with a prepared statement that includes (?) question mark placeholders.

The IN, OUT or INOUT options can be used to call stored procedures having input / output parameters and generating a result set. Use the IN, OUT, or INOUT options to indicate if a parameter is respectively for input, output, or both.

The INTO clause can be used to provide the fetch buffers that receive the row values.

Use the WITH REOPTIMIZATION clause to indicate that the query execution plan has to be re-optimized.

The CONTINUE FOREACH instruction interrupts processing of the current row and starts processing the next row. The runtime system fetches the next row and resumes processing at the first statement in the block.

The EXIT FOREACH instruction interrupts processing and ignores the remaining rows of the result set.

The IN, OUT, or INOUT options can only be used for simple variables; you cannot specify those options for a complete record with the record.* notation.

Example

```
MAIN
   DEFINE clist ARRAY[200] OF RECORD
            cnum INTEGER,
            cname CHAR(50)
          END RECORD
   DEFINE i INTEGER
   DATABASE stores
   DECLARE c1 CURSOR FOR SELECT customer_num, cust_name FROM
 customer
   LET i=0
   FOREACHc1 INTO clist[i+1].*
      LET i=i+1
      DISPLAY clist[i].*
```

```
   END FOREACH
   DISPLAY "Number of rows found: ", i
END MAIN
```

Positioned updates/deletes

Describes row modification based on a FOR UPDATE cursor.

- Understanding positioned update or delete on page 514
- DECLARE (SELECT ... FOR UPDATE) on page 515
- UPDATE ... WHERE CURRENT OF on page 516
- DELETE ... WHERE CURRENT OF on page 517
- Examples on page 517

Understanding positioned update or delete

When declaring a database cursor with a `SELECT` statement using a unique table and ending with the `FOR UPDATE` keywords, you can update or delete database rows by using the `WHERE CURRENT OF` keywords in the `UPDATE` or `DELETE` statements. Such an operation is called *positioned update* or *positioned delete*.

Some database servers do not support **hold** cursors (`WITH HOLD`) declared with a `SELECT` statement including the `FOR UPDATE` keywords. The SQL standards require *for update cursors* to be automatically closed at the end of a transaction. Therefore, it is strongly recommended that you use positioned updates in a transaction block.

Do not confuse positioned update with the use of `SELECT FOR UPDATE` statements that are not associated with a database cursor. Executing `SELECT FOR UPDATE` statements is supported by the language, but you cannot perform positioned updates since there is no cursor identifier associated to the result set.

To perform a positioned update or delete, you must declare the database cursor with a `SELECT FOR UPDATE` statement.

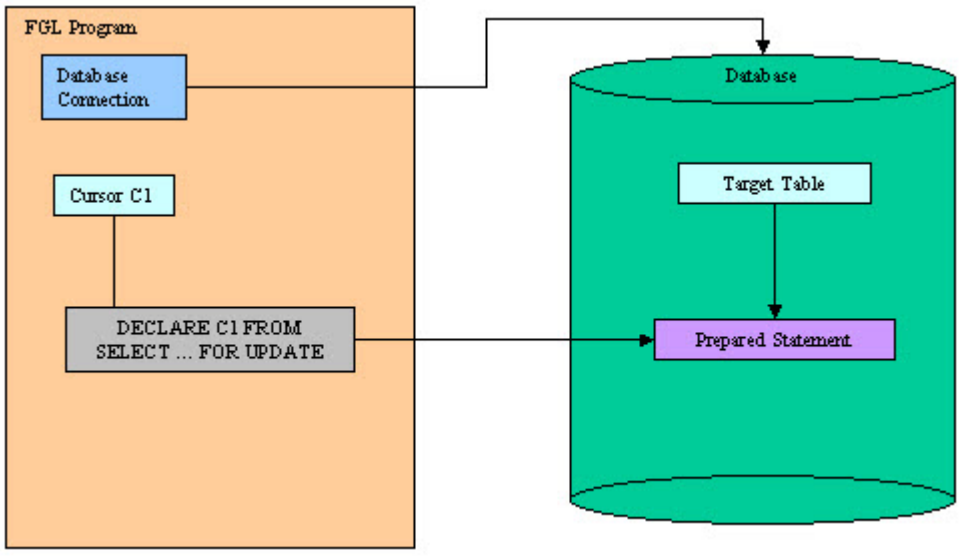


Figure 28: SELECT FOR UPDATE statement

Then, start a transaction, open the cursor and fetch a row.

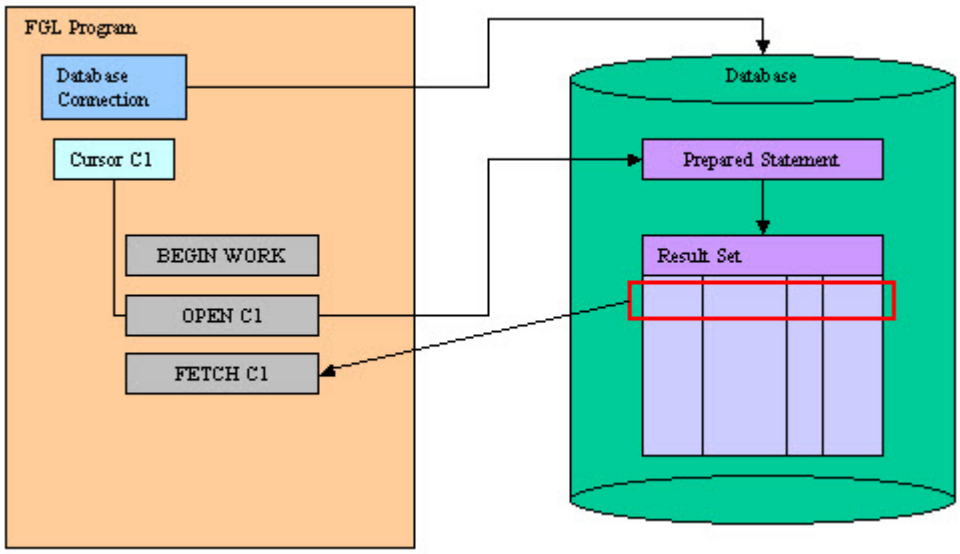


Figure 29: Open a cursor

Finally, you update or delete the current row and you commit the transaction.

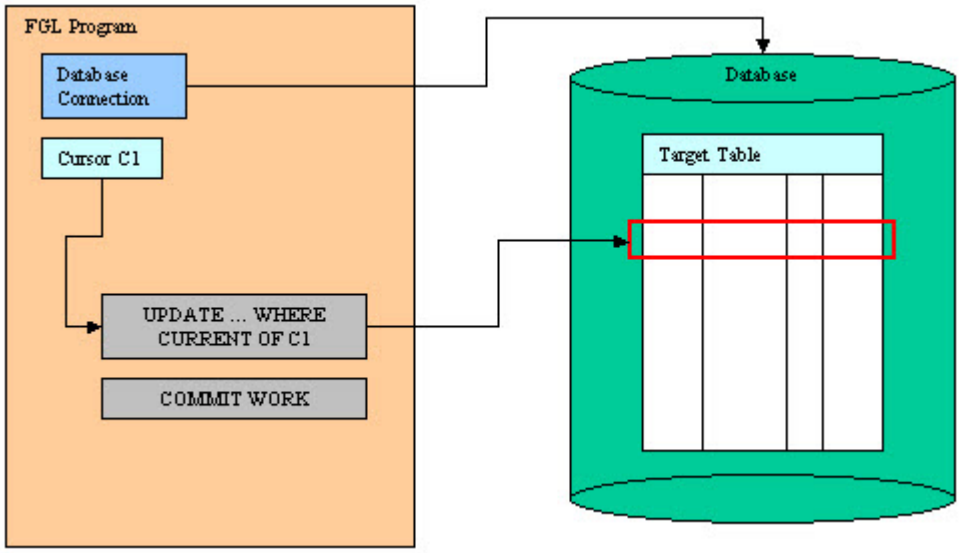


Figure 30: Delete the row

DECLARE (SELECT ... FOR UPDATE)

Associate a database cursor with a SELECT statement to perform positioned updates and deletes

Syntax

```
DECLARE cid [SCROLL] CURSOR [WITH HOLD]
    FOR { select-statement | sid }
```

1. *cid* is the identifier of the database cursor.
2. *select-statement* is a SELECT statement defined in static SQL, with the FOR UPDATE keywords.
3. *sid* is the identifier of a prepared SELECT statement including the FOR UPDATE keywords.

Usage

DECLARE ... FOR UPDATE will define a cursor that can be used to do positioned updates and deletes with the WHERE CURRENT OF clause.

DECLARE must precede any other statement that refers to the cursor during program execution.

To perform positioned updates, the *select-statement* must include the FOR UPDATE keywords.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared. Therefore, you must execute the DECLARE, UPDATE or DELETE instructions in the same module.

The static *select-statement* used in the DECLARE can contain ? (question mark) parameter placeholders, that can be bound to program variables with the USING clause of the OPEN instruction.

Use the WITH HOLD option carefully, because this feature is specific to IBM® Informix® servers. Other database servers do not behave as Informix® does with such cursors. For example, if the SELECT is not declared FOR UPDATE, most database servers keep cursors open after the end of a transaction, but IBM® DB2® automatically closes all cursors when the transaction is rolled back.

UPDATE ... WHERE CURRENT OF

Updates the current row in a result set of a database cursor declared for update.

Syntax

```
UPDATE table-specification
   SET
       column = { variable | sql-expression }
       [,...]
   WHERE CURRENT OF cid
```

1. *table-specification* identifies the target table (see UPDATE for more details).
2. *column* is a name of a table column.
3. *variable* is a program variable, a record member or an array member used as a parameter buffer to provide values.
4. *sql-expression* is an expression supported by the database server, this can be a literal or NULL for example.
5. *cid* is the identifier of the database cursor declared for update.

Usage

Use UPDATE ... WHERE CURRENT OF to modify the values of the row currently pointed by the associated FOR UPDATE cursor.

The UPDATE statement does not advance the cursor to the next row, so the current row position remains unchanged.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared. Therefore, you must execute the DECLARE,UPDATE or DELETE instructions in the same module.

There must be a current row in the result set. Make sure that the SQL status returned by the last FETCH is equal to zero.

If the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you are restricted to updating only those columns in a subsequent UPDATE ... WHERE CURRENT OF statement.

DELETE ... WHERE CURRENT OF

Deletes the current row in a result set of a database cursor declared for update.

Syntax

```
DELETE FROM table-specification
   WHERE CURRENT OF cid
```

1. *table-specification* identifies the target table
2. *cid* is the identifier of the database cursor declared for update.

Usage

Use `DELETE ... WHERE CURRENT OF` to remove the row currently pointed by the associated `FOR UPDATE` cursor.

After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you reposition the cursor with a FETCH statement.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared. Therefore, you must execute the `DECLARE`, `UPDATE` or `DELETE` instructions in the same module.

There must be a current row in the result set. Make sure that the SQL status returned by the last `FETCH` is equal to zero.

Examples

Example 1: Positioned UPDATE statement

```
MAIN
  DEFINE pname CHAR(30)
  DATABASE stock
  DECLARE uc CURSOR FOR
    SELECT name FROM item WHERE key=123 FOR UPDATE
  BEGIN WORK
    OPEN uc
    FETCH uc INTO pname
    IF sqlca.sqlcode=0 THEN
      LET pname = "Dummy"
      UPDATE item SET name=pname WHERE CURRENT OF uc
    END IF
    CLOSE uc
  COMMIT WORK
  FREE uc
END MAIN
```

SQL insert cursors

Explains how to insert a log of rows into a table efficiently.

- Understanding SQL insert cursors on page 518
- DECLARE (insert cursor) on page 520
- OPEN (insert cursor) on page 521
- PUT (insert cursor) on page 521
- FLUSH (insert cursor) on page 521
- CLOSE (insert cursor) on page 522
- FREE (insert cursor) on page 522

- Examples on page 522

Understanding SQL insert cursors

An *insert cursor* is a database cursor declared with a restricted form of the `INSERT` statement, designed to perform buffered row insertion in database tables.

The insert cursor simply inserts rows of data; it cannot be used to fetch data. When an insert cursor is opened, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program executes `PUT` statements. The rows are written to disk only when the buffer is full. You can use the `CLOSE` , `FLUSH` , or `COMMIT WORK` statement to flush the buffer when it is less than full. You must close an insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly.

When the database server supports buffered inserts, an insert cursor increases processing efficiency (compared with embedding the `INSERT` statement directly). This process reduces communication between the program and the database server and also increases the speed of the insertions.

Before using the insert cursor, you must declare it with the `DECLARE` instruction using an `INSERT` statement.

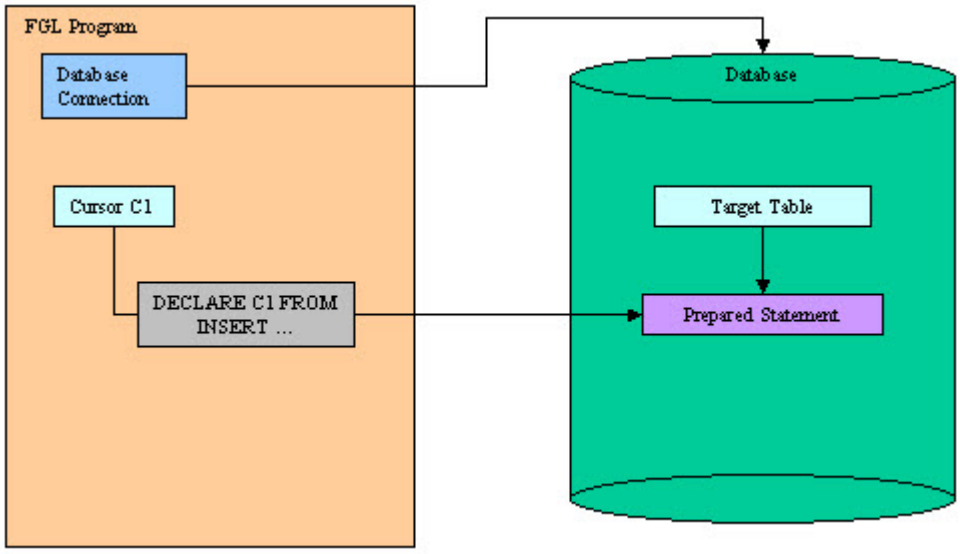


Figure 31: Declaring a cursor

Once declared, you can open the insert cursor with the `OPEN` instruction. This instruction prepares the insert buffer. When the insert cursor is opened, you can add rows to the insert buffer with the `PUT` statement.

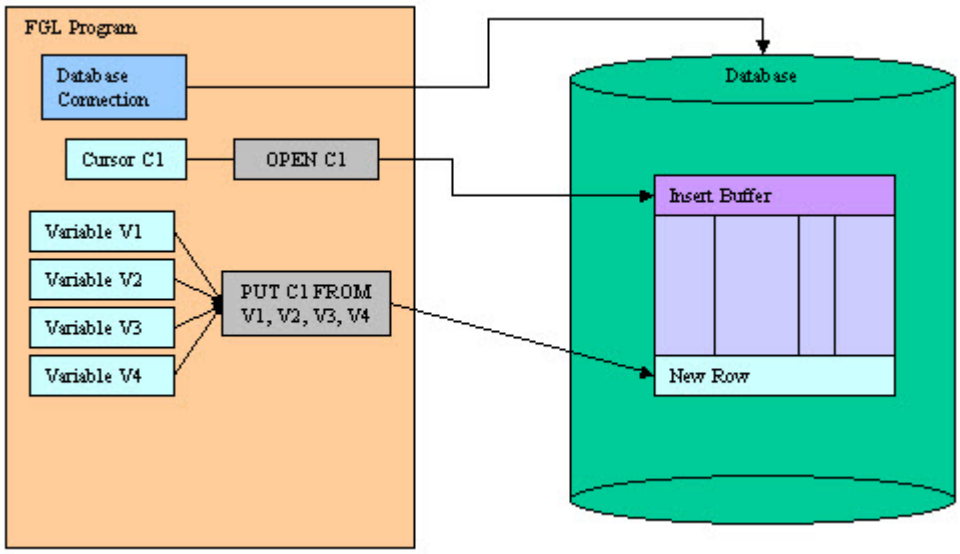


Figure 32: OPEN and PUT statements

Rows are automatically added to the database table when the insert buffer is full. To force row insertion in the table, you can use the FLUSH instruction.

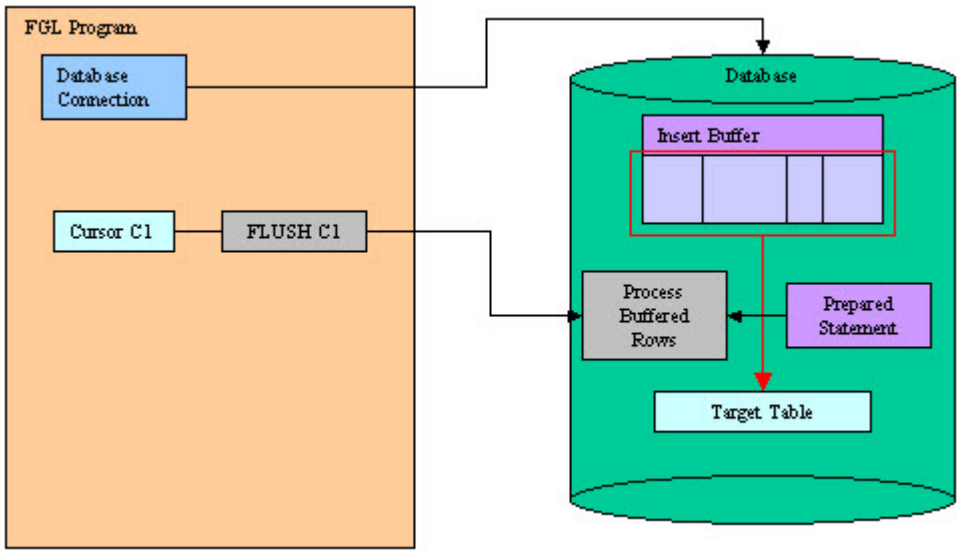


Figure 33: FLUSH statement

Finally, when all rows are added, you can CLOSE the cursor and if you no longer need it, you can de-allocate resources with the FREE instruction.

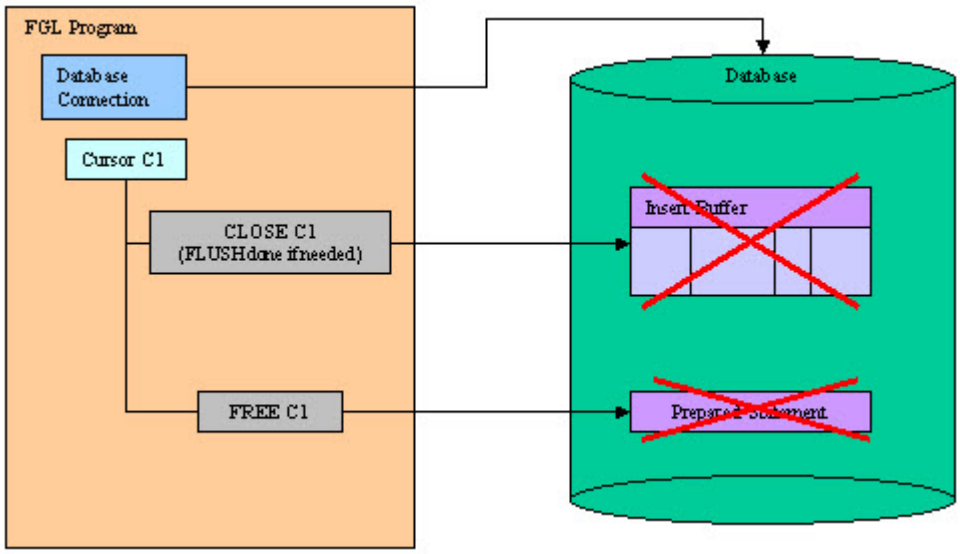


Figure 34: CLOSE and FREE statements

By default, insert cursors must be opened inside a transaction block, with `BEGIN WORK` and `COMMIT WORK`, and they are automatically closed at the end of the transaction. If needed, you can declare insert cursors with the `WITH HOLD` clause, to allow uninterrupted row insertion across multiple transactions.

DECLARE (insert cursor)

The `DECLARE` with an `INSERT` instruction defines an insert cursor.

Syntax

```
DECLARE cid CURSOR [WITH HOLD] FOR { insert-statement | sid }
```

1. *cid* is the identifier of the insert cursor.
2. *insert-statement* is an `INSERT` statement defined in static SQL.
3. *sid* is the identifier of a prepared `INSERT` statement.

Usage

Use the `DECLARE` instruction with an `INSERT` instruction to define a new insert cursor in the current database session.

The `INSERT` statement is parsed, validated and the execution plan is created.

`DECLARE` must precede any other statement that refers to the cursor during program execution.

The scope of reference of the *cid* cursor identifier is local to the module where it is declared.

The static *insert-statement* statement can include a list of variables in the `VALUES` clause. These variables are automatically read by the `PUT` statement; you do not have to provide the list of variables in that statement. As an alternative, use the ? (question mark) SQL parameter placeholder in the `VALUE` clause to bind program variables provided in the `FROM` clause of the `PUT` instruction.

When declaring a cursor with a prepared *sid* statement, the statement can include ? (question mark) placeholders for SQL parameters. In this case you must provide a list of variables in the `FROM` clause of the `PUT` statement.

Use the `WITH HOLD` option to declare cursors that have uninterrupted inserts across multiple transactions.

Resources allocated by the `DECLARE` can be released later by the `FREE` instruction.

The number of declared cursors in a single program is limited by the database server and the available memory. Make sure that you free the resources when you no longer need the declared insert cursor.

The identifier of a cursor that was declared in one module cannot be referenced from another module.

OPEN (insert cursor)

Initializes an insert cursor.

Syntax

```
OPEN cid
```

1. *cid* is the identifier of the insert cursor.

Usage

The `OPEN` statement initializes the insert cursor if the specified cursor was declared with an `INSERT` statement.

Once the insert cursor is opened, you can add rows with the `PUT` statement.

When used with an insert cursor, the `OPEN` instruction cannot include a `USING` clause.

A subsequent `OPEN` statement closes the cursor and then reopens it.

If the insert cursor was not declared `WITH HOLD` option, the`OPEN` instruction generates an SQL error if there is no current transaction started.

If you release cursor resources with a `FREE` instruction, you cannot use the cursor unless you declare the cursor again.

PUT (insert cursor)

Adds a new row to the insert cursor buffer.

Syntax

```
PUT cid FROM pvar [,...]
```

1. *cid* is the identifier of the insert cursor.
2. *pvar* is a variable containing an input value for the new row.

Usage

The `PUT` instruction adds a row to the insert cursor buffer.

If the insert cursor was not declared `WITH HOLD` option, the `PUT` instruction generates an SQL error if there is no current transaction started.

If the insert buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block, and the buffer is emptied. As a result, some `PUT` statement executions cause rows to be written to the database, and some do not.

FLUSH (insert cursor)

Flushes the buffer of an insert cursor.

Syntax

```
FLUSH cid
```

1. *cid* is the identifier of the insert cursor.

Usage

When flushing an insert cursor, all buffered rows are inserted into the target database table and the insert buffer is cleared.

The insert buffer may be automatically flushed by the runtime system if there no room when a new row is added with the PUT instruction.

CLOSE (insert cursor)

Flushes and closes an insert cursor.

Syntax

```
CLOSE cid
```

1. *cid* is the identifier of the insert cursor.

Usage

If rows are present in the insert buffer, they are inserted into the target table.

Closing the insert cursor releases the resources allocated for the insert buffer on the database server.

After using the CLOSE instruction, you must reopen the cursor with OPEN before adding new rows with PUT/FLUSH.

FREE (insert cursor)

Releases resources allocated for an insert cursor.

Syntax

```
FREE cid
```

1. *cid* is the identifier of the insert cursor.

Usage

After executing the FREE statement, all resources allocated to the insert cursor are released.

The cursor should be explicitly closed before it is freed.

If you release cursor resources with this instruction, you cannot use the cursor unless you declare the cursor again.

Examples

Example 1: Insert Cursor declared with a Static INSERT

```
MAIN
   DEFINE i INTEGER
   DEFINE rec RECORD
            key INTEGER,
            name CHAR(30)
          END RECORD
   DATABASE stock
   DECLARE ic CURSOR FOR
     INSERT INTO item VALUES (rec.*)
```

```
   BEGIN WORK
     OPEN ic
     FOR i=1 TO 100
         LET rec.key = i
         LET rec.name = "Item #" || i
         PUT ic
         IF i MOD 50 = 0 THEN
             FLUSH ic
         END IF
     END FOR
     CLOSE ic
   COMMIT WORK
   FREE ic
END MAIN
```

Example 2: Insert Cursor declared with an SQL text

```
MAIN
   DEFINE i INTEGER
   DEFINE rec RECORD
           key INTEGER,
           name CHAR(30)
         END RECORD
   DATABASE stock
   DECLARE ic CURSOR FROM "INSERT INTO item VALUES (?,?)"
   BEGIN WORK
     OPEN ic
     FOR i=1 TO 100
         LET rec.key = i
         LET rec.name = "Item #" || i
         PUT ic FROM rec.*
         IF i MOD 50 = 0 THEN
             FLUSH ic
         END IF
     END FOR
     CLOSE ic
   COMMIT WORK
   FREE ic
END MAIN
```

Example 3: Insert Cursor declared with 'hold' option

```
MAIN
   DEFINE name CHAR(30)
   DATABASE stock
   DECLARE ic CURSOR WITH HOLD FOR
     INSERT INTO item VALUES (1,name)
   OPEN ic
   LET name = "Item 1"
   PUT ic
   BEGIN WORK
     UPDATE refs SET name="xyz" WHERE key=123
   COMMIT WORK
   PUT ic
   PUT ic
   FLUSH ic
   CLOSE ic
   FREE ic
END MAIN
```

SQL load and unload

Describes the instructions to export/import information from/to a database.

- LOAD on page 524
- UNLOAD on page 527

LOAD

Inserts data from a file into an existing database table.

Syntax

```
LOAD FROM filename [ DELIMITER delimiter]
{
  INSERT INTO table-specification [ ( column [,...]) ]
|
  insert-string
}
```

where *table-specification* is:

```
[dbname[@dbserver]:][owner.]table
```

1. *filename* is a string expression containing the name of the file the data is read from.
2. *delimiter* is the character used as the value delimiter.
3. The `INSERT` clause is a pseudo `INSERT` statement (without the `VALUES` clause), where you can specify the list of columns in braces.
4. *dbname* identifies the database name.
5. *dbserver* identifies the database server (INFORMIXSERVER).
6. *owner* identifies the owner of the table, with optional double quotes.
7. *table* is the name of the database table.
8. *column* is a name of a table column.
9. *insert-string* is a string expression containing the pseudo-INSERT statement.

Usage

The `LOAD` instruction reads serialized data from an input file and inserts new rows in a database table specified in the `INSERT` clause. A file created by the `UNLOAD` statement can be used as input for the `LOAD` statement if its values are compatible with the schema of *table*.

The `LOAD` statement must include a pseudo-`INSERT` statement (either directly or as text in a variable) to specify where to store the data. `LOAD` appends the new rows to the specified table, synonym, or view, but does not overwrite existing data. It cannot add a row that has the same key as an existing row.

The *dbname*, *dbserver* and *owner* prefix of the table name should be avoided for maximum SQL portability.

The number and the order of columns in the `INSERT` statement must match the values of the input file.

The `LOAD` instruction cannot be prepared with a `PREPARE` statement, however `LOAD` can take a string literal as parameter, that allows to build the `INSERT` statement at runtime.

The variable or string following the `LOAD FROM` keywords must specify the name of a file of ASCII characters (or characters that are valid for the current locale) that holds the data values that are to be inserted.

Each set of data values in *filename* that represents a new row is called an input record. Each input record must contain the same number of delimited data values. If the `INSERT` clause has no list of columns, the

sequence of values in each input record must match the columns of *table* in number and order. Each value must have the literal format of the column data type, or of a compatible data type.

If LOAD is executed within a transaction block (BEGIN WORK / COMMIT WORK), the rows inserted by the LOAD instruction are part of the transaction. With some database servers the insert rows remain locked until the COMMIT WORK or ROLLBACK WORK statement terminates the transaction. Consider locking the whole table to

If the database does not support transactions, a failing LOAD statement cannot remove any rows that were loaded before the failure occurred. You must manually remove the already loaded records from either the load file or from the receiving table, repair the erroneous records, and rerun LOAD.

If the database supports transactions, you can do the following actions:

- Run LOAD as a singleton transaction, so that any error causes the entire LOAD statement to be automatically rolled back.
- Run LOAD within an explicit BEGIN WORK / COMMIT WORK transaction block, so that a data error merely stops the LOAD statement in place with the transaction still open.

A single character delimiter instructs LOAD to read data in the default format. When using "CSV" as delimiter specification, the LOAD instruction will read the data in CSV format. If the DELIMITER clause is not specified, the delimiter is defined by the DBDELIMITER environment variable. If the DBDELIMITER environment variable is not set, the default is a | pipe. The field delimiter can be a blank character. It cannot be backslash or any hexadecimal digit (0-9, A-F, a-f). If the delimiter specified in the LOAD command is NULL, the runtime system will use the default delimiter or DBDELIMITER if the variable is defined.

At this time, data type description of the input file fields is implicit; in order to create the SQL parameter buffers to hold the field values for inserts, the LOAD instruction uses the current database connection to get the column data types of the target table. Those data types depend on the type of database server. For example, IBM® Informix® DATE columns do not store the same data as the Oracle DATE data type. Therefore, be careful when using the LOAD/UNLOAD instructions; if the application connects to different kinds of database servers, it can result data conversion errors.

Pay attention to numeric (DECIMAL, MONEY) and date/time values (DATE, DATETIME): These must match the current format settings (DBMONEY, DBDATE). As a general programming pattern, use simple INSERT statements to load default and configuration data into your database, in order to be independent from the numeric and date format settings.

Default LOAD format

The next table describes how data values should be represented in the input file used by the LOAD instruction. Values must be serialized with a character string following the SQL data type of the receiving column of the table.

Table 158: Data representation for the default LOAD format

Data type	Input Format
CHAR, VARCHAR, TEXT	Values can have more characters than the declared maximum length of the column, but any extra characters are ignored. A backslash (\ ) is required before any literal backslash or any literal delimiter character, and before any NEWLINE character anywhere in character value. Blank values can be represented as one or more blank characters between delimiters, but leading blanks must not precede other CHAR, VARCHAR, or TEXT values.
DATE	In the default locale, values must be in month/day/year format unless another format is specified by DBDATE environment variable. The day and month must be a 2-digit number, and the year must be a 4-digit number.
DATETIME	DATETIME values must be in the format:

Data type	Input Format
	`year-month-day hour:minute:second.fraction` or a contiguous subset, without the `DATETIME` keyword or qualifiers. Time units outside the declared column precision can be omitted. The `year` must be a four-digit number; all other time units (except `fraction`) require two digits.

`INTERVAL`	`INTERVAL` values must be formatted:
	`year-month`
	or
	`day hour:minute:second.fraction`
	or a contiguous subset thereof, without the `INTERVAL` keyword or qualifiers. Time units outside the declared column precision can be omitted. All time units (except `year` and `fraction`) require two digits.

`DECIMAL`, `MONEY`	Values must use the decimal separator defined by DBFORMAT/DBMONEY. For `MONEY`, values can include currency symbols, but these are not required.
`BYTE`	Values must be ASCII-hexadecimals; no leading or trailing blanks.
`SERIAL`, `BIGSERIAL`, `SERIAL8`	Values can be represented as 0 to tell the database server to supply a new serial value. You can specify a literal integer greater than zero, but if the column has a unique index, an error results if this number duplicates an existing value.

The NEWLINE character must terminate each input record in *filename*. Specify only values that the language can convert to the data type of the database column. For database columns of character data types, inserted values are truncated from the right if they exceed the declared length of the column.

NULL values of any data type must be represented by consecutive delimiters in the input file; you cannot include anything between the delimiter symbols.

The `LOAD` statement expects incoming data in the format specified by environment variables like `DBFORMAT`, `DBMONEY`, `DBDATE`, `GL_DATE`, and `GL_DATETIME`. The precedence of these format specifications is consistent with forms and reports. If there is an inconsistency, an error is reported and the `LOAD` is canceled.

The backslash symbol (`\`) serves as an escape character in the input file to indicate that the next character in a data value is a literal. The `LOAD` statement scans for backslash escaped elements to read special characters in the following contexts:

- The backslash character appears anywhere in the value.
- The delimiter character appears anywhere in the value.
- The NEWLINE character appears anywhere in a value.

CSV LOAD format

The CSV (comma separated values) format is similar to the default format when using a simple comma delimiter, with the following differences:

- Input values might be surrounded with " double quotes.
- If an input value contains a comma or a NEWLINE, it is not escaped be the value must be quoted in the file.
- Double-quote characters in input values are doubled and will be converted to a unique " character; the value must be quoted.
- Backslash characters are not escaped in the input file and are read as; the value must be quoted.
- Leading and trailing blanks are kept (no truncation).
- No ending delimiter is expected at the end of the input record.

Example

```
MAIN
  DATABASE stores
  BEGIN WORK
  DELETE FROM items
  LOAD FROM "items01.unl" INSERT INTO items
  LOAD FROM "items02.unl" INSERT INTO items
  COMMIT WORK
END MAIN
```

UNLOAD

Copies data from the database tables into a file.

Syntax

```
UNLOAD TO filename [ DELIMITER delimiter]
{
  select-statement
|
  select-string
}
```

1. *filename* is a string expression containing the name of the file the data is written to.
2. *delimiter* is the character used as the value delimiter.
3. *select-statement* is static SELECT statement.
4. *select-string* is string expression containing the SELECT statement.

Usage

The UNLOAD instruction serializes into a file the SQL data produced by a SELECT statement.

You cannot use the PREPARE statement to pre-process an UNLOAD statement, you can however use a string literal to build the SELECT statement at runtime.

The *filename* after the TO keyword identifies an output file in which to store the rows retrieved from the database by the SELECT statement. In the default (U.S. English) locale, this file contains only ASCII characters. (In other locales, output from UNLOAD can contain characters from the codeset of the locale.)

The UNLOAD statement must include a SELECT statement (directly, or in a variable) to specify what rows to copy into *filename*. UNLOAD does not delete the copied data.

A single character delimiter instruct UNLOAD to write data in the default format. When using "CSV" as delimiter specification, the UNLOAD instruction will write the data in CSV format. If the DELIMITER clause is not specified, the delimiter is defined by the DBDELIMITER environment variable. If the DBDELIMITER environment variable is not set, the default is a | pipe. The field delimiter can be a blank character. It cannot be backslash or any hexadecimal digit (0-9, A-F, a-f). If the delimiter specified in the UNLOAD command is NULL, the runtime system will use the default delimiter or DBDELIMITER if the variable is defined.

When using a *select-string* , do not attempt to substitute question marks (?) in place of host variables to make the SELECT statement dynamic, because this usage has binding problems.

At this time, data type description of the output file fields is implicit; in order to create the fetch buffers to hold the column values, the UNLOAD instruction uses the current database connection to get the column data types of the generated result set. Those data types depend on the type of database server. For example, IBM® Informix® INTEGER columns are 4-bytes integers, while the Oracle INTEGER data type

is actually a `NUMBER(10,0)` type. Therefore, you should take care when using this instruction; if your application connects to different kinds of database servers, you may get data conversion errors.

Default UNLOAD format

A set of values in output representing a row from the database is called an *output record*. A NEWLINE character (ASCII 10) terminates each output record.

The `UNLOAD` statement represents each value in the output file as a character string by using the current locale, according to the data type of the database column:

Table 159: Default UNLOAD format

Data type	Output Format
`CHAR, VARCHAR, TEXT`	Trailing blanks are dropped from `CHAR` and `TEXT` (but not from `VARCHAR`) values. A backslash (`\`) is inserted before any literal backslash or delimiter character and before a NEWLINE character in a character value.
`DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, SMALLINT`	Values are written as literals with no leading blanks. `MONEY` values are represented with no leading currency symbol. Zero values are represented as 0 for `INTEGER` or `SMALLINT` columns, and as 0.00 for `FLOAT`, `SMALLFLOAT`, `DECIMAL`, and `MONEY` columns.
`DATE`	Values are written in the format `month/day/year` unless some other format is specified by the `DBDATE` environment variable.
`DATETIME`	`DATETIME` values are formatted `year-month-day hour:minute:second.fraction` or a contiguous subset, without `DATETIME` keyword or qualifiers. Time units outside the declared precision of the database column are omitted.

`INTERVAL`	INTERVAL values are formatted
	`year-month`
	or
	`day hour:minute:second.fraction`
	or a contiguous subset, without `INTERVAL` keyword or qualifiers. Time units outside the declared precision of the database column are omitted.

`BYTE`	`BYTE` Values are written in ASCII hexadecimal form, without any added blank or NEWLINE characters. The logical record length of an output file that contains `BYTE` values can be very long, and thus might be very difficult to print or to edit.

NULL values of any data type are represented by consecutive delimiters in the output file, without any characters between the delimiter symbols.

The backslash symbol (`\`) serves as an escape character in the output file to indicate that the next character in a data value is a literal. The `UNLOAD`statement automatically inserts a preceding backslash to prevent literal characters from being interpreted as special characters in the following contexts:

- The backslash character appears anywhere in the value.
- The delimiter character appears anywhere in the value.
- The NEWLINE character appears anywhere in a value.

CVS UNLOAD format

The CSV (comma separated values) format is similar to the standard format when using a simple comma delimiter, with the following differences:

- A comma character generates a quoted output value, and the comma is written as is (not escaped).
- A " double-quote character generate quoted output value and the quote in the value is doubled.
- NEWLINE characters generate a quoted output value, and the NEWLINE is written as is (not escaped).
- Backslash characters are written as is in the output value (i.e. not escaped).
- Leading and trailing blanks are not truncated in the output value.
- No ending delimiter is written at the end of the output record.

Example

```
MAIN
  DEFINE var INTEGER
  DATABASE stores
  LET var = 123
  UNLOAD TO "items.unl"
    SELECT * FROM items WHERE item_num > var
END MAIN
```

SQL adaptation guides

This section includes the SQL adaptation guides for various supported databases. The adaptation guides provide you with information about installation and configuration requirements, as well as details on what is and is not supported when using database-specific SQL.

- SQL guide for IBM Informix database servers 5.x, 7.x, 8.x, 9.x, 10.x, 11.x on page 529
- SQL adaptation guide for IBM DB2 UDB 10.x on page 540
- SQL adaptation guide for IBM Netezza 6.x on page 572
- SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014 on page 592
- SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x on page 625
- SQL adaptation guide for Oracle Database 11, 12 on page 643
- SQL adaptation guide for PostgreSQL 9.x on page 683
- SQL adaptation guide for SQLite 3.x on page 709
- SQL adaptation guide for SAP Sybase ASE 16.x on page 723
- SQL adaptation guide for SAP HANA DB (SPS09+)

SQL guide for IBM® Informix® database servers 5.x, 7.x, 8.x, 9.x, 10.x, 11.x

Purpose of the Informix® SQL guide

This section contains information to configure your Genero runtime system to work with an Informix® database engine, and describes the IBM® Informix® SQL features that are not supported (or partially supported) by Genero BDL.

Understand that Genero BDL was designed to work with IBM® Informix® databases, so most of the IBM® Informix® SQL features are supported. However, new features implemented in recent server versions need modifications in the Genero BDL compilers and runtime system to be supported.

Some topics show an enhancement reference note with a number, identifying the request id as filed in our internal "TODO" database. If the SQL feature is mission critical for your application, contact the support center and mention the enhancement identifier.

Installation (Runtime Configuration)
ODI adaptation guide Installation topics.
Supported IBM® Informix® server and CSDK versions

1. Genero BDL is certified with all IBM® Informix® servers from version **5.x** to the latest **11.x** version, including the Standard Engine, On-Line and IDS server families, as long as the IBM® Informix® Client SDK is compatible with the server.
2. Genero BDL is certified with IBM® Informix® SDK version **3.50** or higher.

Install IBM® Informix® and create a database - database configuration/design tasks

1. Install the IBM® Informix® database software (IDS for example) on your database server.
2. Install the IBM® Informix® Software Development Kit (SDK) on your application server.

   With some IBM® Informix® distributions (IDS 11), this package is included in the server bundle. You should check the IBM® web site for SDK upgrades or patches. Genero BDL is certified with IBM® Informix® SDK version 3.50 or higher.
3. Setup the IDS server (onconfig file, etc)
   a) Starting with IDS version 11, the `TEMPTAB_NOLOG` is set to 1 by default.

      Consider setting this parameter to 0, if you want to log temporary table changes. This can affect the behavior of programs expecting that a ROLLBACK WORK cancels changes done on a temporary table.
4. Define a database user dedicated to your application: the application administrator.

   This user will manage the database schema of the application (all tables will be owned by it). With IBM® Informix®, database users reference Operating System users, and must be part of the IBM® Informix® group. See IBM® Informix® documentation.
5. Connect to the server as IBM® Informix® user (for example with the dbaccess tool) and give all requested database administrator privileges to the application administrator.

```
GRANT CONNECT TO appadmin;
GRANT RESOURCE TO appadmin;
GRANT DBA TO appadmin;
```

6. Define the database locale before creating the database.

   According to the language(s) supported in your application, consider using UTF-8 locale by setting the Informix environment variables defining the locale for the database server and data: CLIENT_LOCALE, DB_LOCALE, SERVER_LOCALE.
7. Connect as application administrator and create an IBM® Informix® database entity, for example with the following SQL statement:

   `CREATE DATABASE` *`dbname`* `WITH BUFFERED LOG;`
8. Create the application tables.

Prepare the runtime environment - connecting to the database

1. In order to connect to IBM® Informix®, you must have a database driver "dbmifx" in FGLDIR/dbdrivers.
2. Make sure the IBM® Informix® client environment variables are properly set.

   Check for example INFORMIXDIR (the path to the installation directory), INFORMIXSERVER (the name of the server defined in the sqlhosts list), etc. For more details, see the IBM® Informix® documentation.
3. In order to connect to an IBM® Informix® server, you must define a line in the `$INFORMIXDIR/etc/sqlhosts` file, referencing the server name specified in the INFORMIXSERVER environment variable.

   On Windows™ platforms, the sqlhost entries are defined in the registry database. See IBM® Informix® documentation.
4. Verify the environment variable defining the search path for IBM® Informix® SDK database client shared libraries.

Table 160: Shared library environment setting for IBM® Informix® SDK version

IBM® Informix® SDK version	Shared library environment setting
All versions	*UNIX*™: Add $INFORMIXDIR/lib, $INFORMIXDIR/lib/esql, $INFORMIXDIR/lib/tools and $INFORMIXDIR/lib/cli to LD_LIBRARY_PATH (or its equivalent). *Windows*™ : Add %INFORMIXDIR%\bin to PATH.

5. Check the database client locale settings (CLIENT_LOCALE, DB_LOCALE, etc).

   The database client locale must match the locale used by the runtime system (LC_ALL, LANG).

6. To verify if the IBM® Informix® client environment is correct, you can start the SQL command interpreter:

```
$ dbaccess - -
> CONNECT TO "dbname" USER "appadmin";
ENTER PASSWORD: password
```

7. Set up the fglprofile entries for database connections.

   Important: Make sure that you are using the ODI driver corresponding to the database client and server version.

Fully supported IBM® Informix® SQL features
Fully supported IBM® Informix® SQL features.
What are the supported IBM® Informix® SQL features?

Genero BDL was first designed for IBM® Informix® databases. The answer to this question is: Every SQL feature that is not listed in the other sections of this chapter.

The following list gives an idea of the IBM® Informix® SQL elements you can use with Genero BDL:

- Database connection control instructions (DATABASE, CONNECT). See Connections.
- Transaction control instructions and concurrency settings (BEGIN WORK, SET ISOLATION). See Transactions.
- Basic, portable data types (INT, BIGINT, DECIMAL, CHAR, VARCHAR, DATE, DATETIME, TEXT, BYTE, etc). See data types.
- Common Data Definition Language statements (CREATE TABLE, DROP TABLE, etc). See Static SQL.
- Common Data Manipulation Language statements (SELECT, INSERT, UPDATE, DELETE, etc). See Static SQL.
- Result set handling with cursors (DECLARE / OPEN / FETCH / CLOSE / FREE). See Result Sets.
- Positioned UPDATEs and DELETEs (UPDATE/DELETE WHERE CURRENT OF). See Positioned Updates.
- Cursors to insert rows (DECLARE / OPEN / PUT / FLUSH). See Insert Cursors.
- Stored procedure calls. See SQL Programming.
- SQL execution status and error messages (SQLCA, SQLSTATE). See Connections.
- Global Language Support with single and multibyte character sets for CHAR/ VARCHAR data storage. See Localization.
- LOAD and UNLOAD utility statements. See I/O SQL instructions.
- Database schema extraction to define program variables LIKE database columns. See Database Schema.

Partially supported IBM® Informix® SQL features

Partially supported IBM® Informix® SQL features.

The BIGSERIAL / SERIAL8 data types

IBM® Informix® database supports the BIGSERIAL and SERIAL8 data types for auto-generated 64 bit integer sequences.

The BIGINT data type can be used to store data from BIGSERIAL SERIAL8 values.

Note however that SQLCA.SQLERRD[2] is defined as an INTEGER and therefore cannot be used to get the last generated serial. To retrieve the last generated BIGSERIAL or SERIAL8, you must use the `dbinfo()` SQL function as in the following code example:

```
MAIN
  DEFINE new_val BIGINT
  INSERT INTO mytable VALUES ( 0, 'aaaa' )
  SELECT dbinfo('bigserial') INTO new_val
    FROM systables WHERE tabid=1
  DISPLAY new_val
END MAIN
```

The NCHAR / NVARCHAR data types

IBM® Informix® supports the standard NCHAR and NVARCHAR data types. These types are equivalent to CHAR and VARCHAR (the same character set is used), except that the collation order is locale specific with NCHAR/NVARCHAR types.

With Genero BDL, you can handle character strings of NCHAR/NVARCHAR database columns by using program variables defined with the CHAR/VARCHAR types. Since the character set is identical for NCHAR/NVARCHAR and CHAR/VARCHAR, not specific consideration needs to be given for the "N" character types.

When extracting a database schema with fgldbsch, NCHAR/NVARCHAR types will be identified in the .sch file by the native Informix® type codes 15 and 16. When compiling .4gl or .per sources referencing NCHAR/NVARCHAR columns in the schema file, the compilers will automatically use the CHAR and VARCHAR Genero BDL types for the type codes 15 and 16.

However, Genero BDL is missing full support of NCHAR and NVARCHAR types as it is not possible to declare program variables directly with the NCHAR / NVARCHAR keywords. Further, sorting features of Genero should follow the same collation order as the IBM® Informix® database when using "N" character types.

Enhancement reference: 20004

The LVARCHAR data type

IBM® Informix® supports the LVARCHAR type as a "large" VARCHAR type. The LVARCHAR type was introduced to bypass the 255 bytes size limitation of the standard VARCHAR type. Starting with IDS version 9.4, the LVARCHAR size limit is 32739 bytes. In older versions the limit was 2048 bytes.

Genero BDL does not support the LVARCHAR type natively, but it has the VARCHAR type which can hold up to 65535 bytes. IBM® Informix® LVARCHAR values can be inserted or fetched by using the BDL VARCHAR type.

Static SQL statements such as CREATE TABLE can include the LVARCHAR column type.

When extracting a schema with fgldbsch, LVARCHAR(N) columns will by default be converted to VARCHAR2(N) in the schema file. VARCHAR2 is a Genero BDL-only pseudo type identified with the type code 201, and allows define VARCHAR variables with a size that can be greater than 255 bytes.

Enhancement reference: 3464

DISTINCT data types

IBM® Informix® supports DISTINCT data types as User Defined Types based on a source data type, but with different casts and functions than those on the source data type.

Genero BDL partially supports the IBM® Informix® DISTINCT data types:

The fgldbsch schema extractor can extract columns defined with a distinct type and write the distinct type code in the .sch schema file. For more details, see the list of distinct types in the Column Definition File (.sch) on page 358

However, there are some restrictions you must be aware of:

- It is not possible to define BDL variables explicitly with the name of a distinct type. Variables must be defined indirectly with the schema by using the DEFINE LIKE statement.
- The static SQL syntax does not support OPAQUE-related syntax elements:
  - The DDL statements CREATE DISTINCT TYPE, DROP TYPE, CREATE CAST, and DROP CAST are not allowed,
  - In CREATE TABLE / ALTER TABLE DDL statements, the data type must be a built-in type.
  - The :: cast operator is not supported.

Enhancement reference: 20003

Stored Procedures

With IBM® Informix® database servers, you can write stored procedures with the SPL (Stored Procedure Language) or with an external language in C or JAVA.

If you plan to support different types of database servers, you must be aware that each DB vendor has defined its own stored procedure language. In such cases, you may consider writing most of your business logic in BDL, and implementing only some stored procedures in the database, mainly to get better performance or to use database features that only exist with stored procedures.

Genero BDL partially supports SP creation, but has full support of SP invocation:

- The Genero BDL static SQL syntax does not include CREATE FUNCTION and CREATE PROCEDURE with a body block. However, you can create stored procedures with an body block by using dynamic SQL (EXECUTE IMMEDIATE), or with CREATE PROCEDURE and the FROM *filename* clause, which is supported by Genero BDL static SQL.
- The EXECUTE FUNCTION or EXECUTE PROCEDURE instruction is not allowed in the static SQL syntax. To invoke a stored procedure with Informix®, you must use the PREPARE instruction, followed by EXECUTE or OPEN. The PREPARE instruction must initiate the EXECUTE FUNCTION/ PROCEDURE instruction.

For more details about stored procedure invocation, see SQL Programming.

Database Triggers

Triggers can be created for IBM® Informix® database tables with the CREATE TRIGGER instruction.

If you plan to support different types of database servers, you must be aware that each DB vendor has defined its own trigger creation syntax and stored procedure language. In such cases, you may consider writing most your business logic in BDL, and implementing only some triggers in the database, mainly to get better performance or use database features that only exist with stored procedures.

Genero BDL partially supports trigger creation:

- The Genero BDL static SQL syntax does not include the CREATE TRIGGER and DROP TRIGGER instructions. However, you can create database triggers by using dynamic SQL (EXECUTE IMMEDIATE).

Optimizer directives

IBM® Informix® SQL allows you to specify query optimization directives to force the query optimizer to use a different path than the implicit plan. With IBM® Informix®, optimizer directives are specified with the following SQL comment markers followed by a plus sign:

```
/*+ optimizer-directives */
{+  optimizer-directives }
--+ optimizer-directives
```

Genero BDL partially supports optimizer directives:

- The static SQL syntax does not allow the C-style optimizer syntax.
- The curly-brace and dash-dash optimizer directive syntaxes cannot be used in static SQL statements, because these correspond to the 4GL language comments.
- However, you can execute queries with optimization directives with Dynamic SQL.

  Tip: Optimization directives are not portable. If you plan to use different types of database servers, you should avoid the usage of query plan hints.

XML publishing support

IBM® Informix® IDS 11.10 introduced a set of XML built-in functions when the **idsxmlvp** virtual processor is turned on. Built-in XML functions are of two types: Those returning LVARCHAR values, and those returning CLOB values. For example, *genxml()* returns an LVARCHAR(32739), while *genxmlclob()* returns a CLOB. XML data is typically stored in LVARCHAR or CLOB columns.

Genero BDL partially supports XML functions:

- Because Genero BDL does not support BLOB/CLOB types, functions returning CLOB values cannot be used. You can however use the XML functions returning LVARCHAR values, and fetch the result into a VARCHAR variable of the appropriate size.
- Some of the XML functions such as genxml() take ROW() values as parameters. Because literal unnamed ROW() expressions are like regular function calls, you can use XML functions in static SQL statements.

Example:

```
FUNCTION get_cust_data(id)
   DEFINE id INT, v VARCHAR(5000)
   SELECT genxml(ROW(cust_name, cust_address), "custdata") INTO v
      FROM customers WHERE cust_id = id
   RETURN v
END FUNCTION
```

DataBlade® modules

IBM® Informix® IDS provides several database extensions implemented with the DataBlade® Application Programming Interface, such as MQ Messaging, Large OBjects management, Text Search DataBlades, Spatial DataBlade® Module, etc.

Genero BDL partially supports DataBlade® modules:

- DataBlade® extensions are based on User Defined Functions and User Defined Types. It is not possible to define program variables with specific User Defined Types. For example, you cannot define a program variable with the ST_Point type implemented by the Spatial DataBlade® module.
- The static SQL grammar does not support DataBlade® specific syntax. For example, it is not possible to create a Basic Text Search index with the USING bts clause of the CREATE INDEX statement.

However, as long as the syntax of the DataBlade® functions follows basic SQL expressions, it can be used in static SQL statements. For example, the next query uses the bts_contains() function of the Basic Text Search extension:

```
SELECT id FROM products WHERE bts_contains( brands, 'standard' )
```

You can also use Dynamic SQL to perform queries with a syntax that is not allowed in the static SQL grammar.

Specific CREATE INDEX clauses

In addition to the standard index-key specification using a column list, the CREATE INDEX statement supported by IBM® Informix® SQL allows specific clauses, for example to define storage options.

Genero BDL partially supports the CREATE INDEX statement; the following are not supported in static SQL grammar:

- The IF NOT EXISTS clause.
- Functional index specification is now allowed in the index-key list.
- Storage options such as IN *dbspace*, EXTEND SIZE, NEXT SIZE.
- The index mode clauses such as FILTERING WITH/WITHOUT ERROR.
- The USING clause.
- The HASH ON clause.
- The FILLFACTOR clause.

You can use Dynamic SQL to execute CREATE INDEX statements with clauses that are not allowed in the static SQL grammar.

Other SQL instructions

Genero BDL static SQL syntax implements common Data Manipulation Statements such as SELECT, INSERT, UPDATE and DELETE. Data Definition Language statements such as CREATE TABLE, CREATE INDEX, CREATE SEQUENCE and their corresponding ALTER and DROP statements are also part of the static SQL grammar. These are supported with a syntax limited to the standard SQL clauses. For example, Genero BDL might not support the most recent CREATE TABLE storage options supported by IBM® Informix® SQL.

Since the first days of the 4GL language the SQL language has been extended, and it has become so large that it's impossible to embed all the existing new statements without introducing grammar conflicts with the 4GL language. In addition, each DB vendor has improved the standard SQL language with proprietary SQL statements that are not portable; it would not be a good idea to use these specific instructions if you plan to make your application run with different types of database engines.

However, the Genero BDL static SQL is constantly improved with standard SQL syntax that works with most types of database servers. For example, Genero BDL supports the ANSI outer join syntax, constraints definition in DDL statements, sequence instructions, BIGINT and BOOLEAN data types, and there is more to come.

If a statement is unsupported in static SQL, that does not mean that you cannot execute it. If you want to execute an SQL instruction that is not part of the static SQL grammar, you can use Dynamic SQL as follows:

- Use PREPARE + EXECUTE for statements that do not generated a result set
- Use (PREPARE/) DECLARE + OPEN for statements returning a result set
- Use EXECUTE IMMEDIATE if no SQL parameters are required and no result set is generated

Dynamic SQL instructions take a string as the input, so there is no limitation regarding the SQL text you can execute; however, only one statement can be executed at a time. It is better, however, to write your SQL statements directly in static SQL when possible, because it makes the code more readable and the syntax is checked at compiled time.

For more details about statements supported in the static SQL syntax, see Static SQL.

Below is a list of the IBM® Informix® SQL statements that are not allowed in the static SQL syntax (last updated from IDS **11.50** SQL instructions). The IBM® Informix® SQL Syntax manual includes ESQL/C specific statements such as ALLOCATE DESCRIPTOR, which are not part of the basic SQL statements supported by the engines. ESQL/C specific statements are not listed here:

```
ALTER ACCESS_METHOD
ALTER FRAGMENT
ALTER FUNCTION
ALTER PROCEDURE
ALTER ROUTINE
ALTER SECURITY LABEL COMPONENT
CREATE ACCESS_METHOD
CREATE AGGREGATE
CREATE CAST
CREATE DISTINCT TYPE
CREATE EXTERNAL TABLE Statement
CREATE FUNCTION (with body)
CREATE OPAQUE TYPE
CREATE OPCLASS
CREATE PROCEDURE (with body)
CREATE ROLE
CREATE ROUTINE FROM
CREATE ROW TYPE
CREATE SCHEMA
CREATE SECURITY LABEL
CREATE SECURITY LABEL COMPONENT
CREATE SECURITY POLICY
CREATE TRIGGER
CREATE VIEW
CREATE XADATASOURCE
CREATE XADATASOURCE TYPE
DROP ACCESS_METHOD
DROP AGGREGATE
DROP CAST
DROP FUNCTION
DROP OPCLASS
DROP PROCEDURE
DROP ROLE
DROP ROUTINE
DROP ROW TYPE
DROP SECURITY
DROP TRIGGER
DROP TYPE
DROP XADATASOURCE
DROP XADATASOURCE TYPE
EXECUTE FUNCTION
EXECUTE PROCEDURE
GRANT FRAGMENT
INFO
MERGE
OUTPUT
RELEASE SAVEPOINT
RENAME COLUMN
RENAME DATABASE
RENAME SECURITY
REVOKE FRAGMENT
SAVE EXTERNAL DIRECTIVES
SAVEPOINT
SET AUTOFREE
SET COLLATION
SET CONSTRAINTS
SET DATASKIP
SET DEBUG FILE
```

```
SET ENCRYPTION PASSWORD
SET ENVIRONMENT
SET INDEXES
SET LOG
SET OPTIMIZATION
SET PDQPRIORITY
SET ROLE
SET SESSION AUTHORIZATION
SET STATEMENT CACHE
SET TRANSACTION
SET TRIGGERS
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE
```

Unsupported IBM® Informix® SQL features
Unsupported IBM® Informix® SQL features.
CLOB and BLOB data types

In addition to the TEXT and BYTE data types (known as Simple Large Objects), IBM® Informix® servers support the CLOB and BLOB types to store large objects. CLOB/BLOB are known as Smart Large Objects. The main difference is that Smart Large Objects support random access to the data - seek, read and write through the LOB as if it was a OS file.

Genero BDL does not support the CLOB and BLOB types:

- It is not possible to define BDL variables with the CLOB or BLOB types, so you cannot manipulate CLOB/BLOB objects within programs.
- Defining a TEXT / BYTE variable to hold CLOB / BLOB column data is not supported; you will get error -609 (Illegal attempt to use a Text/Byte host variable).
- The static SQL syntax for DDL statements like CREATE TABLE does not allow the CLOB / BLOB keywords for column types.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a CLOB or BLOB column.

You can, however:

- Create a table with CLOB/BLOB columns by using Dynamic SQL.
- Use the Smart Large Object functions FILETOBLOB(), FILETOCLOB(), LOCOPY(), LOTOFILE() in static SQL statements.

Enhancement reference: 476

The LIST data type

In IBM® Informix® databases, the LIST type is a collection type that can store ordered elements of a specific base type. Unlike the MULTISET type, the elements of a LIST have ordinal positions. Elements can be duplicated.

Genero BDL does not support the IBM® Informix® LIST data type.

- It is not possible to define BDL variables with the LIST type.
- The static SQL syntax does not support collection-related syntax elements:
  - DDL statements like CREATE TABLE cannot use the LIST keyword for column types,
  - The collection-derived notation `TABLE( )` is not allowed,
  - The `INSERT AT` *`position`* instruction is not supported,
  - The `LIST { }` literal syntax is not allowed.
  - The *`value`* `IN` *`identifier`* syntax is not allowed.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a LIST column.

The MULTISET data type

The MULTISET IBM® Informix® data type is a collection type that can store non-ordered elements of a specific base type. Unlike the LIST type, the elements of a MULTISET have no ordinal positions. Elements can be duplicated.

Genero BDL does not support the IBM® Informix® MULTISET data type:

- It is not possible to define BDL variables with the MULTISET type.
- The static SQL syntax does not support collection-related syntax elements:
  - DDL statements like CREATE TABLE cannot use the MULTISET keyword for column types,
  - The collection-derived notation `TABLE( )` is not allowed,
  - The `MULTISET { }` literal syntax is not allowed.
  - The *`value`* `IN` *`identifier`* syntax is not allowed.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a MULTISET column.

The SET data type

The SET IBM® Informix® data type is a collection type that stores non-ordered unique elements of a specific base type. Unlike the LIST type, the elements of a LIST have no ordinal positions. Elements cannot be duplicated.

Genero BDL does not support the IBM® Informix® SET data type:

- It is not possible to define BDL variables with the SET type.
- The static SQL syntax does not support collection-related syntax elements:
  - DDL statements like CREATE TABLE cannot use the SET keyword for column types,
  - The collection-derived notation `TABLE( )` is not allowed,
  - The `SET { }` literal syntax is not allowed.
  - The *`value`* `IN` *`identifier`* syntax is not allowed.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a SET column.

The ROW data types

IBM® Informix® supports the named and unnamed ROW data types. A ROW type is a complex type that combines several table columns. You create a ROW type with the CREATE ROW TYPE instruction, and then you can reuse the type definition for a table column.

Genero BDL does not support the IBM® Informix® ROW data types:

- It is not possible to define BDL variables with a named ROW type. The equivalent would be a RECORD variable, but data is not mapped directly from a structured ROW column, you must list individual fields of the ROW column.
- The static SQL syntax does not support ROW-related syntax elements:
  - The DDL statements CREATE ROW TYPE, DROP ROW TYPE, CREATE CAST and DROP CAST are not allowed,
  - In CREATE TABLE / ALTER TABLE DDL statements, the data type must be a built-in type.
  - The :: cast operator is not supported when specifying a ROW() literal. However, the CAST() expressions are allowed.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a column defined with a ROW type.

However:

- Static SQL allows multilevel single-dot notation, so you can, for example, identify a ROW field as *employee.address.city*.
- Dynamic SQL can be used to insert or update rows with ROW type columns.

- Individual ROW column fields can be fetched to BDL program variables, as long as the basic types match.

Enhancement reference: 19159

OPAQUE data types

Opaque User Defined Types can be implemented in IBM® Informix® with the CREATE OPAQUE TYPE statement. The storage structure of an OPAQUE type is unknown to the database server, data can only be accessed through user-defined routines.

Genero BDL does not support the IBM® Informix® OPAQUE data types:

- It is not possible to define BDL variables with an opaque type.
- The static SQL syntax does not support OPAQUE-related syntax elements:
  - The DDL statements CREATE OPAQUE TYPE, DROP TYPE, CREATE CAST and DROP CAST are not allowed,
  - In CREATE TABLE / ALTER TABLE DDL statements, the data type must be a built-in type.
  - The :: cast operator is not supported. However, the CAST() expressions are allowed.
- The fgldbsch schema extractor will report an invalid data type if you try to get the schema for a table with a column defined with a OPAQUE type.

The :: cast operator

IBM® Informix® SQL implements the :: cast operator and the CAST() expressions to do an explicit cast of a value:

```
CREATE TABLE tab ( v INTEGER )
INSERT INTO tab VALUES ( 123456::INTEGER )
SELECT 'abcdef'::CHAR(20)||'.' FROM tab
SELECT CAST('abcdef' AS CHAR(20))||'.' FROM tab
```

Genero BDL does not support the :: cast operator in the static SQL grammar. However, the CAST() expressions are allowed. If you need to use the :: cast operator, you must use Dynamic SQL to perform such queries.

Enhancement reference: 19190

Table inheritance

IBM® Informix® SQL allows you to define a table hierarchy through named row types. Table inheritance allows a table to inherit the properties of the supertable in the meaning of constraints, storage options, triggers. You must first create the types with CREATE ROW TYPE, then you can create the tables with the UNDER keyword to define the hierarchy relationship.

```
CREATE ROW TYPE person_t ( name VARCHAR(50) NOT NULL,
 address VARCHAR(200), birthdate DATE )
CREATE ROW TYPE employee_t ( salary INTEGER, manager VARCHAR(50) )
CREATE TABLE person OF TYPE person_t
CREATE TABLE employee OF TYPE employee_t UNDER person
```

A table hierarchy allows you to do SQL queries whose row scope is the supertable and its subtables. For example, after inserting one row in the person table and another one in the employee table, if you UPDATE the name column without a WHERE clause, it will update all rows from both tables. To limit the set of rows affected by the statement to rows of the supertable, you must use the ONLY keyword:

```
UPDATE ONLY(person) SET birthdate = NULL
SELECT * FROM ONLY(person)
```

Genero BDL static SQL grammar does not include the syntax elements related to table hierarchy management. You can however use Dynamic SQL to perform such queries.

Enhancement reference: 19200

SQL adaptation guide for IBM® DB2® UDB 10.x

Installation (Runtime Configuration)
IBM® DB2® related installation topics.
Install DB2® and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the IBM® DB2® Universal Server on your database server.
2. Create a DB2® database entity: *dbname*

   To create the database entity in DB2, use the graphical tool provided by IBM® called "DB2 Data Studio", or from the command line, use the `db2` command interpreter in a DB2 operating system user session (db2inst). Consider creating your database with the correct database locale (codeset and territory), for example:

   ```
   $ db2
   ...
   db2 => CREATE DATABASE dbname
            AUTOMATIC STORAGE YES
            USING CODESET UTF-8 TERRITORY EN_US
   DB20000I  The CREATE DATABASE command completed successfully.
   ```
3. Connect to the new created database with the DB2 administrator user.

   Open a database connect in the DB2 Data Studio, or use the `db2` command interpreter as in the following example:

   ```
   db2 => connect to dbname

      Database Connection Information

    Database server        = DB2/LINUX 10.1.0
    SQL authorization ID   = DB2INST
    Local database alias   = dbname
   ```
4. Declare a database user dedicated to your application: the application administrator. This user will manage the database schema of the application (all tables will be owned by it).

   Create the user with the DB2 Data Studio, or use the `db2` command interpreter as follows:

   ```
   db2 => GRANT CONNECT ON DATABASE TO USER appadmin
   DB20000I  The SQL command completed successfully.
   ```
5. Give all requested database administrator privileges to the application administrator.

   Grant the privileges to the new created user in the DB2 Data Studio, or use the `db2` command interpreter as follows:

   ```
   db2 => GRANT CREATETAB ON DATABASE TO USER appadmin
   DB20000I  The SQL command completed successfully.
   ```
6. If you plan to use temporary table emulation, you must setup the database for DB2® global temporary tables (create a user temporary tablespace and grant privileges to all users).

   See Temporary tables on page 560.
7. Connect as the application administrator:

   Open a new database connect in the DB2 Data Studio, or use the `db2` command interpreter as follows:

   ```
   db2 => connect to dbname user appadmin using password
   ```

```
  Database Connection Information

Database server        = DB2/LINUX 10.1.0
SQL authorization ID   = DB2INST
Local database alias   = dbname
```

8. Create the application tables with `CREATE TABLE` statements.

   Convert Informix® data types to DB2® data types. See issue Data Type Conversion Table for more details.

9. If you plan to use SERIAL column emulation, you must prepare the database.

   See SERIAL data types on page 551.

Prepare the runtime environment - connecting to the database

1. In order to connect to IBM® DB2®, you must have the database driver "`dbmdb2`" in `FGLDIR/dbdrivers`.
2. If you want to connect to a remote DB2® server, the "IBM® DB2® Client Application Enabler" must be installed and configured on the computer running the BDL applications.

   You must declare the data source set up as follows:

   a) Login as root.

      1. Create a user dedicated to the db2 client instance environment, for example, "db2cli1".
      2. Create a client instance environment with the `db2icrt` tool as in following example:

         ```
         # db2dir /instance/db2icrt -a server -s client instance-user
         ```

   b) Login as the instance user (environment should be set automatically, verify DB2DIR).

      1. Catalog the remote server node:

         ```
         # db2 "catalog tcpip node db2node remote hostname server tcp-service"
         ```

      2. Catalog the remote database:

         ```
         # db2 "catalog database datasource at node db2node authentication
          server"
         ```

      3. Test the connection to the remote database:

         ```
         # db2 "connect to datasource user dbuser using password"
         ```

         (where *dbuser* is a database user declared on the remote database server)

   See IBM® DB2® documentation for more details.

3. **Important:** If you have a non-English environment, you may need to set the `PATCH2=15` configuration parameter in the `DB2CLI.INI` file to ensure that DECIMAL values will be properly inserted or fetched:

   ```
   [datasource]
   PATCH2=15
   ```

   For more details, see the DB2® README.TXT file in the SQLLIB directory.

4. Make sure that the DB2® client environment variables are properly set.

   Check variables such as DB2DIR (the path to the installation directory), DB2INSTANCE (the name of the DB2® instance), INSTHOME (the path to the home directory of the instance owner). On UNIX™, you will find environment settings in the file `$INSTHOME/sqllib/db2profile`. See IBM® DB2® documentation for more details.

5. Check the database client locale settings (DB2CODEPAGE, etc).

The database client locale must match the locale used by the runtime system (LC_ALL, LANG).

6. Verify the environment variable defining the search path for DB2 CLI database client shared libraries (libdb2.so on UNIX™, DB2CLI.DLL on Windows™).

Table 161: Shared library environment setting for DB2® UDB version

DB2® UDB version	Shared library environment setting
DB2® UDB 9.x and higher	*UNIX*™: Add $DB2DIR/lib32 (for 32 bit) or $DB2DIR/lib64 (for 64 bit) to LD_LIBRARY_PATH (or its equivalent). *Windows*™: Add %DB2DIR%\bin to PATH.

7. To verify if the DB2® client environment is correct, you can, for example, start the db2 command interpreter and connect to the server:

```
$ db2
db2 => CONNECT TO dbname USER username USING password
```

8. Setup the **fglprofile** entries for database connections.
   a) Define the IBM DB2 database driver:

```
dbi.database.dbname.driver = "dbmdb2"
```

   b) The "source" parameter defines the name of the IBM DB2 database name.

```
dbi.database.dbname.source = "test1"
```

   c) Define the database schema selection if needed:

   Use the following entry to define the database schema to be used by the application. The database interface will automatically perform a SET SCHEMA *name* instruction to switch to a specific schema:

```
dbi.database.dbname.db2.schema = 'name'
```

   Here *dbname* identifies the database name used in the BDL program (DATABASE *dbname*) and *name* is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.

Database concepts
IBM® DB2® related database concept topics.
Database concepts

As with Informix®, an IBM® DB2® database server can handle more than one database entity. Informix® servers have an ID (INFORMIXSERVER) and databases are identified by name. IBM® DB2® instances are identified by the DB2INSTANCE environment variable and databases have to be cataloged as data sources (see IBM® DB2® documentation for more details).

Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from Informix® to IBM® DB2®. Most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be reused for the IBM® DB2® database.

Storage concepts are quite similar in Informix® and in IBM® DB2®, but the names are different.

These tables compares Informix® storage concepts to IBM® DB2® storage concepts:

Table 162: Physical units of storage (Informix® vs. DB2®)

Informix®	IBM® DB2®
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage. You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	One or more "**containers**" are created for each "tablespace" to physically store the data of all logical structures. Like Informix® "chunks", "containers" can be an OS file or a raw device. You can add "containers" to a "tablespace" in order to increase the size of the logical unit of storage or you can define EXTEND options.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	At the finest level of granularity, IBM® DB2® stores data in "**data blocks**" with size corresponding to a multiple of the operating system's block size. You set the "data block" size when creating the database.
An "**extent**" consists of a collection of contiguous "pages" that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	An "**extent**" is a specific number of contiguous "data blocks", obtained in a single allocation. When creating a table, you can specify the first extent size and the size of future extents with the STORAGE() option. For a single table, "extents" can be located in different "data files" of the same "tablespace".

Table 163: Logical units of storage (Informix® vs. DB2®)

Informix®	IBM® DB2®
A "**table**" is a logical unit of storage that contains rows of data values.	Same concept as Informix®.
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as Informix®. An IBM® DB2® instance can manage several databases.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, Informix® defaults to the current database dbspace.	Database tables are created in a specific "**tablespace**", which defines a logical place to store data. The main difference with Informix® "dbspaces", is that IBM® DB2® tablespaces belong to a "database", while Informix® "dbspaces" are external to a database.

Table 164: Other storage concepts (Informix® vs. DB2®)

Informix®	IBM® DB2®
When initializing an Informix® engine, a "**root dbspace**" is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.).	Each IBM® DB2® database uses a set of "**control files**" to store internal information. These files are located in a dedicated directory: ".../$DB2INSTANCE/NODEnnnn"
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing. The "**logical log**" is a set of "**logical-log files**" used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with transaction log. Informix® combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.	DB2® uses "**database log files**" to record SQL transactions.

Data consistency and concurrency

Data consistency involves readers that want to access data currently modified by writers and *concurrency data access* involves several writers accessing the same data for modification. *Locking granularity* defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®

Informix® uses a locking mechanism to manage data consistency and concurrency. When a process modifies data with UPDATE, INSERT or DELETE, an exclusive lock is set on the affected rows. The lock is held until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks according the isolation level. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode.

Control:

- Isolation level: SET ISOLATION TO ...
- Lock wait mode: SET LOCK MODE TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit locking: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is **read committed**.
- The default lock wait mode is "not wait".
- The default locking granularity is on per page.

IBM® DB2®

As in Informix®, IBM® DB2® uses locks to manage data consistency and concurrency. The database manager sets exclusive locks on the modified rows and shared locks when data is read, according to the isolation level. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction. The lock granularity is at the row or table level. For more details, see DB2's Administration Guide, "Application Consideration".

Control:

- Lock wait mode: Always WAIT. Only the Lock Timeout can be changed, but this is a global database parameter.
- Isolation level: Can be set through an API function call or with a database client configuration parameter.
- Locking granularity: Row level or Table level.
- Explicit locking: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Cursor Stability (readers cannot see uncommitted data, no shared lock is set when reading data).

Solution

The SET ISOLATION TO ... Informix® syntax is replaced by an ODBC API call setting the SQL_ATTR_TXN_ISOLATION connection attribute. The next table shows the isolation level mappings done by the database driver:

Table 165: Isolation level mappings done by the IBM® DB2® UDB database driver

SET ISOLATION instruction in program	ODBC SQL_ATTR_TXN_ISOLATION connection attribute
SET ISOLATION TO DIRTY READ	SQL_TXN_READ_UNCOMMITTED
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	SQL_TXN_READ_COMMITTED
SET ISOLATION TO CURSOR STABILITY	SQL_TXN_REPEATABLE_READ
SET ISOLATION TO REPEATABLE READ	SQL_TXN_SERIALIZABLE

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

See Informix® and IBM® DB2® documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Informix® and IBM® DB2® handle transactions differently. The differences in the transactional models can affect the program logic.

- Informix® native mode (non ANSI):
  - DDL statements can be executed (and canceled) in transactions.
  - Transactions must be started with `BEGIN WORK`. Statements executed outside of a transaction are automatically committed.
- IBM® DB2®:
  - DDL statements can be executed (and canceled) in transactions.
  - Beginning of transactions are implicit; two transactions are delimited by `COMMIT` or `ROLLBACK`.

Transactions in stored procedures:

Avoid using transactions in stored procedures to allow the client applications to handle transactions, in accordance with the transaction model.

Savepoints:

- Informix® version 11.50 introduces **savepoints** with the following instructions:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] ]
RELEASE SAVEPOINT name
```

- IBM® DB2® supports **savepoints** too. However, there are differences:

1. Savepoints must be declared with the `ON ROLLBACK RETAIN CURSORS` clause
2. Rollback must always specify the savepoint name

Solution

The Informix® behavior is simulated with an autocommit mode in the IBM® DB2® interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program. Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with IBM® DB2®.

> **Note:** If you want to use savepoints, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`.

See also SELECT FOR UPDATE

Database users

Until version 11.70.xC2, Informix® database users had to be created at the operating system level and be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers. Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

IBM® DB2® users are operating system users with a specific DB2® environment. The database administrator must grant the CONNECT authority to these users.

- Database *authorities* involve actions on a database as a whole. When a database is created, some authorities are automatically granted to anyone who accesses the database. For example, CONNECT, CREATETAB, BINDADD and IMPLICIT_SCHEMA authorities are granted to all users.
- Database *privileges* involve actions on specific objects within the database. When a database is created, some privileges are automatically granted to anyone who accesses the database. For example, SELECT privilege is granted on catalog views and EXECUTE and BIND privilege on each successfully bound utility is granted to all users.

Together, privileges and authorities act to control access to an instance and its database objects. Users can access only those objects for which they have the appropriate authorization, that is, the required privilege or authority.

> **Note:** As in Informix®, DB2® user names that connect to the database server must be a maximum of **eight** characters long.

Solution

Set up the IBM® DB2® environment for each user as described in the documentation.

Setting privileges

Informix® and IBM® DB2® user privileges management is quite similar.

- IBM® DB2® provides user groups to define.
- IBM® DB2® users must have at least the CONNECT authority to access the database.

  `GRANT CONNECT ON DATABASE TO (PUBLIC|user|group)`
- Informix® users must have at least the CONNECT privilege to access the database:

  `GRANT CONNECT TO (PUBLIC|user)`

Solution

Make sure DB2® users have the right privileges to access the database.

See also Temporary Tables

Data dictionary
IBM® DB2® related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way: As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

IBM® DB2® 9.x does not implement a BOOLEAN SQL type.

Solution

The DB2® database interface converts BOOLEAN type to CHAR(1) columns and stores '1' or '0' values in the column.

CHARACTER data types

Informix® supports the following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases)

IBM® DB2® implements the following character data types:

- CHAR(N) with N<= 254 bytes
- VARCHAR(N) with N <= 32672 bytes
- GRAPHIC(N) with N <= 127 characters
- VARGRAPHIC(N) with N <=16336 characters

Like Informix®, IBM® DB2® uses Byte Length Semantics to define the length of CHAR/VARCHAR columns. However, GRAPHIC and VARGRAPHIC lengths are specified in characters (i.e. max number of double-byte characters).

The character set used by DB2® to store CHAR and VARCHAR data is defined in the database locale section when creating a new database. If your application uses UTF-8, consider creating the DB2 database with the UTF-8 codeset.

DB2® can automatically convert from/to the client and server characters sets. In the client applications, you define the character set with the DB2CODEPAGE profile variable.

Solution

Informix® CHAR(N) types must be mapped to DB2® CHAR(N) types, and Informix® VARCHAR(N) or LVARCHAR(N) columns must be mapped to DB2® VARCHAR(N).

Important:

- DB2® does not support NCHAR/NVARCHAR types. If your programs create tables with these types, you must review your code. The DB2® driver does not automatically convert the NCHAR/NVARCHAR Informix® types to GRAPHIC/VARGRAPHIC, because the meaning of the length is different.
- Check that your database schema does not use CHAR or VARCHAR types with a length exceeding the DB2® limits. Especially, the Informix® CHAR type has a very long size limit compared to DB2® CHAR.

When using a multibyte character set (such as UTF-8), if the DB2 database was created with the appropriate codeset (UTF-8), you can use the CHAR/VARCHAR columns, and user byte length semantics in programs. If the database code set is non multi-byte, you must use the GRAPHIC and VARGRAPHIC data types to store multi-byte character data, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a DB2® database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in DB2® database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

See also the section about Localization.

NUMERIC data types

Informix® provides the following data types to store numbers:

Table 166: Informix® numeric data types

Informix® data type	Description
SMALLINT	16 bit signed integer
INT / INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
INT8	64 bit signed integer (replaced by BIGINT)
DEC / DECIMAL	Equivalent to DECIMAL(16)
DEC(p) / DECIMAL(p)	Floating-point decimal number
DEC(p,s) / DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL / SMALLFLOAT	32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)]	64-bit floating point decimal (C double)

Most data types supported by IBM® DB2® UDB are compatible to Informix® data types. DB2® V 9.1 introduces the DECFLOAT(16) and DECFLOAT(34) floating point decimal types to store large decimals. The next table lists the Informix® types and DB2® equivalents.

Table 167: Informix® numeric data types and DB2® equivalents

Informix® data type	IBM® DB2® equivalent
INT8	Use BIGINT instead
DECIMAL(p)	With DB2® **V9.1**, DECIMAL(p<=16) can be stored in DECFLOAT(16) and DECIMAL(p>16) can be stored in DECFLOAT(34). With older versions of DB2®, we can use DECIMAL(p*2,p), but with a limitation of 15 for the original Informix® DECIMAL precision.
DECIMAL(32,s)	DB2® decimals maximum precision is 31 digits!
MONEY	DECIMAL(16,2)
MONEY(p)	DECIMAL(p,2)
MONEY(p,s)	DECIMAL(p,s)
SMALLFLOAT	REAL
FLOAT[(n)]	FLOAT[(n)] (DOUBLE)

Solution

SQL scripts to create databases must be converted manually. Tables created from BDL programs do not have to be converted; the database interface detects the MONEY data type and uses the DECIMAL type for DB2®.

The maximum precision for DB2® decimals is 31 digits, while Informix® supports 32 digits.

- When using DB2® V8 and prior:

  There is no DB2® equivalent for the Informix® DECIMAL(p) floating point decimal (i.e. without a scale). If your application is using such data types, you must review the database schema in order to use DB2® compatible types. To workaround the DB2® limitation, the DB2® database drivers convert DECIMAL(p) types to a DECIMAL(2*p, p), to store all possible numbers an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 15 ((2*15) = 30), since DB2® maximum DECIMAL precision is 31. If the original precision is bigger than 15, a CREATE TABLE statement executed from a Genero program will fail with a DB2® SQLSTATE 42611.
- When using DB2® V9.1 and higher:

  The DECIMAL(p) data type is converted to DECFLOAT(16) (for p<=16) or DECFLOAT(34) (for p>16) to store floating point decimals. If you create tables with DECFLOAT columns, you will lose the original DECIMAL precision when extracting the schema with fgldbsch, because IBM® DB2® supports only two precision specifications (16 or 34). Note also the DECFLOAT(34) will be extracted as DECIMAL(32), since the Genero DECIMAL type has a maximum precision of 32 digits.

DATE and DATETIME data types

Informix® provides two data types to store date and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

IBM® DB2® provides following data type to store dates:

- DATE = for year, month, day storage.
- TIME = for hour, minute, second storage.

- TIMESTAMP = for year, month, day, hour, minute, second, fraction storage.

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string content matches environment parameters (i.e. DBDATE, GL_DATETIME). As Informix®, IBM® DB2® can convert quoted strings to dates, times or timestamps. Only one format is possible: 'yyyy-mm-dd' for dates, 'hh:mm:ss' for times and'yyyy-mm-dd hh:mm:ss:f' for timestamps.

Date time arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- In IBM® DB2®, the result of an arithmetic expression involving DATE values is a NUMBER of days, the decimal part is the fraction of the day (0.5 = 12H00, 2.00694444 = (2 + (10/1440)) = 2 days and 10 minutes)).
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. IBM® DB2® does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix® specific and have no equivalent in IBM® DB2®.

Solution

DB2® has the same DATE data type as Informix® DATE columns.

DB2® TIME data type can be used to store Informix® DATETIME HOUR TO SECOND values. The database interface makes the conversion automatically.

Informix® DATETIME values with any precision from YEAR to FRACTION(5) can be stored in DB2® TIMESTAMP columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the DB2® TIMESTAMP value will be "1900-01-01 11:45:00.0".

Important:

- Using integers as a number of days in an expression with dates is not supported by IBM® DB2®. Check your code to detect where you are using integers with DATE columns.
- Literal DATETIME and INTERVAL expressions (i.e. DATETIME (1999-10-12) YEAR TO DAY) are not converted.
- It is strongly recommended that you use BDL variables in dynamic SQL statements instead of quoted strings representing DATEs. For example:

  ```
  LET stmt = "SELECT ... FROM customer WHERE creat_date >'", adate,"'"
  ```

  is not portable, use a question mark place holder instead and OPEN the cursor USING adate:

  ```
  LET stmt = "SELECT ... FROM customer WHERE creat_date > ?"
  ```
- DATE arithmetic expressions using SQL parameters (USING variables) are not fully supported. For example:

  `"SELECT ... WHERE datecol < ? +1" generates an error at PREPARE time.`
- SQL Statements using expressions with TODAY / CURRENT / EXTEND must be reviewed and adapted to the native syntax.

INTERVAL data type

Informix® INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

DB2® does not provide a data type corresponding the Informix® INTERVAL data type.

Solution

The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, BDL INTERVAL values can be stored into and retrieved from CHAR columns.

SERIAL data types

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a `SELECT dbinfo('bigserial')` query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

IBM® DB2® version 7.1 supports IDENTITY columns:

```
CREATE TABLE tab ( k INTEGER GENERATED ALWAYS AS IDENTITY);
```

To get the last generated IDENTITY value after an INSERT, DB2® provides the following function:

```
IDENTITY_VAL_LOCAL()
```

IBM® DB2® version 8.1 supports SEQUENCES:

```
CREATE SEQUENCE sq1 START WITH 100;
```

To create a new sequence number, you must use the "NEXTVAL FOR" operator:

```
INSERT INTO table VALUES ( NEXTVAL FOR sq1, ... )
```

To get the last generated sequence number, you must use the "PREVVAL FOR" operator:

```
SELECT PREVVAL FOR sq1 ...
```

Solution

To emulate Informix® serials with IBM® DB2®, you can use IDENTITY columns (1), or insert triggers using sequences (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification.

> **Important:** The trigger-based solution is provided to simplify the conversion from Informix, but is slower as the solution using identity columns. We strongly recommend that you use native IDENTITY columns instead to get best performances.

The method used to emulate SERIAL types is defined by the `ifxemul.datatype.serial.emulation` FGLPROFILE parameter:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = {"native"|"trigseq"}
```

- native: uses IDENTITY columns.
- trigseq: uses insert triggers with sequences.

The default emulation technique is "`native`".

This entry must be used in conjonction with:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial = {true|false}`

If the `datatype.serial` entry is set to false, the emulation method is ignored.

Using the `native` serial emulation

In database creation scripts, all SERIAL[(n)] data types must be converted by hand to:

```
INTEGER GENERATED ALWAYS AS IDENTITY[( START WITH n, INCREMENT BY 1)]
```

while the SERIAL8 and BIGSERIAL[(n)] types must be converted to:

```
BIGINT GENERATED ALWAYS AS IDENTITY[( START WITH n, INCREMENT BY 1)]
```

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to an IDENTITY specification.

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a call to the IDENTITY_VAL_LOCAL() function. However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must use the IDENTITY_VAL_LOCAL() function.

Since IBM® DB2® does not allow you to specify the value of IDENTITY columns, it is mandatory to convert all INSERT statements to remove the SERIAL column from the list. For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.*) -- will use the serial column
```

must be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

Using the `trigseq` serial emulation

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, SERIAL8/BIGSERIAL must be converted to BIGINT, and you must create a sequence and a trigger for each table using a SERIAL. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the sequence and the insert trigger.

Note:

IBM® DB2® performs NOT NULL data controls before the execution of triggers. If the serial column must be NOT NULL (for example, because it is part of the primary key), you cannot specify a NULL value for that column in INSERT statements.

For example, the following statement:

```
INSERT INTO tab VALUES (NULL,p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Important:

- IBM® DB2® triggers are not automatically dropped when the corresponding table is dropped. They become ***inoperative*** instead. Database administrators must take care of this behavior when managing schemas.
- With IBM® DB2®, INSERT statements using NULL for the SERIAL column will produce a new serial value:

  ```
  INSERT INTO tab ( col_serial, col_data ) VALUES ( NULL, 'data' )
  ```

  This behavior is mandatory in order to support INSERT statements which do not use the serial column:

  ```
  INSERT INTO tab (col_data) VALUES ('data')
  ```

  Check if your application uses tables with a SERIAL column that can contain a NULL value.
- With DB2®, trigger creation is not allowed on temporary tables. Therefore, the "**trigseq**" method cannot work with temporary tables using serials.

ROWIDs

When creating a table, Informix® automatically adds a "ROWID" integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

IBM® DB2® ROWID columns were introduced in version 9.7. Unlike Informix® integer row ids, DB2® row ids are based on VARCHAR(16) FOR BIT DATA (128 bit integer) that are usually represented as a 32 char hexadecimal representation of the value. The IBM® DB2® ROWID is actually an alternative syntax for RID_BIT(), and a qualified reference to ROWID like *tablename*.ROWID is equivalent to RID_BIT(*tablename*).

For example: `x'070000000000000000000065CE770000'`

In DB2® SQL, to find a row with a rowid, you must specify the rowid value as an hexadecimal value:

```
SELECT * FROM customer WHERE ROWID = x'070000000000000000000065CE770000'
```

or convert the ROWID to an hexadecimal representation and then you can compare to a simple string:

```
SELECT * FROM customer WHERE HEX(ROWID) ='070000000000000000000065CE770000'
```

With Informix®, SQLCA.SQLERRD[6] contains the ROWID of the last INSERTed or UPDATEd row. This is not supported with DB2 because DB2 are not INTEGERs.

Solution

If the BDL application uses ROWIDs, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported).

The DB2® database driver will convert the ROWID keyword to HEX(ROWID), so it can be used as a VARCHAR(32) with the hexadecimal representation of the BIT DATA. You need however to replace all INTEGER variable definitions by VARCHAR(32) or CHAR(32).

To emulate Informix® integer ROWIDs, you can also use the DB2® GENERATE_UNIQUE built-in function, or the IDENTITY attribute of the INTEGER or BIGINT data types.

All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the IBM® DB2® interface.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

IBM® DB2® supports the LONG VARCHAR/CLOB and BLOB/VARGRAPHIC/DBCLOB types for large objects storage

Solution

The DB2® database interface can convert BDL TEXT data to CLOB and BYTE data to BLOB. Note that DB2® CLOB and BLOB columns are created with a size of 500K, while Genero TEXT/BYTE program variables have a limit of 2 gigabytes; make sure that the large object data does not exceed this limit.

Constraints

Constraint naming syntax

Both Informix® and DB2® support primary key, unique, foreign key, default and check constraints. But the constraint naming syntax is different: DB2® expects the "CONSTRAINT" keyword **before** the constraint specification, and Informix® expects it **after** .

UNIQUE constraint example:

Table 168: UNIQUE constraint example (Informix® vs IBM® DB2®)

Informix®	**IBM® DB2®**

`CREATE TABLE emp (`
`  ...`
`  emp_code CHAR(10) UNIQUE`
`  CONSTRAINT pk_emp,`
	`CREATE TABLE emp (`
	`  ...`
	`  emp_code CHAR(10)`
	`  CONSTRAINT pk_emp UNIQUE,`
	`  ...`

Primary keys

Like Informix®, DB2® creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using "CREATE UNIQUE INDEX" to define unique constraints is obsolete (use primary keys or a secondary key instead).

> **Note:** DB2® primary key constraints do not allow NULLs; make sure your tables do not contain NULLs in the primary key columns.

Unique constraints

Like Informix®, DB2® creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

> **Note:** DB2® unique constraints do not allow NULLs; make sure your tables do not contain NULLs in the unique columns.

Foreign keys

Both Informix® and DB2® support the ON DELETE CASCADE option.

Check constraints

The check condition may be any valid expression that can be evaluated to TRUE or FALSE,including functions and literals. You must verify that the expression is not Informix-specific.

Null constraints

Informix® and DB2® support NOT NULL constraints, but Informix® does not allow you to give a name to "NOT NULL" constraints.

Solution

Constraint naming syntax: The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for DB2®.

Triggers

Informix® and IBM® DB2® provide triggers with similar features, but the trigger creation syntax and the programming languages are totally different.

Informix® triggers define which stored procedures must be called when a database event occurs (before | after insert | update | delete ...), while IBM® DB2® triggers can hold a procedural block.

IBM® DB2® provides specific syntax to define triggers. See documentation for more details.

Solution

Informix® triggers must be converted to IBM® DB2® triggers "by hand".

Stored procedures

Both Informix® and IBM® DB2® support stored procedures and user functions, but the programming languages are totally different.

Solution

Informix® stored procedures must be converted to IBM® DB2® "by hand".

See SQL Programming for more details about executing stored procedures with DB2®.

Name resolution of SQL objects

Case sensitivity in object names:

- Informix® database object names are not **case-sensitive** in non-ANSI databases.

```
CREATE TABLE Tab1 ( Key INT, Col1 CHAR(20) )
SELECT COL1 FROM TAB1
```

- IBM® DB2® database object names are case-sensitive. When a name is used without double quotes, it is automatically converted to uppercase letters. When using double quotes, the names are not converted:

```
CREATE TABLE tab1 ( Key INT, Col1 CHAR(20) )
-- Table name is "TAB1", column names are "KEY" and "COL1"
CREATE TABLE "Tab1" ( "Key" INT, "Col1" CHAR(20) )
-- Table name is "Tab1", column names are "Key" and "Col1"
```

The DB2® schema concept:

With non-ANSI Informix® databases, you do not have to give a schema name before the tables when executing an SQL statement.

`SELECT ... FROM table-name WHERE ...`

In an IBM® DB2® database, tables always belong to a database **schema**. When executing a SQL statement, a schema name must be used as the high-order part of a two-part object name, unless the current schema corresponds to the table's schema.

The default (implicit) schema is the current user's name but it can be changed with the "SET SCHEMA" instruction.

Example: The table "TAB1" belongs to the schema "SCH1". User "MARK" (implicit schema is "MARK") wants to access "TAB1" in a SELECT statement:

```
SELECT ... FROM TAB1 WHERE ...
-- Error "MARK"."TAB1" is an undefined name. SQLSTATE=42704
SELECT ... FROM SCH1.TAB1 WHERE ...
-- OK.
SET SCHEMA SCH1
-- Changes the current schema to SCH1.
SELECT ... FROM TAB1 WHERE ...
-- OK.
```

> **Note:** When executing the "SET SCHEMA" instruction, the database interface does not use double quotes around the schema name (= name is converted to uppercase letters). Make sure that the schema name is created with uppercase letters in the database.

DB2® provides "**aliases**", but they cannot be used to make a database object name public because aliases belong to schemas also.

Solution

Case sensitivity in object names:

Avoid the usage of double quotes around the database object names. All names will be converted to uppercase letters.

The DB2® schema concept:

After a connection, the database interface can automatically execute a `SET SCHEMA name` instruction if the following FGLPROFILE entry is defined:

`dbi.database.dbname.db2.schema= "name"`

Here *dbname* identifies the database name used in the BDL program (DATABASE *dbname*) and *name* is the schema name to be used in the SET SCHEMA instruction. If this entry is not defined, no "SET SCHEMA" instruction is executed and the current schema defaults to the user's name.

Examples:

```
dbi.database.stores.db2.schema= "STORES1"
dbi.database.accnts.db2.schema= "ACCSCH"
```

> **Note:** DB2® does not check the schema name when the SET SCHEMA instruction is executed. Setting a wrong schema name results in "undefined name" errors when performing subsequent SQL instructions like SELECT, UPDATE, INSERT.

In accordance with this automatic schema selection, you must create a DB2® schema for your application:

1. Connect as a user with the DBADM authority.
2. Create an administrator user dedicated to your application. For example, "STORESADM". Make sure this user has the IMPLICIT_SCHEMA privilege (this is the default in DB2®).
3. Connect as the application administrator "STORESADM" to create all database objects (tables, indexes, ...). In our example, a "STORESADM" schema will be created implicitly and all database objects will belong to this schema.

As a second option you can create a specific schema with the following SQL command:

```
CREATE SCHEMA "name" AUTHORIZATION "appadmin"
```

See the IBM® DB2® manuals for more details about schemas.

The ALTER TABLE instruction

Informix® and IBM® DB2® use different implementations of the ALTER TABLE instruction. For example:

- Informix® allows you to use multiple ADD clauses separated by commas. DB2® does not expect parentheses and the comma separator:

  Informix®:

  ```
  ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
  ```

  IBM® DB2®:

  ```
  ALTER TABLE customer ADD col1 INTEGER ADD col2 CHAR(20)
  ```

- Depending on the values currently stored, Informix® can change the data type of a column, while DB2® only supports changing the size of CHAR and VARCHAR columns:

  Informix®:

  ```
  ALTER TABLE customer MODIFY ( col1 INTEGER )
  ```

  IBM® DB2®:

  ```
  ALTER TABLE customer ALTER COLUMN col1 SET data type VARCHAR(200)
  ```

Solution

No automatic conversion is done by the database interface. Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

Data type conversion table: Informix to DB2

Table 169: Data type conversion table (Informix to DB2 UDB)

Informix® data types	**DB2® data types (V<9.1)**	**DB2® data types (V>=9.1)**
CHAR(n)	CHAR(n) (limit = 254c!)	CHAR(n) (limit = 254c!)
VARCHAR(n[,m])	VARCHAR(n) (limit = 32672c!)	VARCHAR(n) (limit = 32672c!)
LVARCHAR(n)	VARCHAR(n) (limit = 32672c!)	VARCHAR(n) (limit = 32672c!)
NCHAR(n)	N/A	N/A
NVARCHAR(n[,m])	N/A	N/A
BOOLEAN	CHAR(1)	CHAR(1)
SMALLINT	SMALLINT	SMALLINT
INT / INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
INT8	BIGINT	BIGINT
SERIAL[(start)]	INTEGER (see note 1)	INTEGER (see note 1)
BIGSERIAL[(start)]	BIGINT (see note 1)	BIGINT (see note 1)
SERIAL8[(start)]	BIGINT (see note 1)	BIGINT (see note 1)

Informix® data types	**DB2® data types (V<9.1)**	**DB2® data types (V>=9.1)**
DOUBLE PRECISION / FLOAT[(n)]	FLOAT[(n)] / DOUBLE	FLOAT[(n)] / DOUBLE
REAL / SMALLFLOAT	REAL	REAL
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s) (limit = 31 digits)	DECIMAL(p,s) (limit = 31 digits)
NUMERIC / DEC / DECIMAL(p) with p<=15	DECIMAL(2*p,p)	DECFLOAT(16)
NUMERIC / DEC / DECIMAL(p) with p>15	N/A	DECFLOAT(16) if p=16, DECFLOAT(34) if p>16
NUMERIC / DEC / DECIMAL	N/A	DECFLOAT(34)
MONEY(p,s)	DECIMAL(p,s) (limit = 31 digits)	DECIMAL(p,s) (limit = 31 digits)
MONEY(p)	DECIMAL(p,2) (limit = 31 digits)	DECIMAL(p,2) (limit = 31 digits)
MONEY	DECIMAL(16,2)	DECIMAL(16,2)
DATE	DATE	DATE
DATETIME HOUR TO SECOND	TIME	TIME
DATETIME q1 TO q2 (different from above)	TIMESTAMP	TIMESTAMP
INTERVAL q1 TO q2	CHAR(50)	CHAR(50)
TEXT	CLOB(500K)	CLOB(500K)
BYTE	BLOB(500K)	BLOB(500K)

Notes:

1. For more details about serial emulation, see SERIAL data types on page 551.

Data manipulation
IBM® DB2® related data manipulation topics.
Reserved words

Even if IBM® DB2® allows SQL reserved keywords as SQL object names ("create table table (column int)"), you should take care in your existing database schema and check that you do not use DB2® SQL words. An example of a common word which is part of DB2® SQL grammar is '**alias** '.

Solution

See IBM® DB2® documentation for reserved keywords.

Outer joins

The original OUTER join syntax of Informix® is different from the IBM® DB2® outer join syntax:

- In Informix® SQL, outer tables are defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM cust, OUTER(order)
 WHERE cust.key = order.custno

SELECT ... FROM cust,OUTER(order,OUTER(item))
 WHERE cust.key = order.custno
    AND order.key = item.ordno
    AND order.accepted = 1
```

- IBM® DB2® supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                     ON cust.key = order.custno

SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
              ON cust.key = order.custno
 WHERE order.accepted = 1
```

See the IBM® DB2® SQL reference for a complete description of the syntax.

Solution

For better SQL portability, you should use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The IBM® DB2® interface can convert most Informix® OUTER specifications to IBM® DB2® outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must figure from left to right in the order of outer levels. Example which does not work: "FROM OUTER(tab2), tab1".
2. The outer join in the WHERE clause must use the table name as prefix. Example: "WHERE tab1.col1 = tab2.col2".

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   Example: `"... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10"`
2. Statements composed by 2 or more SELECT instructions using OUTERs are not supported.

   Example: `"SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"`

   Note:

   - Table aliases are detected in OUTER expressions.

     `OUTER example with table alias: "OUTER( tab1 alias1)"`
   - In the outer join, *outer table.column-name* can be placed on both right or left sides of the equal sign.

     `OUTER join example with table on the left: "WHERE outertab.col1 = maintab.col2 "`
   - Table names detection is not case-sensitive.

     Example: `"SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2"`
   - Temporary tables are supported in OUTER specifications.

Transactions handling

Informix® and IBM® DB2® handle transactions differently. The differences in the transactional models can affect the program logic.

- Informix® native mode (non ANSI):
  - DDL statements can be executed (and canceled) in transactions.
  - Transactions must be started with `BEGIN WORK`. Statements executed outside of a transaction are automatically committed.
- IBM® DB2®:

- DDL statements can be executed (and canceled) in transactions.
- Beginning of transactions are implicit; two transactions are delimited by `COMMIT` or `ROLLBACK`.

Transactions in stored procedures:

Avoid using transactions in stored procedures to allow the client applications to handle transactions, in accordance with the transaction model.

Savepoints:

- Informix® version 11.50 introduces **savepoints** with the following instructions:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] ]
RELEASE SAVEPOINT name
```

- IBM® DB2® supports **savepoints** too. However, there are differences:

1. Savepoints must be declared with the `ON ROLLBACK RETAIN CURSORS` clause
2. Rollback must always specify the savepoint name

Solution

The Informix® behavior is simulated with an autocommit mode in the IBM® DB2® interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program. Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with IBM® DB2®.

> **Note:** If you want to use savepoints, always specify the savepoint name in `ROLLBACK TO SAVEPOINT`.

See also SELECT FOR UPDATE

Temporary tables

Informix® temporary tables are created through the `CREATE TEMP TABLE` DDL instruction or through a `SELECT ... INTO TEMP` statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

> **Note:** BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

IBM® DB2® 7 supports the DECLARE GLOBAL TEMPORARY TABLE instruction. Native DB2® temporary tables are quite similar to Informix® temporary tables with some exceptions:

- A 'user temporary table space' must exist for the database.
- Users must have 'USE' privilege on a 'user temporary table space'.
- For usage, the temporary table name must be prefixed by 'SESSION'.
- No constraints or indexes can be created on temporary tables.

For more details, see the DB2® documentation.

Solution

In accordance with some prerequisites, temporary tables creation in BDL programs can be supported by the database interface.

How does it work ?

- Informix-specific statements involving temporary table creation are automatically converted to IBM® DB2® "DECLARE GLOBAL TEMPORARY TABLE" statements.
- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to add the SESSION prefix to the table name automatically.

Prerequisites

- Fulfill the DB2® prerequisites to create global temporary tables, at minimum you must create a user temporary table space and grant the usage to database users:

  ```
  CREATE USER TEMPORARY TABLESPACE tempspace01 MANAGED BY AUTOMATIC STORAGE
  GRANT USE OF TABLESPACE tempspace01 TO PUBLIC
  ```

  See DB2® documentation for more details.

Limitations

- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using a temp table name for other database objects, like columns. The following example illustrates this limitation:

  ```
  CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
  SELECT tmp1 FROM table_x WHERE ...
  ```

- Only the 'native' serial emulation mode is supported with temporary tables. See the issue about SERIALs for more details.

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R'   -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

IBM® DB2® provides different functions (SUBSTR, SUSTR2, SUBSTRING), to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTR(col1,2,2) = 'RO'
SELECT SUBSTR('Some text',6,3) ...
SELECT SUBSTRING(col,1,3,CODEUNITS32) ...
```

Solution

You must replace all Informix® `col[x,y]` expressions by `SUBSTRING(col,x,y-x+1,CODEUNITS32)`.

Important:

- In UPDATE instructions, setting column values through subscripts will produce an error with IBM® DB2®:

  `UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...`

  is converted to:

  `UPDATE tab1 SET SUBSTR(col1,2,3-2+1) = 'RO' WHERE ...`

- Column subscripts in ORDER BY expressions produce an error with IBM® DB2®:

  `SELECT ... FROM tab1 ORDER BY col1[1,3]`

is converted to:

```
SELECT ... FROM tab1 ORDER BY SUBSTR(col1,1,3-1+1)
```

String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'string'`

Informix® allows double quotes as string delimiters, but IBM® DB2® doesn't. This is important since many BDL programs use that character to delimit the strings in SQL commands.

This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

Solution

The IBM® DB2® database interface can automatically replace all double quotes by single quotes. However, we recommend that you use only single quotes to enforce portability.

Escaped string delimiters can be used inside strings as in the following:

```
'This is a single quote: '''
'This is a single quote: \''
"This is a double quote: """
"This is a double quote: \""
```

Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string!

For example, if the program executes the SQL statement:

`WHERE "tabname"."colname" = "string"`

replacing all double quotes by single quotes would produce:

`WHERE 'tabname'.'colname' = 'string'`

This would produce an error since '*tabname*'.'*colname*' is not allowed by IBM® DB2®.

Getting one row with SELECT

With Informix®, you must use the system table with a condition on the table id:

```
SELECT user FROM systables WHERE tabid=1
```

With IBM® DB2®, you have to do this:

```
SELECT user FROM SYSIBM.SYSTABLES WHERE NAME='SYSTABLE'
```

Solution

Check the BDL sources for `"FROM systables WHERE tabid=1"` and use dynamic SQL to resolve this problem.

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements, while IBM® DB2® supports the LIKE statement only.

MATCHES requires * and ? wild-card characters, and LIKE uses the % and _ wild-cards was equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES allows you to use brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

The IBM® DB2® LIKE operator has no operator for `[ ]` brackets character ranges.

With IBM® DB2® , columns defined as CHAR(N) are blank padded, and trailing blanks are significant in the LIKE expressions. As result, with a CHAR(5) value such as `'abc '` (with 2 trailing blanks), the expression `(colname LIKE 'ab_')` will not match. To workaround this behavior, you can do `(RTRIM(colname) LIKE '`*`pattern`*`')`. However, consider adding the condition `AND (colname LIKE '`*`patten`*`%')` to force the DB server to optimize the query of the column is indexed. The `CONSTRUCT` instruction uses this technique when the entered criteria does not end with a * star wildcard.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no `[ ]` bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

SQL functions

Both Informix® and DB2® provide numerous built-in SQL functions. Most Informix® SQL functions have the same name and purpose in DB2® (DAY(), MONTH(), YEAR(), UPPER(), LOWER(), LENGTH()).

Table 170: Informix® and IBM® DB2® built-in SQL functions

Informix®	IBM® DB2®
today	current date
current hour to second	current time
current year to fraction(5)	current timestamp
trim([leading | trailing | both "char" FROM] "string")	ltrim() and rtrim()
pow(x,y)	power(x,y)

Solution

You must review the SQL statements using TODAY / CURRENT / EXTEND expressions.

You can create user defined functions (UFs) in the DB2® database.

Querying system catalog tables

As in Informix®, IBM® DB2® provides system catalog tables (systables, syscolumns, etc.) in each database, but the table names and their structures are quite different.

Solution

No automatic conversion of Informix® system tables is provided by the database interface.

The GROUP BY clause

Informix® allows you to use column numbers in the GROUP BY clause

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY 1
```

IBM® DB2® does not support column numbers in the GROUP BY clause.

Solution

Use column names instead:

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY ord_date
```

The star (asterisk) in SELECT statements

Informix® allows you to use the star character in the select list along with other expressions:

```
SELECT col1, * FROM tab1 ...
```

IBM® DB2® does not support this. You must use the table name as a prefix to the star:

```
SELECT col1, tab1.* FROM tab1 ...
```

Solution

Always use the table name with stars.

The LENGTH() function

Informix® provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

IBM® DB2® has a equivalent function with the same name, but there is some difference:

- Informix® does not count the trailing blanks neither for CHAR not for VARCHAR expressions, while IBM® DB2® counts the trailing blanks.
- With the IBM® DB2® LENGTH function, when using a CHAR column, values are always blank padded, so the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

Solution

You must check if the trailing blanks are significant when using the LENGTH()function.

If you want to count the number of characters by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

BDL programming

IBM® DB2® related programming topics.

Informix® specific SQL statements in BDL

The BDL compiler supports several Informix-specific SQL statements that have no meaning when using IBM® DB2®:

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution

Review your BDL source and remove all static SQL statements that are Informix-specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® databases with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

IBM® DB2® does not support insert cursors.

Solution

Insert cursors are emulated by the IBM® DB2® database interface.

Cursors WITH HOLD

Informix® provides the WITH HOLD option to prevent cursors being closed when a transaction ends.

This feature is well supported when using the DB2® interface, except when a transaction is canceled with a ROLLBACK, because DB2® automatically closes all cursors when you rollback a transaction.

Solution

Check that your source code does not use WITH HOLD cursors after transactions canceled with ROLLBACK.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to prevent several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

In both Informix® and DB2®, locks are released when closing the cursor or when the transaction ends; DB2's locking granularity is at the row level.

To control the behavior of the program when locking rows:

- Informix® provides a specific instruction to set the wait mode:

  ```
  SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
  ```

  The default mode is NOT WAIT. This as an Informix-specific SQL statement.
- DB2® has no equivalent for "SET LOCK MODE TO NOT WAIT". The "**Lock timeout**" can be changed but this is a database parameter (global to all processes)!

Solution

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by IBM® DB2®.

SQL parameters limitation

The IBM® DB2® SQL parser does not allow some uses of the '?' SQL parameter marker.

The following SQL expressions are not supported:

```
? IS [NOT] NULL
? operator ?
function( ? )
```

SQL instructions containing these expressions raise an error during the statement preparation.

Solution

Check that your BDL programs do not use these types of conditional expressions.

If you really need to test a BDL variable during the execution of a SQL statement, you must use the CAST() function for DB2® only:

```
WHERE CAST( ? AS INTEGER ) IS NULL
```

See the DB2® documentation for more details.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into an text file, and the LOAD instruction inserts rows from an text file into a database table.

IBM® DB2® does not provide LOAD and UNLOAD instructions.

Solution

LOAD and UNLOAD instructions are supported.

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

DB2® UDB 9 supports SQL Interruption in a similar way as Informix®. The db client must issue an SQLCancel() ODBC call to interrupt a query.

Solution

The DB2® database driver supports SQL interruption and converts the native SQL error code -952 to the Informix® error code -213.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

DB2® UDB supports native scrollable cursors.

Solution

The DB2® database driver uses the native DB2® scrollable cursors by setting the CLI statement attribute SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC.

Connecting to DB2® OS/400®

> **Note:** Some of the following actions can be taken via the OS/400® Operations Navigator.

DB2® Architecture on OS/400®

On OS/400® machines, the DB2 Universal Database™ is integrated to the operating system. Therefore, some concepts change. For example, the physical organization of the database is quite different from UNIX™ or Windows™ platforms.

Table 171: Common terms for the physical organization of the database with DB2® OS/400®

SQL Terms	DB2® OS/400® Terms
Table	Physical file
Row	Record
Column	Field
Index	Keyed logical file, access path
View	Non keyed logical file
Schema	Library, Collection, Schema (OS/400® V5R1 only)
Log	Journal
Isolation Level	Commitment control level

A Collection is a library containing a Journal, Journal Receivers, Views on the database catalogs.

Login to the AS/400® server

First, login to the AS/400® machine with a 5250 display emulation. All the commands are executed in the 5250 display emulation (or telnet connection).

Collection (Schema) Creation

A collection or library in DB2® for OS/400® is equivalent to a schema in DB2® for UNIX™.

1. Launch "Interactive SQL"

   ```
   STRSQL COMMIT(*NONE)
   ```
2. Create a Collection

   CREATE COLLECTION

   Press F4

   Enter field values:

   LIBRARY: name of the collection (Schema)

   ASP: 1

   WITH DATA DICTIONARY: Y

   Press ENTER

   Press F3 to quit (choose Option 1 (save and exit)).

   > **Note:** The name of the Schema should not begin with “Q”; libraries beginning with “Q” are system libraries.

This procedure creates:

- A library for your new database,
- A catalog with a data dictionary,
- A journal (QSQJRN),
- A journal receiver (QSQJRN0001).

Source Physical File Creation

Each table in the database is stored in a Physical file. They can be created in the control center with SQL scripts (CREATE TABLE), or with OS/400® commands.

The table creation script file must be copied in the library in the form: *library/sourcefile.member*

Creation of a physical file:

Type:

CRTSRCPF

Enter field values:

FILE = name of the table (10 characters max).

LIBRARY = name of the library in which the table is created (schema).

RECORD LENGTH = length of the script creation file (in bytes)

MEMBER = *FILE

Execution of the SQL creation script:

Type

RUNSQLSTM

Press F10 for additional parameters

Enter field values:

SOURCE FILE = name of the source file of the script creation file

LIBRARY = name of the library (schema)

SOURCE MEMBER = name of the member of the script creation file

NAMING FIELD = *SQL (SQL Naming convention library.table)

COMMITMENT CONTROL = *NONE

IBM® SQL FLAGGING FIELD = *FLAG

If errors occur, you can use WRKSPLF to display error information saved in the spool file. Use option 5 in the Opt Field on the line of the script file you tried to execute.

Trigger Creation

With DB2® on OS/400®, triggers need to be external programs written in a high level language such as C, COBOL, RPG, or PL/I.

To create a trigger, use the following steps:

1. Create an OS/400® Source file for the trigger programs

   Create a source physical file on your AS/400® for the trigger programs. Each trigger program will be stored in a separate member within this source file.

   Type:

   CRTSRCPF FILE(*library*/*file*)

   where:

   - *library*: name of the library you created for your new database
   - *file*: name you want to call the trigger source physical file

   The file name should be ten characters or fewer.
2. Create a member for each trigger program

   Create a source file member for each trigger program. After the creation of trigger programs (in the next step), the programs will be forwarded to these members.

   Type:

   ADDPFM

Enter field values:

FILE = name of the source file you just created

LIBRARY = name of the library you created for your database

MEMBER = name you want to give the trigger source member

Repeat this operation for each trigger.

3. Create trigger programs in an OS/400® supported high level language

   The OS/400-compatible languages include: ILE C/400®, ILE COBOL, ILE RPG, COBOL, PL/I, and RPG.

   The script creation file of the trigger should be send via FTP into *library/sourcefile.member,* where *sourcefile* and member are the values specified in the previous step.

4. Compile the trigger programs

   Once the trigger programs are in AS/400® members, you can compile them. Use whichever compiler is appropriate for the language you used to create the trigger program.

5. Bind the trigger programs

   After you compile the trigger programs, "bind" each compiled program file. Binding will establish a relationship between the program and any tables or views the program specifies.

   Type:

   CRTPGM PGM (*library*/*program*) ACTGRP(*CALLER)

   where:

   library is the name of the library you created for your new database

   program is the name of the compiled trigger program

   Repeat this operation for each trigger.

6. Add the trigger programs to physical files

   The final step for migrating triggers is to add each program to a physical file. This will tie the trigger program to the table that calls it.

   Type:

   ADDPFTRG

   Enter field values:

   PHYSICAL FILE = name of the table you want to attach the trigger to

   PHYSICAL FILE LIBRARY = name of the database library

   TRIGGER TIME = either *BEFORE or *AFTER.

   TRIGGER EVENT = *INSERT, *DELETE, or *UPDATE.

   PROGRAM = name of the compiled program file

   PROGRAM LIBRARY = name of the database library.

   REPLACE TRIGGER = *YES.

   ALLOW REPEATED CHANGES = *YES.

   Note: The trigger program should be in the same library as the database.

The trigger program is now tied to the table specified in the *Physical File* field and will be called each time the database action you specified occurs. The trigger program may be called from interactive SQL, another AS/400® program, or an ODBC insert, delete, update, or procedure call.

Permission Definition

On OS/400®, database security is managed at the operating system level, not at the database level. When you set up permissions for the database, you determine the degree of access (read, add, delete, etc.) individual users, groups, and authorization lists may have. This operation can easily be done via Operation Navigator.

The privileges must include the following system authorities:

- *USE to the Create Physical File (CRTPF) command.
- *EXECUTE and *ADD to the library into which the table is created.
- *OBJOPR and *OBJMGT to the journal.
- *CHANGE to the data dictionary if the library into which the table is created is an SQL collection with a data dictionary.

To define a foreign key, the privileges must include the following on the parent table:

- The REFERENCES privilege or object management authority for the table.
- The REFERENCES privilege on each column of the specified parent key.
- Ownership of the table.

The REFERENCES privilege on a table consists of:

- Being the owner of the table.
- Having the REFERENCES privilege to the table.
- Having the system authorities of either *OBJREF or *OBJMGT to the table.

The REFERENCES privilege on a column consists of:

- Being the owner of the table.
- Having the REFERENCES privilege to the column.
- Having the system authority of *OBJREF to the column or the system authority of *OBJMGT to the table.

To EXECUTE a user-defined function, the privilege consists of:

- Being owner of the user-defined function.
- Having EXECUTE privilege to the user-defined function.
- Having the system authorities of *OBJOPR and *EXECUTE to the user-defined function.

Relational DB Directory Entry Creation

The relational database directory is equivalent to the database directory of the DB2® client. This is necessary to access the database with DRDA® clients (Distributed Relational Database Architecture™) like DB2® client.

Use the WRKRDBDIRE tool to add the entry in the database directory:

- Type

  WRKDBDIRE
- Type Option 1 (add)
- Enter field values:

  ADDRESS = *LOCAL

  TYPE = *IP

Start the DDM server on the OS/400® which listens on the DRDA® 446 port:

- Type STRTCPSVR *DDM

Start the database server:

- Type STRHOSTSVR

- Enter field values:

  SERVER TYPE = *DATABASE

  REQUIRED PROTOCOL: *ANY

The DDM/DRDA server that listens on TCP/IP port 446 handles requests from a DRDA® client (examples are DB2 Connect™ or another AS/400®).

The database server is not needed for DRDA® clients, but it is needed for Client Access.

If a TCP/IP connection is desired, then your AS/400® server cannot have a release prior to V4R2 installed.

To manually configure the connection via the DB2® command line, you will need to enter catalog commands:

```
> db2 catalog tcpip node <node-name> remote <as400-adress> server 446
> db2 catalog db <db-name-alias> at node <node-name> authentication dcs
> db2 catalog dcs db <db-name-alias> as <local-RDB-name-of-AS400>
```

If you catalogue the DB2® UDB for iSeries® server incorrectly, you may get an SQL5048N error message. SQL7008N is another common error in that the DB2® UDB for iSeries® tables being accessed on the server are not being journaled. To correct the SQL7008N error, you need to start journaling your tables or change the isolation level to No Commit.

The proper CCSID value (normally 37 for US English customers) is needed for any tables on the iSeries® accessed via DB2 Connect™. You can view the CCSID value with the DSPFD CL command or Operations Navigator. CCSID values can be changed with the ALTER TABLE statement or CHGPF CL command. Furthermore, to successfully connect, you may need to change one of the following: the CCSID of the job, the CCSID of the user profile used, or the system CCSID value (QCCSID) if it's the default 65535.

DB2® Client Configuration on Windows™

To configure a DB2® client on Windows™ platforms, use the Client Configuration Assistant. This tool is available only under Microsoft™ Windows™. Under UNIX™, you have to use the command line as described in the previous chapter.

1. Source:
   - Select “Manually configure a connection to a database”.
2. Protocol:
   - Select “TCP/IP”.
   - Check “The database physically resides on a host or AS/400® System”.
3. TCP/IP:
   - Host Name: AS/400® system name.
   - Port Number: Port where DDM/DRDA server is listening (default: 446).
4. Database:
   - Database name: name defined in the relational database directory entries (with WRKRDBDIRE).
5. ODBC:
   - You can register the database as an ODBC data source. Not needed for DRDA® connection used by ODI.
6. Node Options:
   - Optional, but needed to access the database via the control center.
   - System name: AS/400® system name.
   - Instance name: not used for a connection to AS400 (because only one instance is running on an AS/400®).
   - Operating System: OS/400®.

7. Security Options:
   - Optional.
8. Host or AS400 Options:
   - Optional.

Differences Between DB2® UNIX™ & DB2® OS/400®

Some of the differences between DB2® for UNIX™ / Windows™ and DB2® for OS/400® are:

- There is only one database on a system; you can not create two instances on the same database server. The database is a single system-wide database. The database name used for the connect statement is the name of the system. Schemas (Collections) can be used to manage different logical databases on the same OS/400® machine.
- There is no TABLESPACE concept on DB2® for iSeries®. All the storage is controlled by the database manager and operating system.
- The identity column is not supported (for serial emulation).
- The SET SCHEMA SQL command is not supported.
- NUMERIC data type is defined as zoned decimal on DB2® for iSeries® and packed decimal on other platforms.
- The FLOAT data type does not use the same storage. For portability across platforms, do not use FLOAT(n).
- Not all features of the CREATE FUNCTION statement are supported on each platform (see documentation).
- iSeries® prior to V5R1 requires the statement to be processed by a special schema processor. iSeries® as of V5R1 would require this only if the statement includes other DDL statements.
- OS/400® supports “SET DEFAULT” clause ON DELETE.
- OS/400® supports DROP statement with CASCADE behavior.
- Syntaxes of CREATE, ALTER and RENAME TABLE are different on the two systems.

Naming Conventions

The naming convention defines how database tables are identified.

DB2® OS/400® can use two kinds of naming conventions:

- The ***SQL** naming convention.

  The table has to be qualified with the name of the collection (schema) which must be the same name as the user connected to the database. All tables have to be in the same database.
- The ***SYS** naming convention.

  If a table is unqualified, it will be searched for in the *CURLIB collection. You can change the library list with the ADDLIBLE command. You may create a small CL program attached to the profile that will change the library list on sign on. You can also globally change the user portion of the library list using the QUSRLIBL system variable, but this would affect all users on the system.

SQL adaptation guide for IBM® Netezza® 6.x

Installation (Runtime Configuration)
IBM® Netezza® related installation topics.
Install IBM® Netezza® and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. An IBM® Netezza® appliance (the server) must be available.
2. Install the IBM® Netezza® client software with the IBM® Netezza® ODBC driver on the application server.
3. Create an IBM® Netezza® database with the `nzsql` utility.

You must connect to the "system" database:

```
$ nzsql -h hostname system username password
```

4. Create your database with the following SQL command:

```
CREATE DATABASE mydatabase ...
```

5. Create a database user dedicated to the administration of the new database and grant privileges:

```
CREATE USER myadmin WITH PASSWORD 'password' ...
GRANT ALL PRIVILEGES on mydatabase TO myadmin
```

6. Create the application tables.

   Convert Informix® data types to Netezza® data types. See Data type conversion table: Informix to Netezza on page 584 for more details.

7. If you plan to use the SERIAL emulation, you must prepare the database.

   See SERIAL data types on page 581 for more details.

Prepare the runtime environment - connecting to the database

1. In order to connect to IBM® Netezza®, you must have the "`dbmntz`" driver in `FGLDIR/dbdrivers`.
2. The IBM® Netezza® client software with ODBC driver is required to connect to a server.

   Check if the ODBC client library (`libnzodbc.*`) is installed on the machine where the BDL programs run.
3. Make sure that the IBM® Netezza® client environment variables are properly set.

   Check for example NZ_DIR (the path to the installation directory), NZ_ODBC_INI_PATH (the path to the ODBC data source file), etc. See IBM® Netezza® documentation for more details.
4. Verify the environment variable defining the search path for Netezza database client shared libraries (libnzodbc.so on UNIX™, ODBC32.DLL on Windows™).

   Table 172: Shared library environment setting for IBM® Netezza®

IBM® Netezza® version	Shared library environment setting
IBM® Netezza® 6 and higher	*UNIX*™: Add $NZ_DIR/lib (for 32 bit) or $NZ_DIR/lib64 (for 64 bit) to LD_LIBRARY_PATH (or its equivalent). *Windows*™: Add %NZ_DIR%\bin to PATH.

5. Check the database client locale settings.

   The database client locale must match the locale used by the runtime system (LC_ALL, LANG).
6. You can test the client environment by trying to connect to the server with the SQL command line tool:

```
$ nzsql -h hostname system username password
```

7. Set up the fglprofile entries for database connections.
   a) Define the Netezza database driver:

```
dbi.database.dbname.driver = "dbmntz"
```

   b) The "`source`" parameter defines the name of the ODBC source.

```
dbi.database.dbname.source = "test1"
```

Database concepts
IBM® Netezza® related database concepts topics.
Database concepts

Like Informix® servers, Netezza® can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

Solution

Create a Netezza® database for each Informix® database.

Data consistency and concurrency

Data consistency involves readers that want to access data currently modified by writers, and *concurrency data access* involves several writers accessing the same data for modification. *Locking granularity* defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In the case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is per page.

Netezza®

Netezza® servers are designed for Data Warehouse applications, not for OLTP applications: Concurrent data access is not the best thing that a Netezza® server can do. There are a bunch of limitations that you must be aware of. You must not expect to be able to migrate an existing OLTP application running against Informix® or Oracle to a Netezza® database server. The purpose of a Netezza-based application is mostly to do queries, with few insert or updates. Typically a Netezza® database is fed with data by using tools such as nzload, not by Genero BDL programs.

Some limitations of Netezza®:

- An application can only execute one cursor (or statement handle) at a time.
- Singular data modification statements (INSERT, UPDATE, DELETE) are much slower than with traditional OLTP database servers. Netezza® is, however, very good when it comes to loading a huge amount of data with special tools like the nzload utility.
- SELECT ... FOR UPDATE is not supported. Regular SELECTs never lock rows.
- Locks can only be set for an entire table with LOCK TABLE.
- A maximum of 31 concurrent INSERT processes are allowed (Netezza® V6), and there must be only INSERTs in a transaction block.

- UPDATE/DELETE statements lock the entire table, but don't prevent SELECTs. Other processes doing UPDATEs/DELETEs will wait until the first session has committed.
- Netezza® (V6) understands the SET TRANSACTION ISOLATION statement, but currently implements only the SERIALIZABLE level.
- There is no way to define the LOCK WAIT mode. With Netezza®, processes always wait for locks to be released.

Solution

Understand that the main difference with Informix® is that Netezza® is not good at concurrent data modification. Note also that readers do not have to wait for writers in Netezza®.

Genero applications should mainly do queries against a Netezza® server. You must review your program logic that modifies data, having in mind that only one process can modify a table at the time. Note however, that if you write short transactions this is not visible to the end users, except that an INSERT / UPDATE / DELETE of a single row takes more time than with another database server.

The SET ISOLATION TO ... Informix® syntax is replaced by SET TRANSACTION ISOLATION LEVEL ... in Netezza®. However, only the REPEATABLE READ level is supported with Netezza®.

The next table shows the isolation level mappings done by the Netezza® database driver:

Table 173: Isolation level mappings done by the Netezza® database driver

SET ISOLATION instruction in program	Native SQL command
SET ISOLATION TO DIRTY READ	**Not supported (SQL Error)**
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	**Not supported (SQL Error)**
SET ISOLATION TO CURSOR STABILITY	**Not supported (SQL Error)**
SET ISOLATION TO REPEATABLE READ	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

> **Important:** Since Netezza® does not support the **lock wait** mode, you must check that your programs do not include a SET LOCK MODE instruction. This instruction will fail with error -6370 if it is executed when connected to Netezza®.

See the Informix® and Netezza® documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Compared to Informix®, Netezza® has some limitations regarding transactions and concurrent data access.

Informix® native mode (non-ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Netezza®:

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.
- A transaction must only contain INSERTs if you want concurrent processes to insert rows at the same time (UPDATEs/DELETEs lock the whole table).
- Only the SERIALIZABLE isolation level is implemented by Netezza®.

  Note: Netezza® cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
SELECT FROM unexisting WHERE key = 123   -- unexisting table = sql error
COMMIT WORK
```

With Informix®, this code will leave the table with one row inside, since the first INSERT statement succeeded. With Netezza®, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction.

Solution

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Netezza®: Informix® transaction handling commands are automatically converted to Netezza® instructions to start, validate or cancel transactions. However, since Netezza® is not designed for OLTP applications, you must review any code doing complex data modifications. See the concurrency topic for more details.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. To get the same behavior in case of error when connected to a different database than Netezza®, you must issue a ROLLBACK to cancel all the SQL statements that succeeded in the transaction, for example with a TRY/CATCH block.

```
TRY
  BEGIN WORK
  ...
  COMMIT WORK
CATCH
  ROLLBACK WORK
END TRY
```

Database users

Until version 11.70.xC2, Informix® database users must be created at the operating system level and be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers. Any database user must have sufficient privileges to connect and use the resources of the database; user rights are defined with the GRANT command.

Netezza® users must be registered in the database with the CREATE USER command, for example:

`CREATE USER` *`name`* `WITH PASSWORD '`*`pswd`*`' IN GROUP ...`

See the Netezza® documentation for more details about user creation and database access/security.

Solution

According to the application logic (is it a multiuser application?), you have to create one or several Netezza® users.

Data dictionary
IBM® Netezza® related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values; Genero BDL implements the BOOLEAN data type in a different way. As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

Netezza® supports the BOOLEAN data type and stores 't' or 'f' values for TRUE and FALSE representation. It is not possible to insert the integer values 1 or 0: Values must be true, false, 't', 'f', '1' or '0'.

Solution

The Netezza® database interface supports the BOOLEAN data type, and converts the BDL BOOLEAN integer values to a CHAR(1) of '1' or '0'.

CHARACTER data types

Informix® supports the following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases.)

Netezza® supports the following character data types:

- CHAR(N) with N <= 64000 characters
- VARCHAR(N) with N <= 64000 characters
- NCHAR(N) with N <= 16000 characters
- NVARCHAR(N) with N <= 16000 characters

Netezza® stores single-byte character data in CHAR/VARCHAR columns, and stores UNICODE (UTF-8 encoded) character strings in NCHAR/NVARCHAR columns. You cannot store UTF-8 strings in CHAR/VARCHAR columns.

NCHAR/NVARCHAR data is always stored in UTF-8. The database character defines the encoding for CHAR and VARCHAR columns and is defined when creating the database with the CREATE DATABASE command; the default is latin9. Note that, at the time of writing these lines, Netezza® V6 does not yet support a different database character set than latin9.

No automatic character set conversion is done by the Netezza® software, this means that the application/client character set must match the database character set.

Solution

If your application uses a single-byte character set (i.e. latin9), you can create tables with the CHAR and VARCHAR types. However, if you want to store UNICODE (UTF-8) strings, you must use the NCHAR/ NVARCHAR types instead when creating tables. In program sources you can use CHAR/VARCHAR; these types can hold single and multibyte character sets, according to the C POSIX locale.

> **Important:** Netezza® (V6 while writing these lines) supports only the latin9 database character set for CHAR / VARCHAR types. Since character set conversion is not supported, you can only implement either latin9 or UTF-8 based applications.

When using a multibyte character set (such as UTF-8), define database columns as NCHAR and NVARCHAR, with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a Netezza® database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in Netezza® database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

See also the section about Localization.

NUMERIC data types

Informix® supports several data types to store numbers:

Table 174: Informix® numeric data types

Informix® data type	Description
SMALLINT	16 bit signed integer
INT / INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
INT8	64 bit signed integer (replaced by BIGINT)
DEC / DECIMAL	Equivalent to DECIMAL(16)
DEC / DECIMAL(p)	Floating-point exact decimal number, with p significant digits
DEC / DECIMAL(p,s)	Fixed-point exact decimal number, with p significant digits as s decimals
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL / SMALLFLOAT	32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)]	64-bit floating point decimal (C double)

Solution

Netezza® supports the following data types to store numbers:

Table 175: Netezza® numeric data types

Netezza® data type	Description
BYTEINT	8-bit value with the range -128 to 127
SMALLINT	16 bit signed integer
INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
NUMERIC(p,s) / DECIMAL(p,s)	Exact decimal number with p significant digits and s decimals (1<=p<=38)
NUMERIC(p) / DECIMAL(p)	Integer with precision p (1<=p<=38)
NUMERIC / DECIMAL	Integer, same as NUMERIC(18,0)
FLOAT(p) with 1 <= p <= 6	16 bit approx floating point (C float)
FLOAT(p) with 7 <= p <= 15	32 bit approx floating point (C double)
REAL	same as FLOAT(6)
DOUBLE PRECISION	same as FLOAT(15)

Important:

There is no Netezza® equivalent for the Informix® DECIMAL(p) floating point decimal (i.e. without a scale). If your application uses such data types, you must review the database schema in order to use Netezza® compatible types.

To workaround the Netezza® limitation, the NTZ database drivers converts DECIMAL(p) types to a DECIMAL(2*p, p), to store all possible numbers that an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 19, since the Netezza® maximum DECIMAL precision is 38(2*19). If the original precision is bigger than 19, a CREATE TABLE statement executed from a Genero program will fail with an SQL error.

DATE and DATETIME data types

Informix® provides two data types to store dates and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

Netezza® provides the following data type to store date and time information:

- DATE = for year, month, day storage.
- TIME = for hour, minute, second, fraction with (6 decimal positions).
- TIME WITH TIME ZONE / TIMETZ = same as TIME, with time zone information.
- TIMESTAMP = for year, month, day, hour, minute, second, fraction (with 6 decimal positions).

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in Informix®, Netezza® can convert quoted strings to date time data. Netezza® accepts different date formats, including ISO date time strings, and you can specify the cast operator (::date, ::time, ::timestamp) after the string literal.

Date arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- In Netezza®, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. Netezza® does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix® specific and have no equivalent in Netezza®.

Using DATE/DATETIME variables in SQL statements

Informix® supports implicit DATE/DATETIME conversions, for example you can use a DATE variable when the target column is a DATETIME. This is not possible with Netezza®: The type of the SQL parameter must match the type of the column in the database table.

Solution

Netezza® has the same DATE data type as Informix® (year, month, day). So you can use Netezza® DATE data type for Informix® DATE columns.

Netezza® TIME data type can be used to store Informix® DATETIME HOUR TO SECOND values. The database interface makes the conversion automatically.

Informix® DATETIME values with any precision from YEAR to FRACTION(5) can be stored in Netezza® TIMESTAMP columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the Netezza® TIMESTAMP value will be "1900-01-01 11:45:00.0".

Note:

- Make sure that you are using the same type for the SQL parameter and the target column, DATE/DATETIME implicit conversion is not supported by Netezza®.

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

Informix® INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

Netezza® implements the INTERVAL data type in a different way than Informix® does.

- Netezza® allows you to specify interval qualifiers (YEAR, MONTH, DAY, ...) but internally it always uses the same base type, storing values of any combination of units. Thus, there is no way to distinguish year-month intervals and day-time intervals with Netezza®.
- The precision of Netezza® intervals includes fraction of seconds with up to 6 significant digits. However, it is not possible to specify the scale of a Netezza® interval as with the Informix® FRACTION(N) qualifier.
- With Netezza®, interval literals must be include the units, as "-923 days 11 hours 22 minutes", while Informix® interval literals have the form INTERVAL(999-99...) qualifier1 TO qualifier2.
- Netezza® normalizes all INTERVAL values to units of seconds, and considers a month to be thirty days for the purpose of interval comparisons. This approximation can lead to inaccuracies.

Solution

The Informix® INTERVAL types of the day-time class can be mapped to the native Netezza® INTERVAL type, for day to second time interval storage.

Since Netezza® does not clearly distinguish year-month interval class, such types are converted to CHAR(50) by the Netezza® driver.

> **Important:** Netezza® (V6 at the time of writing) has several bugs regarding the INTERVAL type; we do not recommend using this type until Netezza® has fixed these problems.

SERIAL data types

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

However, Netezza® does not have a SERIAL data type. Version 6 of the database supports SEQUENCEs, but not triggers. The lack of triggers support makes it impossible to emulate Informix® SERIALs.

Solution

If you are using Informix® SERIALs or BIGSERIALs, you must review the application logic and database schema to replace SERIAL/BIGSERIAL columns with INTEGER/BIGINT columns, and generate the new keys from a SEQUENCE as described in the SQL Programming page.

ROWIDs

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Netezza® implements ROWIDs like Informix®, except that the rowids are stored in a 64 bit integer in Netezza®.

Solution

ROWIDs can be used with Netezza® as with Informix®, as long as you fetch rowid values into a BIGINT variable. But you should avoid ROWID-based code and use primary key constraints instead.

The SQLCA.SQLERRD[6] register cannot be supported, because Netezza® rowids are 64 bit integers (BIGINT) while SQLCA.SQLERRD[6] is a 32 bit integer (INTEGER). Therefore, all references to SQLCA.SQLERRD[6] must be removed because this variable will not contain the ROWID of the last INSERTed or UPDATEd row.

Indexes

Like most database servers, Informix® supports index creation on table columns. Indexes can be used to make the server find rows rapidly:

```
CREATE INDEX cust_ix1 ON customer (cust_name)
```

Netezza® does not support index creation on tables. There is no need for indexes in a Netezza® database because performance is achieved by distributing data rows over several disks. Netezza® tracks min/max values of each column per disk extent to ignore extents which do not contain the values the query is looking for. See Netezza® documentation for more details.

Solution

You must remove all CREATE INDEX instructions from your programs and SQL scripts that create database tables.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

Netezza® (V6) does not support large objects in the database.

Solution

If your application need to store large objects with TEXT and BYTE data types, you cannot use a Netezza® server.

Constraints

Constraint naming syntax

Both Informix® and Netezza® support primary key, unique, foreign key, default and check constraints, but the constraint naming syntax is different. Netezza® expects the "CONSTRAINT" keyword **before** the constraint specification and Informix® expects it **after**.

UNIQUE constraint example

Table 176: UNIQUE constraint example (Informix® vs Netezza®)

Informix®	Netezza®

`CREATE TABLE emp (`
`  ...`
`  emp_code CHAR(10) UNIQUE`
`  CONSTRAINT pk_emp,`
	`CREATE TABLE emp (`
	`  ...`
	`  emp_code CHAR(10)`
	`  CONSTRAINT pk_emp UNIQUE,`
	`  ...`

> **Important:** Netezza® allows you to create tables with the UNIQUE and PRIMARY KEY and FOREIGN KEY syntax, but the constraints are not enforced.

Solution

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint-naming clauses for Netezza®.

Since Netezza® does not enforce constraints, you must test for unique values and foreign key references at the program level.

Triggers

Informix® supports triggers on database tables.

Netezza® does not support triggers.

Solution

Informix® triggers must be re-written in 4GL.

Stored procedures

Informix® supports stored procedures with the SPL language, and with Java/ C as User Defined Routines.

Netezza® supports stored procedures with the NZPLSQL language.

In Netezza® (V6), a stored procedure must always return a value (see the RETURNS clause). The value returned from a stored procedure can be either a simple scalar value, or a result set (REFTABLE). Netezza® has a limited support for stored procedures producing result sets (you must use dynamic SQL in the stored procedure). See the Netezza® documentation for more details.

> **Note:** Netezza® does not support OUTPUT parameters for stored procedures, only one single value or a result set can be returned.

Solution

Informix® stored procedures must be re-written in the Netezza® language, and the call from programs is slightly different from Informix®.

To call a stored procedure returning a simple scalar value, do following:

```
PREPARE s1 FROM "SELECT myproc(?,?,?)"
EXECUTE s1 USING var1, var2, var3 INTO res
```

To call a stored procedure returning a result set:

```
PREPARE s1 FROM "SELECT myproc(?,?,?)"
OPEN s1 USING var1, var2, var3
FETCH s1 INTO record.*
FETCH s1 INTO record.*
...
```

See SQL Programming for more details about executing stored procedures with Netezza.

Name resolution of SQL objects

Informix® uses the following form to identify an SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

With Netezza®, an object name takes the following form:

```
[database.[schema].]identifier
```

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

Data type conversion table: Informix to Netezza

Table 177: Data type conversion table (Informix to Netezza)

Informix® data types	Netezza® data types
CHAR(n)	CHAR(n) or NCHAR(n) if UTF-8
VARCHAR(n[,m])	VARCHAR(n) or NVARCHAR if UTF-8
NCHAR(n)	NCHAR(n) (UTF-8)
NVARCHAR(n[,m])	NVARCHAR(n) (UTF-8)
BOOLEAN	BOOLEAN
SMALLINT	SMALLINT
INT / INTEGER	INTEGER
BIGINT	BIGINT
INT8	BIGINT
SERIAL[(start)]	N/A (see note 1)
BIGSERIAL[(start)]	N/A (see note 1)
SERIAL8[(start)]	N/A (see note 1)
DOUBLE PRECISION / FLOAT[(n)]	DOUBLE
REAL / SMALLFLOAT	REAL
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL(p)	DECIMAL(p*2,p)
NUMERIC / DEC / DECIMAL	DECIMAL(32,16)
MONEY(p,s)	DECIMAL(p,s)
MONEY(p)	DECIMAL(p,2)
MONEY	DECIMAL(16,2)
TEXT	N/A
BYTE	N/A
DATE	DATE
DATETIME HOUR TO SECOND	TIME
DATETIME YEAR TO FRACTION(p)	TIMESTAMP
INTERVAL YEAR[(p)] TO MONTH	CHAR(50)
INTERVAL YEAR[(p)] TO YEAR	CHAR(50)
INTERVAL MONTH[(p)] TO MONTH	INTERVAL
INTERVAL DAY[(p)] TO FRACTION(n)	INTERVAL
INTERVAL DAY[(p)] TO SECOND	INTERVAL
INTERVAL DAY[(p)] TO MINUTE	INTERVAL
INTERVAL DAY[(p)] TO HOUR	INTERVAL

Informix® data types	**Netezza® data types**
INTERVAL DAY[(p)] TO DAY	INTERVAL
INTERVAL HOUR[(p)] TO FRACTION(n)	INTERVAL
INTERVAL HOUR[(p)] TO SECOND	INTERVAL
INTERVAL HOUR[(p)] TO MINUTE	INTERVAL
INTERVAL HOUR[(p)] TO HOUR	INTERVAL
INTERVAL MINUTE[(p)] TO FRACTION(n)	INTERVAL
INTERVAL MINUTE[(p)] TO SECOND	INTERVAL
INTERVAL MINUTE[(p)] TO MINUTE	INTERVAL
INTERVAL SECOND[(p)] TO FRACTION(n)	INTERVAL
INTERVAL SECOND[(p)] TO SECOND	INTERVAL
INTERVAL FRACTION[(p)] TO FRACTION(n)	INTERVAL

Notes:

1. For more details about serial emulation, see SERIAL data types on page 581.

Data manipulation
IBM® Netezza® related data manipulation topics.
Reserved words

Informix® allows the use of SQL language keywords for database object names (tables, columns):

```
CREATE TABLE table ( int INT, date DATE )
```

In Netezza®, SQL object names like table and column names cannot be SQL reserved keywords.

Solution

Table or column names which are Netezza® reserved keywords must be renamed.

See the Netezza® SQL Reference guide for a list of reserved keywords.

Outer joins

In Informix® SQL, outer tables can be defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER (b)
 WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
 WHERE a.key = b.akey
   AND b.key1 = c.bkey1
   AND b.key2 = c.bkey2
```

Netezza® supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                     ON cust.key = order.custno
SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
            ON cust.key = order.custno
```

```
WHERE order.cdate > current date
```

See the Netezza® reference for a complete description of the syntax.

Solution

For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The Netezza® interface can convert most Informix® OUTER specifications to ANSI outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item, and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work: `"FROM OUTER(tab2), tab1"`
2. The outer join in the WHERE part must use the table name as prefix.

   Example: c_fgl_odiagntz_009.dita.

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   Example: `"... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10"`
2. Statements composed of 2 or more SELECT instructions using OUTERs are not supported.

   Example: `"SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"`

Remarks:

1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias: `"OUTER( tab1 alias1)"`
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.

   OUTER join example with table on the left: `"WHERE outertab.col1 = maintab.col2 "`
3. Table names detection is not case-sensitive.

   Example: `"SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2"`
4. Temporary tables are supported in OUTER specifications.

Transactions handling

Compared to Informix®, Netezza® has some limitations regarding transactions and concurrent data access.

Informix® native mode (non-ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Netezza®:

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Statements executed outside of a transaction are automatically committed.

- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.
- A transaction must only contain INSERTs if you want concurrent processes to insert rows at the same time (UPDATEs/DELETEs lock the whole table).
- Only the SERIALIZABLE isolation level is implemented by Netezza®.

  Note: Netezza® cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
SELECT FROM unexisting WHERE key = 123   -- unexisting table = sql error
COMMIT WORK
```

With Informix®, this code will leave the table with one row inside, since the first INSERT statement succeeded. With Netezza®, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction.

Solution

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Netezza®: Informix® transaction handling commands are automatically converted to Netezza® instructions to start, validate or cancel transactions. However, since Netezza® is not designed for OLTP applications, you must review any code doing complex data modifications. See the concurrency topic for more details.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. To get the same behavior in case of error when connected to a different database than Netezza®, you must issue a ROLLBACK to cancel all the SQL statements that succeeded in the transaction, for example with a TRY/CATCH block.

```
TRY
  BEGIN WORK
  ...
  COMMIT WORK
CATCH
  ROLLBACK WORK
END TRY
```

Temporary tables

Informix® temporary tables are created through the CREATE TEMP TABLE DDL instruction or through a SELECT ... INTO TEMP statement. Temporary tables are automatically dropped when the SQL session ends, but they can be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

Netezza® support temporary tables as Informix® does, with a little syntax difference in the SELECT INTO TEMP instruction.

Solution

Temporary tables are well supported with native Netezza® temp tables.

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R'   -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

Netezza® provides the SUBSTRING(... from ... to ...) function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1 from 2 for 2) = 'RO'
SELECT SUBSTRING('Some text' from 6 for 3) ... -- Gives 'tex'
```

Solution

You must replace all Informix® col[x,y] expressions by SUBSTRING(col from x for (y-x+1)).

Note:

- In UPDATE instructions, setting column values through subscripts will produce an error with PostgreSQL:

  `UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...`

  is converted to:

  `UPDATE tab1 SET SUBSTRING(col1 from 2 for (3-2+1)) = 'RO' WHERE ...`
- Column subscripts in ORDER BY expressions are also converted and produce an error with PostgreSQL:

  `SELECT ... FROM tab1 ORDER BY col1[1,3]`

  is converted to:

  `SELECT ... FROM tab1 ORDER BY SUBSTRING(col1 from 1 for(3-1+1))`

The LENGTH() function

In Informix®, the LENGTH() function counts the number of bytes of a string expression by ignoring the trailing blanks.

Netezza® supports LENGTH() and CHARACTER_LENGTH() functions, but these count the number of characters (not bytes), and trailing blanks are significant.

Netezza® returns NULL if the LENGTH() parameter is NULL. Informix® returns zero instead.

Solution

The Netezza® database interface cannot simulate the behavior of the Informix® LENGTH() SQL function. Review the program logic and make sure you do not pass NULL values to the LENGTH() SQL function.

Name resolution of SQL objects

Informix® uses the following form to identify an SQL object:

`[database[@dbservername]:][{owner|"owner"}.]identifier`

With Netezza®, an object name takes the following form:

`[database.[schema].]identifier`

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'string'`

Informix® allows double quotes as string delimiters, but Netezza® doesn't. This is important since many BDL programs use that character to delimit the strings in SQL commands.

> **Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

Solution

The Netezza® database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like following:

```
'This is a single quote: '''
'This is a single quote: \''
"This is a double quote: """
"This is a double quote: \""
```

> **Important:** Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string!

For example, if the program executes the SQL statement:

`WHERE "tabname"."colname" = "string"`

replacing all double quotes by single quotes would produce:

`WHERE 'tabname'.'colname' = 'string'`

This would produce an error since 'tabname'.'colname' is not allowed by Netezza®.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements. Netezza® supports the LIKE statement as in Informix®, plus the ~ operators that are similar but different from the Informix® MATCHES operator.

MATCHES requires * and ? wildcard characters, and LIKE uses the % and _ wildcards was equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES allows brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

The Netezza® LIKE operator has no operator for `[ ]` brackets character ranges.

The Netezza® ~ operator expects regular expressions as follows: `( col ~ 'a.*' )`

With Netezza® , columns defined as CHAR(N) are blank padded, and trailing blanks are significant in the LIKE expressions. As result, with a CHAR(5) value such as `'abc '` (with 2 trailing blanks), the expression

`(colname LIKE 'ab_')` will not match. To workaround this behavior, you can do `(RTRIM(colname) LIKE 'pattern')`. However, consider adding the condition `AND (colname LIKE 'patten%')` to force the DB server to optimize the query of the column is indexed. The `CONSTRUCT` instruction uses this technique when the entered criteria does not end with a * star wildcard.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no `[ ]` bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

Querying system catalog tables

As in Informix®, Netezza® provides system catalog tables (actually, system views). But the table names and their structure are quite different.

Solution

No automatic conversion of Informix® system tables is provided by the database interface.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( col-list ) = ( val-list )
```

or

```
UPDATE table SET table.* = myrecord.*
```

```
UPDATE table SET * = myrecord.*
```

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column = value [,...]
```

BDL programming
IBM® Netezza® related programming topics.
UPDATE limitations in Netezza

Netezza® has some limitations regarding the UPDATE statement:

- Like DELETE, an UPDATE statement locks the entire table.
- It is not possible to UPDATE *distribution columns*:
  - Netezza® database tables get distributed across all of the nodes using the distribution column. You can specify the distribution column(s) when you create the table. See Netezza® documentation for more details.
  - If you try to update a distribution column, you get error 46 "Attempt to UPDATE a distribution column".

Solution

Review the program logic if the UPDATE statements in your programs use distribution columns, and keep in mind that an UPDATE will lock the entire table.

Informix® specific SQL statements in BDL

The BDL compiler supports several Informix-specific SQL statements that have no meaning when using Netezza®.

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution

Review your BDL source and review all SQL statements which are Informix-specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement.

- When this type of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.
- For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

Netezza® does not support insert cursors.

Solution

Insert cursors are emulated by the Netezza® database interface.

Cursors WITH HOLD

Informix® closes opened cursors automatically when a transaction ends, unless the WITH HOLD option is used in the DECLARE instruction.

With Netezza®, cursors can be kept open when a transaction ends. However, cursors declared with a SELECT FOR UPDATE are not supported with Netezza®.

Solution

Since WITH HOLD cursors are usually declared with SELECT FOR UPDATE and because Netezza® does not support SELECT FOR UPDATE, you must review the program logic if you are using cursors declared WITH HOLD.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

In both Informix® and Netezza®, locks are released when closing the cursor or when the transaction ends.

Netezza® does not support SELECT FOR UPDATE statements.

Solution

You must review the program logic if you use SELECT FOR UPDATE statements. Actually Netezza® systems are designed for data warehouse applications, not for OLTP applications. In a DW context, concurrent data access is not required or a priority.

UPDATE/DELETE WHERE CURRENT OF

Informix® allows positioned UPDATEs and DELETEs with the "`WHERE CURRENT OF cursor`" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

Netezza® servers do no support SELECT FOR UPDATE, and does not set locks. Thus, positioned UPDATEs/DELETEs with the WHERE CURRENT OF<cursor> clause cannot be supported with Netezza®.

Solution

You must review the program logic and rewrite all positioned UPDATEs/DELETEs with a WHERE condition based on primary keys or rowids.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

Netezza® does not provide LOAD and UNLOAD instructions, but provides external tools like the nzload utility.

Solution

LOAD and UNLOAD instructions are supported.

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

Netezza® supports SQL Interruption in a way similar to Informix®. However, when the statement is interrupted, Netezza® rolls the transaction back and returns a "Transaction rolled back by user", SQL error number 46.

Solution

The Netezza® database driver supports SQL interruption and converts the native SQL error 46 to the Informix® error code -213.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

Netezza® does support native scrollable cursors.

Solution

The Netezza® database driver emulates scrollable cursors by fetching rows in a temporary file.

See Scrollable cursors on page 422 for more details about scroll cursor emulation.

SQL adaptation guide for SQL SERVER 2005, 2008, 2012, 2014

Installation (Runtime Configuration)
Microsoft™ SQL Server related installation topics.
Install SQL SERVER and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the Microsoft™ SQL SERVER on your computer.

   Important: Make sure that you select the correct collation when installing SQL Server: The default collation will apply to the tempdb database and will also be used for temporary tables, instead of inheriting the collation of the current database. If the default server collation does not match the collation of the current database, you will experience character set conflicts between permanent tables and temporary tables (SQL Server error message 468).
2. Create a SQL SERVER database entity with the SQL SERVER Management Studio.

   In the database properties:

   a) Choose the right code page / collation to get a case-sensitive database; this cannot be changed later.

   Remember collation defines the character set for CHAR/VARCHAR columns, while NCHAR/NVARCHAR columns are always storing UNICODE (UCS-2) characters. Informix® collation order is codeset based for CHAR/VARCHAR/TEXT columns. If you want to get the same sort order with SQL Server, you will need to use a binary collation such as Latin1_General_BIN.

   b) Make sure the "ANSI NULL Default" option is true if you want to have the same default NULL constraint as in Informix® (i.e. a column created without a NULL constraint will allow null values, users must specify NOT NULL to deny nulls).

   c) Make sure the "Quoted Identifiers Enabled" option is false to use database object names without quotes as in Informix®.
3. Create and declare a database user dedicated to your application: the application administrator .
4. If you plan to use SERIAL emulation based on triggers using a registration table, create the SERIALREG table and create the serial triggers for all tables using a SERIAL.

   See SERIAL data types on page 606.
5. Create the application tables.

   Convert Informix® data types to SQL SERVER data types. See Data type conversion table: Informix to SQL Server on page 615. In order to make application tables visible to all users, make sure that the tables are created with the 'dbo' owner.

Prepare the runtime environment - connecting to the database

1. Genero BDL provides several database drivers based on different ODBC clients. This list describes each of them:

   - For Windows™ platforms, use the SNC database driver based on SQL Native Client ODBC driver (SQLNCLI*.DLL) for Microsoft™ SQL SERVER 2005 and higher. Make sure that the dbmsnc* driver matches the SQNCLI*.DLL.

     The SNC driver is supported starting from Genero 2.10.
   - For Unix platforms, Genero supports the FTM driver is based on the FreeTDS ODBC client (www.freetds.org).

     This driver can be used with FreeTDS to connect from a UNIX™ platform to a Windows™ platform running SQL SERVER.

     You need at least FreeTDS version 0.83, recommended version is 0.95 to connect to recent SQL Server versions such as 2014.

     The FTM driver is supported starting from Genero 2.11.
   - For Unix platforms, Genero supports the ESM driver is based on the EasySoft ODBC driver for SQL Server (www.easysoft.com).

     This driver can be used with EasySoft to connect from a UNIX™ platform to a Windows™ platform running SQL SERVER.

     You need at least EasySoft version 1.2.3.

The ESM driver is supported starting from Genero 2.21.

2. Check that the Genero distribution package has installed the SQL SERVER database driver you need (i.e. a "dbmsnc", "dbmftm" or "dbmesm" driver must exist in `FGLDIR/dbdrivers`.
3. An ODBC data source must be configured to allow the BDL program to establish connections to SQL SERVER.

   Make sure you select the correct ODBC driver (SNC = "SQL Native Client", FTM = "FreeTDS", ESM = "EasySoft").

   > **Important:** When using the FTM (FreeTDS) or ESM (EasySoft) database driver, you have to define the ODBCINI and ODBCINST environment variable to point to the odbc.ini and odbcinst.ini files.

4. Install and configure the database client software:

   a) When using the SNC database driver, you must have the "Microsoft™ SQL SERVER Native Client" software installed on the computer running Genero applications.

   Since the SNC driver is using ODBC32.DLL, there is no need to set the PATH environment variable to a specific database client library path.

   The database client **locale** is defined by the regional settings of the application server and must match the locale used by the BDL application. Character set conversion (current code set <=> Wide-Char) is done by the SNC ODI driver according to the LANG environment variable. If the LANG environment variable is not defined, the application character set defaults to the ANSI code page (ACP).

   b) When using the FTM database driver, you must install FreeTDS (www.freetds.org).

   Make sure the FreeTDS environment variables are properly set. Check for example FREETDS (the path to the configuration file). See FreeTDS documentation for more details.

   With the FTM driver, there is no need to install a driver manager like unixODBC: The FTM database driver is linked directly with the `libtdsodbc.so` shared library. Verify the environment variable defining the search path for that database client shared library (LD_LIBRARY_PATH or equivalent).

   You must create the odbc.ini and odbcinst.ini files to defined the data source.

   Do not forget to define the client character set for FreeTDS (`client charset` parameter in freetds.conf or `ClientCharset` parameter in odbc.ini). You may need to link FreeTDS with the libiconv library to support character set conversions.

   > **Important:** You must set the TDS protocol version according to the SQL Server version (2005, 2008, etc), by setting the `tds version` parameter in freetds.conf or `TDS_Version` in `odbc.ini`. For example, when using SQL Server 2005, you must use the TDS protocol version 7.1. For SQL Server version 2008, 2012 and 2014, use `TDS_Version=7.3`.

   See FreeTDS documentation for more details about installation and data source configuration in ODBC files.

   c) When using the ESM database driver, you must install EasySoft ODBC for SQL Server (www.easysoft.com).

   Make sure the EasySoft environment variables are properly set. Check for example EASYSOFT_ROOT (the path to the installation directory). See FreeTDS documentation for more details.

   With the ESM driver, there is no need to install a driver manager like unixODBC: The ESM database driver is linked directly with the `libessqlsrv.so` shared library. Verify the environment variable defining the search path for that database client shared library (LD_LIBRARY_PATH or equivalent)

   You must create the odbc.ini and odbcinst.ini files to defined the data source.

   Do not forget to define the client character set for EasySoft with the `Client_CSet` parameter in odbc.ini. The client character set is an iconv name and must match the locale of your Genero application.

When using CHAR/VARCHAR types in the database and when the database collation is different from the client locale, you must also set the `Server_CSet` parameter to an iconv name corresponding to the database collation. For example, if `Client_CSet=BIG5` and the db collation is `Chinese_Taiwan_Stroke_BIN`, you must set `Server_CSet=BIG5HKSCS`, otherwise invalid data will be returned from the server.

You must also set the following DSN parameters:

```
AnsiNPW=Yes
Mars_Connection=No
QuotedId=No
```

See EasySoft documentation for more details about installation and data source configuration in ODBC files.

5. On Windows™ platforms, BDL programs are executed in a CONSOLE environment, not a GUI environment. CONSOLE and GUI environments may use different code pages on your system. Start the "SQL SERVER Configuration Manager" to setup your client environment and make sure no wrong character conversion occurs. See Microsoft™ SQL SERVER documentation for more details.
6. Set up the fglprofile entries for database connections.
   a) Define the SQL Server database driver according to the database client used:

   ```
   dbi.database.dbname.driver = { "dbmsnc" | "dbmesm" | "dbmftm" }
   ```

   b) The "`source`" parameter defines the name of the ODBC source.

   ```
   dbi.database.dbname.source = "test1"
   ```

   c) With the SNC driver you might consider setting the `snc.widechar` FGLPROFILE parameter to false if your database columns are defined with the CHAR/VARCHAR/TEXT types (by default the driver is prepared to work with the "UNICODE" types NCHAR/NVARCHAR/NTEXT). See CHARACTER data types on page 600 for more details.

   ```
   dbi.database.dbname.snc.widechar = false
   ```

   d) If required, define the serial emulation method to "trigseq", when the INSERT statements use all columns of the table, including the serial column. For more details, see SERIAL data types on page 606.

   ```
   dbi.database.dbname.ifxemul.datatype.serial.emulation = "trigseq"
   ```

Database concepts
Microsoft™ SQL Server related database concepts topics.
Database concepts

As in Informix®, an SQL SERVER engine can manage multiple database entities. When creating a database object like a table, Microsoft™ SQL SERVER allows you to use the same object name in different databases.

Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from Informix® to Microsoft™ SQL SERVER. Most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be reused in an SQL SERVER database.

Storage concepts are quite similar in Informix® and in Microsoft™ SQL SERVER, but the names are different.

These table compares Informix® storage concepts to Microsoft™ SQL SERVER storage concepts:

Table 178: Physical units of storage

Informix®	**Microsoft™ SQL SERVER**
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (= UNIX™ partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage. You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	SQL SERVER uses "**filegroups**", based on Windows NT™ operating system files and therefore define the physical location of data.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and writeto databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	As in Informix®, SQL SERVER stores data in "**pages**" with a size fixed at 2Kb in V6.5 and 8Kb in V7 and later.
An "**extent**" consists of a collection of continuous "pages" that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	An "**extent**" is a specific number of 8 contiguous pages, obtained in a single allocation. Extents are allocated in the filegroup used by the database.

Table 179: Logical units of storage

Informix®	**Microsoft™ SQL SERVER**
A "**table**" is a logical unit of storage that contains rows of data values.	Same concept as Informix®.
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as Informix®. When creating a "**database**", you must specify which "database devices" (V6.5) or "filegroup" (V7) has to be used for physical storage.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, Informix® defaults to the current database dbspace.	Database tables are created in a database based on "database devices" (V6.5) or a "filegroup" (V7), which defines the physical storage.
The total disk space allocated for a table is the "**tblspace**", which includes "pages" allocated for data, indexes, blobs, tracking page usage within table extents.	No equivalent.

Table 180: Other concepts relating to storage

Informix®	**Microsoft™ SQL SERVER**
When initializing an Informix® engine, a "**root dbspace**" is created to store information about all databases, including storage information (chunks used, other dbspaces, etc.).	SQL SERVER uses the "**master**" database to hold system stored procedures, system messages, SQL SERVER logins, current activity information, configuration parameters of other databases.
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing. The "**logical log**" is a set of "**logical-log files**" used to record logical operations during on-line processing. All transaction information is stored in the logical log files if a database has been created with transaction log. Informix® combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.	Each database has its own "**transaction log**" that records all changes to the database. The "transaction log" is based on a "database device" (V6.5) or "filegroup" (V7) which is specified when creating the database. SQL SERVER checks the "transaction logs" for automatic recovery.

Data consistency and concurrency

Data consistency involves readers which want to access data currently modified by writers and *concurrency data access* involves several writers accessing the same data for modification. *Locking granularity* defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®

Informix® uses a locking mechanism to manage data consistency and concurrency. When a process modifies data with UPDATE, INSERT or DELETE, an exclusive lock is set on the affected rows. The lock is held until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set shared locks according to the isolation level . In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the lock wait mode .

Control:

- Isolation level: SET ISOLATION TO ...
- Lock wait mode: SET LOCK MODE TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit locking: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is READ COMMITTED.
- The default lock wait mode is NOT WAIT.
- The default locking granularity is per page.

SQL SERVER

As in Informix®, SQL SERVER uses locks to manage data consistency and concurrency. The database manager sets exclusive locks on the modified rows and shared locks or update locks when data is read, according to the isolation level . The locks are held until the end of the transaction. When multiple

processes want to access the same data, the latest processes must wait until the first finishes its transaction or the lock timeout occurred. The locking strategy of SQL SERVER is row locking with possible promotion to page or table locking. SQL SERVER dynamically determines the appropriate level at which to place locks for each Transact-SQL statement.

Starting with SQL Server 2005, you can enhance concurrency by turning on snapshot isolation level, to make SQL Server use a copy of the row when it is changed by a transaction. To turn this feature on, you must set the database property ALLOW_SNAPSHOT_ISOLATION ON. Setting the READ_COMMITTED_SNAPSHOT ON option allows access to versioned rows under the default READ COMMITTED isolation level (otherwise, snapshot isolation must be specified by every SQL Session).

Control:

- Lock wait mode: SET LOCK_TIMEOUT <milliseconds> (returns error 1222 on time out).
- Isolation level: SET TRANSACTION ISOLATION LEVEL ...
- Locking granularity: Row, Page or Table level (Automatic - See Dynamic Locking).
- Explicit locking: SELECT ... FROM ... WITH (UPDLOCK) (See Locking Hints)

Defaults:

- The default isolation level is READ COMMITTED (readers cannot see uncommitted data).
- The default LOCK_TIMEOUT is -1 (indicates no timeout period, wait forever).

Solution

The SET ISOLATION TO ... in programs is converted to SET TRANSACTION ISOLATION LEVEL ...for SQL Server. The next table shows the isolation level mappings done by the database driver:

Table 181: Isolation level mappings done by the Microsoft™ SQL Server database driver

SET ISOLATION instruction in program	Native SQL command
SET ISOLATION TO DIRTY READ	SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SET ISOLATION TO CURSOR STABILITY	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
SET ISOLATION TO REPEATABLE READ	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

When using `SET LOCK MODE ...` in the programs, it will be converted to a `SET LOCK_TIMEOUT` instruction for SQL SERVER:

Table 182: SET LOCK MODE as handled by the Microsoft™ SQL Server database driver

SET LOCK MODE instruction in program	Native SQL command
SET LOCK MODE TO WAIT	SET LOCK_TIMEOUT -1 (wait forever)
SET LOCK MODE TO WAIT *seconds*	SET LOCK_TIMEOUT *seconds* * 1000 (wait N milliseconds)
SET LOCK MODE TO NOT WAIT	SET LOCK_TIMEOUT 0 (do not wait)

See Informix® and SQL SERVER documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Informix® and Microsoft™ SQL SERVER handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Microsoft™ SQL SERVER supports named and nested transactions:

- Transactions are started with BEGIN TRANSACTION [*name*].
- Transactions are validated with COMMIT TRANSACTION [*name*].
- Transactions are canceled with ROLLBACK TRANSACTION [*name*].
- Savepoints can be placed with SAVE TRANSACTION *name*.
- Transactions can be rolled back to a savepoint with ROLLBACK TRANSACTION TO *name*.
- Savepoints can not be released.
- Statements executed outside of a transaction are automatically committed (autocommit mode). This behavior can be changed with "SET IMPLICIT_TRANSACTION ON".
- DDL statements are not supported in transactions blocks.

Transactions in stored procedures: avoid using transactions in stored procedure to allow the client applications to handle transactions, according to the transaction model.

Solution

Informix® transaction handling commands are automatically converted to Microsoft™ SQL SERVER instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Microsoft™ SQL SERVER.

> **Important:** If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Database users

Until version 11.70.xC2, Informix® database users must be created at the operating system level and be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as most other db servers. Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

Before a user can access an SQL SERVER database, the system administrator (SA) must add the user's **login** to the SQL SERVER Login list and add a **user name** for that database. The user name is a name that is assigned to a login ID for the purpose of allowing that user to access a specified database. Database users are members of a **user group**; the default group is 'public'.

Microsoft™ SQL SERVER offers two authentication modes: The **SQL SERVER authentication mode**, which requires a login name and a password, and the **Windows NT™ authentication mode**, which uses the security mechanisms within Windows NT™ when validating login connections. With this mode, user

do not have to enter a login ID and password - their login information is taken directly from the network connection.

Solution

Both SQL SERVER and Windows NT™ authentication methods can be used to allow BDL program users to connect to Microsoft™ SQL SERVER and access a specific database.

If you don't specify the `USER/USING` clause in the `CONNECT TO` instruction, operating system authentication takes place.

See SQL SERVER documentation for more details on database logins and users.

Setting privileges

Informix® and Microsoft™ SQL SERVER user privileges management are quite similar.

Microsoft™ SQL SERVER provides **user groups** to grant or revoke permissions to more than one user at the same time.

Data dictionary
Microsoft™ SQL Server related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way: As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

SQL SERVER provides the BIT data type to store boolean values.

Solution

The SQL SERVER database interfaces converts BOOLEAN type to BIT columns and stores 1 or 0 values in the column.

CHARACTER data types

Informix® supports following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases)

SQL Server provides the following data types to store character data:

- CHAR(N) with N<= 8000 bytes (single or multibyte charset)
- VARCHAR(N) with N<= 8000 bytes (single or multibyte charset)
- VARCHAR(MAX) with a limit of 2^31-1 bytes (single or multibyte charset)
- NCHAR(N) with N<= 4000 (Unicode/UCS-2) characters
- NVARCHAR(N) with N<= 4000 (Unicode/UCS-2) characters
- NVARCHAR(MAX) with a limit of 2^31-1 bytes (Unicode/UCS-2)

To store large text data (LOBs), Microsoft™ SQL Server version 2005 introduced the VARCHAR(MAX) type as a replacement for the old TEXT type.

The use of NCHAR, NVARCHAR character types is the same as CHAR, VARCHAR, TEXT respectively, except:

- The encoding is UCS-2 (an UTF-16 subset).
- The length N in N[VAR]CHAR(N) defines a number of characters, not bytes.
- Since each character occupies 2 bytes, twice the space is needed to store the same strings as with CHAR/VARCHAR.
- The maximum size of NCHAR and NVARCHAR column is 4000 characters, compared to 8000 chars for CHAR/VARCHAR using a single-byte character set.
- Unicode string literals are specified with a leading N. For example: N'###'
- The LIKE statement behaves differently with CHAR and NCHAR columns when using the N prefix before the search pattern.

Note that SQL Server uses Byte Length Semantics to define the size of CHAR/VARCHAR columns, while NCHAR and NVARCHAR sizes are expressed in character units.

SQL Server defines the character encoding for CHAR and VARCHAR columns with the database collation. The database collation can be specified when creating a new database. Character strings are always stored in the UCS-2 encoding for NCHAR/NVARCHAR columns.

Automatic charset conversion is supported by SQL Server between the client application and the server. The client charset is defined by the Windows™ operating system, in the language settings for non-Unicode applications.

Solution

According to the character set used by your application, you must either use CHAR/VARCHAR or NCHAR/NVARCHAR columns with SQL Server. If the charset is single-byte , you can use CHAR/VARCHAR columns. If the charset set is multibyte or Unicode (i.e. UTF-8), you must use NCHAR/NVARCHAR columns in SQL Server.

See also the section about Localization.

Make sure that the regional language settings for non-Unicode applications corresponds to the locale used by Genero programs.

Check that your database tables does not use CHAR or VARCHAR types with a length exceeding the SQL SERVER limit.

When using a multibyte character set (such as UTF-8), define database columns as NCHAR and NVARCHAR, with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a SQL Server database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a NCHAR(10 (characters)) column a in SQL Server database, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

Using the SNC driver

The SNC driver can work in *char* or in *wide-char* mode. The character size mode can be controlled by the following FGLPROFILE entry:

```
dbi.database.dbname.snc.widechar= { true | false }
```

By default the **SNC** database driver works in Wide Char mode (true).

Using SNC driver in char mode

The *char* mode can be used with applications defining character string columns with CHAR/VARCHAR/TEXT types. It is not mandatory (i.e. the *wide-char* mode could be used), but it appears that SQL Server behaves in different ways when wide-char bindings are used for CHAR/VARCHAR/TEXT columns.

When defining CHAR(*n*)/VARCHAR(*n*) columns in SQL Server, you specify *n* as a number of bytes, therefore you should use byte length semantics (the default) in Genero programs, with FGL_LENGTH_SEMANTICS=BYTE.

Using SNC driver in wide-char mode

NCHAR / NVARCHAR and NTEXT SQL Server column data types must be used to store Unicode data. The *wide-char* mode should be used for applications using these types. And in such case, the runtime system must use a UTF-8 locale, with character length semantics (FGL_LENGTH_SEMANTICS=CHAR).

In *wide-char* mode, all string literals of an SQL statement are automatically changed to get the N prefix. Thus, you don't need to add the N prefix by hand in all of your programs. This solution makes your Genero code portable to other databases.

Using the ESM driver

When using the ESM (EasySoft) database driver, string literals get the N prefix only if the current locale (LANG / LC_ALL) defines a multibyte code set such as .big5 or .utf8. String literals are not touched if the locale uses a single-byte character set.

When using the ESM (EasySoft) database driver, SQL Statements are prepared with SQLPrepare(), by using the current character set. EasySoft takes in charge the conversion from the client charset to UCS-2 before sending the SQL text to the server. ODBC SQL parameters with character string data are bound (SQLBindParameter) with the C type SQL_C_CHAR and with the SQL type SQL_W[VAR]CHAR (=UNICODE) type. As a result, the necessary character set conversion is taken in charge by EasySoft. However, it is critical to declare the correct client character set in EasySoft configuration files. The EasySoft client character set is defined by the "Client_CSet" parameter in odbc.ini.

Using the FTM driver

When using the FTM (FreeTDS) database driver, string literals get the N prefix only if the current locale (LANG / LC_ALL) defines a multibyte code set such as .big5 or .utf8. String literals are not touched if the locale uses a single-byte character set.

With the FTM (FreeTDS) database driver, SQL Statements are prepared with SQLPrepare(), by using the current character set. FreeTDS takes in charge the conversion from the client charset to UCS-2 before sending the SQL text to the server. ODBC SQL parameters with character string data are bound (SQLBindParameter) with the C type SQL_C_CHAR and with the SQL type SQL_W[VAR]CHAR (=UNICODE) or with SQL_[VAR]CHAR, according to the current locale. The SQL_W[VAR]CHAR type is used if the current locale is a multibyte encoding. When using a single-byte encoding, parameters are bound with the SQL_[VAR]CHAR type. As a result, the necessary character set conversion is taken in charge by FreeTDS and is optimized when using a single-byte character set.

> **Important:** It is critical to declare the correct client character set in FreeTDS configuration files.

The FreeTDS client character set is defined with "ClientCharset" parameter in odbc.ini.

NUMERIC data types

Microsoft™ SQL SERVER offers numeric data types which are quite similar to Informix® numeric data types. This table shows general conversion rules for numeric data types:

Table 183: Numeric data types (Informix® vs. Microsoft™ SQL Server)

Informix®	**Microsoft™ SQL SERVER**
SMALLINT	SMALLINT
INTEGER (synonym: INT)	INTEGER (synonym: INT)
BIGINT	BIGINT
INT8	BIGINT
DECIMAL[(p[,s])] (synonyms: DEC, NUMERIC) DECIMAL(p,s) defines a fixed point decimal where p is the total number of significant digits and s the number of digits that fall on the right of the decimal point. DECIMAL(p) defines a floating point decimal where p is the total number of significant digits. The precision p can be from 1 to 32. DECIMAL is treated as DECIMAL(16).	DECIMAL[(p[,s)] (synonyms: DEC, NUMERIC) DECIMAL[(p[,s])] defines a fixed point decimal where p is the total number of significant digits and s the number of digits that fall on the right of the decimal point. The maximum precision is 38. Without any decimal storage specification, the precision defaults to 18 and the scale defaults to zero: • DECIMAL in SQL SERVER = DECIMAL(18,0) in Informix® • DECIMAL(p) in SQL SERVER = DECIMAL(p,0) in Informix®
MONEY[(p[,s])	SQL SERVER provides the MONEY and SMALLMONEY data types, but the currency symbol handling is quite different. Therefore, Informix® MONEY columns should be implemented as DECIMAL columns in SQL SERVER.
SMALLFLOAT (synonyms: REAL)	REAL
FLOAT[(n)] (synonyms: DOUBLE PRECISION) The precision (n) is ignored.	FLOAT(n) (synonyms: DOUBLE PRECISION) Where n must be from 1 to 15.

Solutions

In BDL programs

When creating tables from BDL programs, the database interface automatically converts Informix® numeric data types to corresponding Microsoft™ SQL SERVER data types.

> **Important:** There is no SQL Server equivalent for the Informix® DECIMAL(p) floating point decimal (i.e. without a scale). If your application is using such data types, you must review the database schema in order to use SQL Server compatible types. To workaround the SQL Server limitation, the SQL Server database drivers convert DECIMAL(p) types to a DECIMAL(2*p, p), to store all possible numbers an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 19, since SQL Server maximum DECIMAL precision is 38(2*19). If the original precision is bigger as 19, a CREATE TABLE statement executed from a Genero program will fail with an SQL Server error 2750.

In database creation scripts

- SMALLINT, INTEGER and BIGINT columns do not have to use another data type in SQL SERVER.
- For DECIMALs, check the precision limit. Always use a precision and a scale.

- Convert MONEY columns to DECIMAL(p,s) columns. Always use a precision and a scale.
- Convert SMALLFLOAT columns to REAL columns.
- Since FLOAT precision is ignored in Informix®, convert this data type to FLOAT(15).

DATE and DATETIME data types

Informix® provides two data types to store dates and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

Microsoft™ SQL SERVER provides two data type to store dates:

- DATETIME = for year, month, day, hour, min, second, fraction(3) storage (from January 1, 1753 through December 31, 9999). Values are rounded to increments of .000, .003, or .007 seconds.
- SMALLDATETIME = for year, month, day, hour, minutes storage (from January 1, 1900, through June 6, 2079). Values with 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute.

Starting with Microsoft™ SQL SERVER 2008, following new date data types are available:

- DATE = for year, month, day storage as Informix® DATEs.
- TIME(n) = for hour, minute, second and fraction(7) storage. Here n defines the precision of fractional seconds.
- DATETIME2(n) = for year, month, day, hour, minute, second and fraction(7) storage. Here n defines the precision of fractional seconds.
- DATETIMEOFFSET(n) = for year, month, day, hour, minute, second, fraction(7) and time zone information storage. Here n defines the precision of fractional seconds.

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in Informix®, Microsoft™ SQL SERVER can convert quoted strings to DATETIME data. The CONVERT() SQL function allows you to convert strings to dates.

Date time arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. Microsoft™ SQL SERVER does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix® specific and have no equivalent in Microsoft™ SQL SERVER.
- Microsoft™ SQL SERVER does not allow direct arithmetic operations on datetimes; the date handling SQL functions must be used instead (DATEADD & DATEDIFF).
- The SQL SERVER provides equivalent functions for YEAR(), MONTH() and DAY(). Be careful with the DAY(*n*) function on SQL SERVER because it begins from January 1, 1900 while Informix® begins from December 31, 1899.

Table 184: Select first day example (Informix® vs. Microsoft™ SQL Server)

Informix®	Microsoft™ SQL SERVER

`SELECT day(0), month(0), year(0)`
`  FROM systables WHERE tabid=1;`
`------ ------ ------`
	`SELECT day(0), month(0), year(0)`
	`----------- ----------- -----------`

Informix®	Microsoft™ SQL SERVER
31 12 1899	1 1 1900

- The SQL SERVER equivalent for WEEKDAY() is the DATEPART(dw,<date>) function. The weekday date part depends on the value set by SET DATEFIRST *n*, which sets the first day of the week (1=Monday ... 7=Sunday (default)).
- SQL SERVER uses a different basis for the day of the week. In SQL SERVER, Sunday is day 7 and Monday is day 1 while Informix® defines Sunday as the day 0 (zero) and Monday as 1.

Solution

The SQL SERVER database drivers will automatically map Informix® date/time types to native SQL SERVER type, according the server version. Conversions are described in this table:

Table 185: Date/time mapping between Informix® and Microsoft™ SQL Server

Informix® date/time type	Microsoft™ SQL SERVER date/time type before SQL SERVER 2008	Microsoft™ SQL SERVER date/time type since SQL SERVER 2008
DATE	DATETIME	DATE
DATETIME HOUR TO SECOND	DATETIME (filled with 1900-01-01)	TIME(0)
DATETIME HOUR TO FRACTION(n)	DATETIME (filled with 1900-01-01)	TIME(n)
DATETIME YEAR TO SECOND	DATETIME	DATETIME2(0)
Any other sort of DATETIME type	DATETIME (filled with 1900-01-01)	DATETIME2(n)

With SQL SERVER 2005 and lower, Informix® DATETIME with any precision from YEAR to FRACTION(3) is stored in SQL SERVER DATETIME columns.

For heterogeneous DATETIME types like DATETIME HOUR TO MINUTE, the database interface fills missing date or time parts to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the SQL SERVER datetime value will be "1900-01-01 11:45:00.0".

Important:

- SQL SERVER SMALLDATETIME can store dates from January 1, 1900, through June 6, 2079. Therefore, we do not recommend using this data type.
- With SQL SERVER 2005 and lower, the fractional second part of a SQL SERVER DATETIME has a precision of 3 digits while Informix® has a precision up to 5 digits. Do not try to insert a datetime value in a SQL SERVER DATETIME with a precision more than 3 digits or a conversion error could occur. You can use the MS SUBSTRING() function to truncate the fraction part of the Informix® datetimes or another BDL solution. The fraction part of a SQL SERVER DATETIME is an approximate value. For example, when you insert a datetime value with a fraction of 111, the database actually stores 110. This may cause problems because Informix® DATETIMEs with a fraction part are exact values with a precision up to 5 digits. Starting with SQL SERVER 2008, the DATETIME2 native type will be used. This new type can store fraction of seconds with a precision of 7 digits, so Informix® DATETIME values can be stored without precision lost.
- When migrating to SQL SERVER 2008, you must pay attention if the database has DATETIME columns used to store Informix® DATETIME HOUR TO SECOND or DATETIME HOUR TO FRACTION(n) types: Before version 2008, those types were stored in SQL SERVER DATETIME

columns (filling missing date part with 1900-01-01). The SNC database driver for SQL SERVER 2008 maps now DATETIME HOUR TO SECOND / FRACTION(n) to a TIME data type, which is not compatible with an SQL SERVER DATETIME type. To solve this problem, SQL SERVER DATETIME columns used to store DATETIME HOUR TO SECOND/FRACTION(n) must be converted to TIME columns (ALTER TABLE).

- When fetching a TIME or DATETIME2 with a precision that is greater as 5 (the DATETIME precision limit), the database interface will allocate a buffer of VARCHAR(16) for the TIME and VARCHAR(27) for the DATETIME2 column. As a result, you can fetch such data into a CHAR or VARCHAR variable.
- Review the program logic if you are using the Informix® WEEKDAY() function because SQL SERVER uses a different basis for the days numbers (Monday = 1).
- Use the SQL SERVER's GETDATE() function to get the system current date.

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

Informix's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

SQL SERVER does not provide a data type corresponding to the Informix® INTERVAL data type.

Solution

The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

SERIAL data types

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

Microsoft™ SQL SERVER IDENTITY columns:

- When creating a table, the IDENTITY keyword must be specified after the column data type:

  ```
  CREATE TABLE tab1 ( k integer identity, c char(10) )
  ```

- You can specify a start value and an increment with "identity(start,incr)".

  ```
  CREATE TABLE tab1 ( k integer identity(100,2), ...
  ```

- A new number is automatically created when inserting a new row:

```
INSERT INTO tab1 ( c ) VALUES ( 'aaa' )
```

- To get the last generated number, Microsoft™ SQL SERVER provides following function:

  ```
  SELECT SCOPE_IDENTITY()
  ```

  The @@IDENTITY global T-SQL variable is not recommended, as it is scope-less.
- To put a specific value into a IDENTITY column, the SET command must be used:

```
SET IDENTITY_INSERT tab1 ON
INSERT INTO tab1 ( k, c ) VALUES ( 100, 'aaa' )
SET IDENTITY_INSERT tab1 OFF
```

Informix® SERIALs and MS SQL SERVER IDENTITY columns are quite similar; the main difference is that MS SQL SERVER does not allow you to use the zero value for the identity column when inserting a new row.

Starting with version 2012, Microsoft™ SQL SERVER supports sequences:

```
-- To create a sequence object:
CREATE SEQUENCE myseq START WITH 100 INCREMENT BY 1;

-- To get a new sequence value:
SELECT NEXT VALUE FOR myseq;

-- To find the current sequence value (last generated)
SELECT convert(bigint, current_value) FROM sys.sequences WHERE name =
 'myseq';

-- To reset the sequence with a new start number:
ALTER SEQUENCE myseq START WITH 100;
```

Solution

To emulation Informix® serials with SQL SERVER, you can use three different solutions:

1. Native SQL SERVER IDENTITY columns.
2. Insert triggers based on sequences (requires SQL SERVER 2012 and +).
3. Insert triggers based on the SERIALREG table (for SQL SERVER prior to 2012).

The method used to emulate SERIAL types is defined by the `ifxemul.datatype.serial.emulation` FGLPROFILE parameter:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial.emulation = {"native"|"trigseq"|"regtable"}`

1. `native`: uses IDENTITY columns.
2. `trigseq`: uses insert triggers with SEQUENCEs.
3. `regtable`: uses insert triggers with the SERIALREG table.

The default emulation technique is "`native`".

This entry must be used in conjonction with:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial = {true|false}`

If the `datatype.serial` entry is set to false, the emulation method is ignored.

The native IDENTITY-based solution is faster, but does not allow explicit serial value specification in insert statements; the others solution are slower but allow explicit serial value specification in INSERT statements.

> **Important:** The trigger-based solutions are provided to simplify the conversion from Informix, but are slower as the solution using IDENTITY columns. To get best performances, we strongly recommend that you use native IDENTITY columns instead of triggers.

1. Using the `native` serial emulation

Make sure that the following FGLPROFILE entry is not defined, in order to use the default "native" emulation:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation ...
```

In database creation scripts, all SERIAL[(n)] data types must be converted by hand to INTEGER IDENTITY[(n,1)] data types, while BIGSERIAL[(n)] data types must be converted by hand to BIGINT IDENTITY[(n,1)] data types.

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER IDENTITY[(n,1)]".

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a "SELECT SCOPE_IDENTITY()". However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must retrieve the last generated serial with the SCOPE_IDENTITY() SQL function.

By default (see SET IDENTITY_INSERT), MS SQL SERVER does not allow you to specify the IDENTITY column in INSERT statements; You must convert all INSERT statements to remove the identity column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file (`DEFINE rec LIKE tab.*`) must also be reviewed:

```
INSERT INTO tab VALUES (rec.*) -- will use the serial column
```

must be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

Since 2.10.06, SELECT * FROM table INTO TEMP with original table having an IDENTITY column is supported: The database driver converts the Informix® SELECT INTO TEMP to the following sequence of statements:

1. `SELECT` *`selection-items`* `INTO` *`#table`* `FROM ... WHERE 1=2`
2. `SET IDENTITY_ INSERT` *`#table`* `ON`
3. `INSERT INTO` *`#table`* `(` *`column-list`* `) SELECT` *`original select clauses`*
4. `SET IDENTITY_ INSERT` *`#table`* `OFF`

See also temporary tables.

2. Using the `trigseq` serial emulation (SQL SERVER 2012 and +)

In order to use the serial emulation based on triggers and sequences, make sure that all database users creating tables in program have permissions to create/drop sequences and triggers.

Define the FGLPROFILE entry to enable "trigseq" serial emulation:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = "trigseq"
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, BIGSERIAL must be converted to BIGINT and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command using a sequence.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers. When using BIGSERIAL[(n)], the column is converted to a BIGINT.

Important:

- SQL SERVER does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.
- SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER. Similarly, a BIGSERIAL column becomes BIGINT.
- When a table is dropped, all associated triggers are also dropped.
- INSERT statements using NULL for the SERIAL column will produce a new serial value, instead of using NULL:

  ```
  INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
  ```

  This behavior is mandatory in order to support INSERT statements which do not use the serial column:

  ```
  INSERT INTO tab (col2) VALUES ('data')
  ```

  Check if your application uses tables with a SERIAL column that can contain a NULL value.

3. Using the `regtable` serial emulation (SQL SERVER versions prior to 2012)

Note: This solution is supported for SQL SERVER versions prior to 2012, if your server is a SQL SERVER 2012 or +, consider using the "trigseq" emulation instead.

In order to use the serial emulation based on triggers and the SERIALREG table, make sure that all database users creating tables in program have permissions to create/drop triggers.

Then, prepare the database and create the SERIALREG table as follows:

```
CREATE TABLE serialreg (
    tablename VARCHAR(50) NOT NULL,
    lastserial BIGINT NOT NULL,
    PRIMARY KEY ( tablename )
)
```

The SERIALREG table and columns have to be created with lower case names, since the SQL SERVER database is created with case sensitive names, because triggers are using this table in lower case.

Define the FGLPROFILE entry to enable "regtable" serial emulation:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation = "regtable"
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, BIGSERIAL must be converted to BIGINT and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column.

Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command using the SERIALREG table.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers. When using BIGSERIAL[(n)], the column is converted to a BIGINT.

Important:

- The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.
- SQL SERVER does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.
- SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER. Similarly, a BIGSERIAL column becomes BIGINT.
- When a table is dropped, all associated triggers are also dropped.
- INSERT statements using NULL for the SERIAL column will produce a new serial value, instead of using NULL:

  ```
  INSERT INTO tab (col1,col2) VALUES ( NULL,'data')
  ```

  This behavior is mandatory in order to support INSERT statements which do not use the serial column:

  ```
  INSERT INTO tab (col2) VALUES ('data')
  ```

  Check if your application uses tables with a SERIAL column that can contain a NULL value.

SQL Server UNIQUEIDENTIFIER data type

SQL Server supports a special type named UNIQUEIDENTIFIER, which can be used to store "Globally Unique Identifiers" (GUIDs). UNIQUEIDENTIFIER values can be generated with the NEWID() function. When creating a table, you typically define a UNIQUEIDENTIFIER column with a DEFAULT clause where the value is produced from a NEWID() call:

```
CREATE TABLE mytab ( k INT, id UNIQUEIDENTIFIER DEFAULT NEWID(), c
 VARCHAR(10) )
```

The UNIQUEIDENTIFIER type is based on the BINARY(16) SQL Server type. The Genero language does not have an equivalent type for BINARY(16). However, BINARY values can be represented as hexadecimal strings in CHAR or VARCHAR variables.

A UNIQUEIDENTIFIER value is usually represented as a GUID identifier, with the following hexadecimal format:

```
XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX (where X can be 0-9, A-F )
```

You typically fetch UNIQUEIDENTIFIER data into a CHAR(36) Genero variable. The resulting value will be expressed in hexadecimal string using the format. You can then reuse that value in an SQL statement, but you have to convert the CHAR(36) hexadecimal string value back to a UNIQUEIDENTIFIER value with the CONVERT() SQL Server function, as shown in this example:

```
DEFINE pi CHAR(36)
CREATE TABLE mytab ( k INT, i UNIQUEIDENTIFIER DEFAULT NEWID(), c
 VARCHAR(10) )
INSERT INTO mytab ( k, c ) VALUES ( 1, 'aaa' )
SELECT i INTO pi FROM mytab WHERE k = 1
UPDATE mytab SET c = 'xxx' WHERE i = CONVERT(UNIQUEIDENTIFIER, pi)
```

When extracting a database schema, UNIQUEIDENTIFIER columns can be clearly distinguished from BINARY(N) columns. The fgldbsch tool will produce a CHAR(36) type code in the .sch file for UNIQUEIDENTIFIER columns.

You can also exclude the UNIQUEIDENTIFIER columns from the table definition in the schema file, by using the x character at the appropriate position of the string passed with the -cv data type conversion option of fgldbsch.

SQL Server ROWVERSION data type

SQL Server provides a special type named ROWVERSION, to stamp row modifications. The ROWVERSION data type replaces the old TIMESTAMP column definition. When you define a column with the ROWVERSION, SQL Server will automatically increment the version column when the row is modified. ROWVERSION is just an incrementing number, it does not preserve date or time information. It be used to control concurrent access to the same rows.

The ROWVERSION type is based on the BINARY(8) SQL Server type. The Genero language does not have an equivalent type for BINARY(8). Therefore, you must fetch ROWVERSION data into a CHAR(16) variable. The resulting value will be expressed in hexadecimal. You can then reuse that value in an UPDATE statement to check that the row was not modified by another process, but you have to convert the CHAR(16) hexadecimal value back to a BINARY(8) value with the CONVERT() SQL Server function, as shown in this example:

```
DEFINE pv CHAR(16)
CREATE TABLE mytab ( k INT, v ROWVERSION, c VARCHAR(10) )
INSERT INTO mytab VALUES ( 1, NULL, 'aaa' )
SELECT v INTO pv FROM mytab WHERE k = 1
UPDATE mytab SET c = 'xxx' WHERE k = 1 AND v =CONVERT(BINARY(8), pv, 2)
```

With SQL Server 2005, the CONVERT() function does not properly transform the hexadecimal string to a binary value. Therefore, you should only use ROWVERSION as SQL parameter starting with SQL Server 2008. ROWVERSION values can however be fetched with SQL Server versions prior to 2008, for example if you have to define record variables based on the table schema, including the ROWVERSION column.

Since ROWVERSION is a synonym for BINARY(8), ROWVERSION columns cannot be clearly identified in ODBC. Therefore, the following conversion rule applies when fetching data from the server:

- If the column is defined as BINARY(N), with N<=128, the data will be fetched as a CHAR(N*2), as an hexadecimal string.
- If the column is defined as BINARY(N), with N>128, the data will be fetched as a BYTE, as a regular binary value.

When extracting a database schema, ROWVERSION columns are identified as TIMESTAMP columns and can be clearly distinguished from BINARY(N) columns. The fgldbsch tool will produce a CHAR(16)type code in the .sch file for ROWVERSION or TIMESTAMP columns.

ROWIDs

When creating a table, Informix® automatically adds a "ROWID" integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Microsoft™ SQL SERVER tables have no ROWIDs.

Solution

If the BDL application uses ROWIDs, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported).

However, if your existing Informix® application depends on using ROWID values, you can use the IDENTITY property of the DECIMAL, INT, NUMERIC, SMALLINT, BIGINT, or TINYINT data types, to simulate this functionality.

All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the Microsoft™ SQL SERVER interface.

Case sensitivity

In Informix®, database object names like table and column names are not case sensitive:

```
CREATE TABLE Customer ( Custno INTEGER, ... )
SELECT CustNo FROM cuSTomer ...
```

In SQL SERVER, database object names and character data are case-insensitive by default:

```
CREATE TABLE Customer ( Custno INTEGER, CustName CHAR(20) )
INSERT INTO CUSTOMER VALUES ( 1, 'TECHNOSOFT' )
SELECT CustNo FROM cuSTomer WHERE custname = 'techNOSoft'
```

The installation program of SQL SERVER allows you to customize the **sort order**. The sort order specifies the rules used by SQL SERVER to collate, compare, and present character data. **It also specifies whether SQL SERVER is case-sensitive**.

Genero compilers convert table and column names to lower case. For example, when writing following static SQL statement:

```
SELECT COUNT(*) FROM CUSTOMER WHERE CUSTNAME LIKE 'S%'
```

The SQL text stored in the pcode module will be:

```
SELECT COUNT(*) FROM customer WHERE custname LIKE 'S%'
```

Solution

Select the case-sensitive sort order when installing SQL SERVER to make queries case-sensitive.

Define the database tables and columns in lower case only, because Genero compilers convert them to lower case.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

Microsoft™ SQL SERVER 2005 and higher provides the VARCHAR(MAX), NVARCHAR(MAX) and VARBINARY(MAX) data types to store large object data. The text, ntext and image data types still exist, but are considered as obsolete and will be removed in a future version.

In SQL Server, the VARCHAR(MAX), NVARCHAR(MAX) and VARBINARY(MAX) types have a limit of 2 gigabytes (2^31 -1 actually). The old text, ntext and image types have the same limit.

Solution

In Genero programs connecting to SQL SERVER, the TEXT and BYTE data types of DDL statements such as CREATE TABLE are respectively converted to VARCHAR(MAX) and VARBINARY(MAX) types.

SQL SERVER database drivers make the appropriate bindings to use TEXT and BYTE Genero types as SQL parameters and fetch buffers, and can be used for SQL SERVER text, image or VARCHAR(MAX), NVARCHAR(MAX) and VARBINARY(MAX) columns.

Genero TEXT/BYTE program variables and the SQL SERVER large object types have the same a limit of 2 gigabytes.

> **Note:** When using a stored procedure that has SET/IF statements and produces a result set with LOBs, the LOB columns must appear at the end of the SELECT list. If LOB columns are followed by

other columns with regular types, the fetching rows will fail. Using SET NOCOUNT ON in the stored procedure does not help, because the cursor type is changed from a server cursor to a default result set cursor.

The ALTER TABLE instruction

Informix® and MS SQL SERVER use different implementations of the ALTER TABLE instruction. For example, Informix® allows you to use multiple ADD clauses separated by comma. This is not supported by SQL SERVER:

Informix®:

```
ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

SQL SERVER:

```
ALTER TABLE customer ADD col1 INTEGER, col2 CHAR(20)
```

Solution

No automatic conversion is done by the database interface. There is even no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server-specific syntax for ALTER TABLE.

Constraints

Constraint naming syntax

Both Informix® and Microsoft™ SQL SERVER support primary key, unique, foreign key, default and check constraints. But the constraint naming syntax is different: SQL SERVER expects the "CONSTRAINT" keyword **before** the constraint specification and Informix® expects it **after**.

Table 186: UNIQUE constraint example (Informix® vs. Microsoft™ SQL Server)

Informix®	Microsoft™ SQL SERVER

`CREATE TABLE emp(`
`  ...`
`  emp_code CHAR(10) UNIQUE`
`  [CONSTRAINT pk_emp],`
`  ...`
	`CREATE TABLE emp (`
	`  ...`
	`  emp_code CHAR(10)`
	`  [CONSTRAINT pk_emp] UNIQUE,`
	`  ...`

Important: SQL SERVER does not produce an error when using the Informix® syntax of constraint naming.

The NULL / NOT NULL constraint

Note: Microsoft™ SQL SERVER creates columns as NOT NULL by default, when no NULL constraint is specified (`colname datatype {NULL | NOT NULL}`). A special option is provided to invert this behavior: ANSI_NULL_DFLT_ON. This option can be enabled with the SET command, or in the database options of SQL SERVER Management Studio.

Solutions

Constraint naming syntax

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for Microsoft™ SQL SERVER.

The NULL / NOT NULL constraint

Before using a database, you must check the "ANSI NULL Default" option in the database properties if you want to have the same default NULL constraint as in Informix® databases.

Triggers

Informix® and Microsoft™ SQL SERVER provide triggers with similar features, but the programming languages are totally different.

Microsoft™ SQL SERVER does not support "BEFORE" triggers.

Microsoft™ SQL SERVER does not support row-level triggers.

Solution

Informix® triggers must be converted to Microsoft™ SQL SERVER triggers "by hand".

> **Important:** To ensure that SQL SERVER generates only necessary result sets in triggers, use the SET NOCOUNT ON at the beginning of your triggers. See SQL SERVER documentation for more details about SET NOCOUNT ON.

Stored procedures

Both Informix® and Microsoft™ SQL SERVER support stored procedures, but the programming languages are totally different.

Solution

Informix® stored procedures must be converted to Microsoft™ SQL SERVER "by hand".

See SQL Programming for more details about executing stored procedures with SQL SERVER.

Name resolution of SQL objects

Informix® uses the following form to identify an SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

With Microsoft™ SQL SERVER, an object name takes the following form:

```
[[database.]owner.]identifier
```

Object names are limited to 128 characters in SQL SERVER and cannot start with one of the following characters: @ (local variable) # (temp object).

To support double quotes as string delimiters in SQL SERVER, you can switch **OFF** the database option "Use quoted identifiers" in the database properties panel. But quoted table and column names are not supported when this option is OFF.

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

Check for single or double quoted table or column names in your source and remove them.

Data type conversion table: Informix to SQL Server

Table 187: Data type conversion table (Informix to SQL Server)

Informix® data types	**SQL SERVER data types (<2008)**	**SQL SERVER data types (>=2008)**
CHAR(n)	CHAR(n) (limit = 8000b!)	CHAR(n) (limit = 8000b!)
VARCHAR(n[,m])	VARCHAR(n) (limit = 8000b!)	VARCHAR(n) (limit = 8000b!)
LVARCHAR(n)	VARCHAR(n) (limit = 8000b!)	VARCHAR(n) (limit = 8000b!)
NCHAR(n)	NCHAR(n) (UNICODE, limit = 4000c!)	NCHAR(n) (UNICODE, limit = 4000c!)
NVARCHAR(n[,m])	NVARCHAR(n) (UNICODE, limit = 4000c!)	NVARCHAR(n) (UNICODE, limit = 4000c!)
BOOLEAN	BIT	BIT
SMALLINT	SMALLINT	SMALLINT
INT / INTEGER	INTEGER	INTEGER
BIGINT	BIGINT	BIGINT
INT8	BIGINT	BIGINT
SERIAL[(start)]	INTEGER (see note 1)	INTEGER (see note 1)
BIGSERIAL[(start)]	BIGINT (see note 1)	BIGINT (see note 1)
SERIAL8[(start)]	BIGINT (see note 1)	BIGINT (see note 1)
DOUBLE PRECISION / FLOAT[(n)]	FLOAT(n)	FLOAT(n)
REAL / SMALLFLOAT	REAL	REAL
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL(p) with p<=19	DECIMAL(2*p,p)	DECIMAL(2*p,p)
NUMERIC / DEC / DECIMAL(p) with p>19	N/A	N/A
NUMERIC / DEC / DECIMAL	DECIMAL(32,16)	DECIMAL(32,16)
MONEY(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
MONEY(p)	DECIMAL(p,2)	DECIMAL(p,2)
MONEY	DECIMAL(16,2)	DECIMAL(16,2)
DATE	DATETIME	DATE
DATETIME HOUR TO MINUTE	DATETIME	TIME(0)
DATETIME HOUR TO FRACTION(n)	DATETIME	TIME(n)
DATETIME YEAR TO SECOND	DATETIME	DATETIME2(0)
DATETIME q1 TO q2 (different from above)	DATETIME	DATETIME2(n)

Informix® data types	**SQL SERVER data types (<2008)**	**SQL SERVER data types (>=2008)**
INTERVAL q1 TO q2	CHAR(50)	CHAR(50)
TEXT	VARCHAR(MAX)	VARCHAR(MAX)
BYTE	VARBINARY(MAX)	VARBINARY(MAX)

Notes:

1. For more details about serial emulation, see SERIAL data types on page 606.

Data manipulation
Microsoft™ SQL Server related data manipulation topics.
Reserved words

Microsoft™ Transact-SQL does not allow you to use reserved words as database object names (tables, columns, constraint, indexes, triggers, stored procedures, ...). An example of a common word which is part of SQL SERVER grammar is '**go**' (see the 'Reserved keywords' section in the SQL SERVER Documentation).

Solution

Database objects having a name which is a Transact-SQL reserved word must be renamed.

All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX™ 'grep' or 'awk' tools. Most modifications can be automatically done with UNIX™ tools like 'sed' or 'awk'.

You can use SET QUOTED_IDENTIFIER ON with double-quotes to enforce the use of keywords in the database objects naming, but it is not recommended.

Outer joins

The original OUTER join syntax of Informix® is different from Microsoft™ SQL SERVER outer join syntax:

In Informix® SQL, outer tables can be defined in the **FROM** clause with the **OUTER** keyword:

```
SELECT ... FROM cust, OUTER(order)
  WHERE cust.key = order.custno

SELECT ... FROM cust, OUTER(order,OUTER(item))
  WHERE cust.key = order.custno
    AND order.key = item.ordno
    AND order.accepted = 1
```

Microsoft™ SQL SERVER supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                     ON cust.key = order.custno

SELECT ...
  FROM cust LEFT OUTER JOIN order
            ON cust.key = order.custno
            LEFT OUTER JOIN item
            ON order.key = item.ordno
 WHERE order.accepted = 1
```

Remark: The old way to define outers in SQL SERVER looks like the following:

```
SELECT ... FROM a, b WHERE a.key *= b.key
```

See the SQL SERVER reference manual for a complete description of the syntax.

Solution

For better SQL portability, you should use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The Microsoft™ SQL SERVER interface can convert simple Informix® OUTER specifications to Microsoft™ SQL SERVER ANSI outer joins.

Prerequisites:

1. The outer join in the WHERE part must use the table name as prefix. Example: "WHERE tab1.col1 = tab2.col2 ".
2. Additional conditions on outer table columns cannot be detected and therefore are not supported: Example: "... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10".
3. Statements composed of 2 or more SELECT instructions using OUTERs are not supported. Example : "SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"

Remarks:

1. Table aliases are detected in OUTER expressions. OUTER example with table alias: "OUTER(tab1 alias1)".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign. OUTER join example with table on the left: "WHERE outertab.col1 = maintab.col2 ".
3. Table names detection is not case-sensitive. Example: "SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2".
4. Temporary tables are supported in OUTER specifications.

Transactions handling

Informix® and Microsoft™ SQL SERVER handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Microsoft™ SQL SERVER supports named and nested transactions:

- Transactions are started with BEGIN TRANSACTION [*name*].
- Transactions are validated with COMMIT TRANSACTION [*name*].
- Transactions are canceled with ROLLBACK TRANSACTION [*name*].
- Savepoints can be placed with SAVE TRANSACTION *name*.
- Transactions can be rolled back to a savepoint with ROLLBACK TRANSACTION TO *name*.
- Savepoints can not be released.
- Statements executed outside of a transaction are automatically committed (autocommit mode). This behavior can be changed with "SET IMPLICIT_TRANSACTION ON".
- DDL statements are not supported in transactions blocks.

Transactions in stored procedures: avoid using transactions in stored procedure to allow the client applications to handle transactions, according to the transaction model.

Solution

Informix® transaction handling commands are automatically converted to Microsoft™ SQL SERVER instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Microsoft™ SQL SERVER.

> **Important:** If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Temporary tables

Informix® temporary tables are created through the **CREATE TEMP TABLE** DDL instruction or through a **SELECT ... INTO TEMP** statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Remark: BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

Microsoft™ SQL SERVER provides local (SQL session wide) or global (database wide) temporary tables by using the '#' or '##' characters as table name prefix. No 'TEMP' keyword is required in CREATE TABLE, and the INTO clause can be used within a SELECT statement to create and fill a temporary table in one step:

```
CREATE TABLE #temp1 ( kcol INTEGER, .... )
SELECT * INTO #temp2 FROM customers WHERE ...
```

Unfortunately, SQL Server temporary tables are created by default with the collation of the tempdb database, instead of inheriting the collation of the current database you are connected to.

Solution

In BDL, Informix® temporary tables instructions are converted to generate native SQL SERVER temporary tables.

Microsoft™ SQL SERVER does not support scroll cursors based on a temporary table.

You must install SQL Server with the same collation as your database, see Installation for more details.

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

Microsoft™ SQL SERVER provides the SUBSTR() function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,2) = 'RO'
SELECT SUBSTRING('Some text',6,3) FROM tab1 -- Gives 'tex'
```

Solution

You must replace all Informix® col[x,y] expressions with SUBSTRING(col,x,y-x+1).

In UPDATE instructions, setting column values through subscripts will produce an error with Microsoft™ SQL SERVER:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
UPDATE tab1 SET SUBSTRING(col1,2,3-2+1) = 'RO' WHERE ...
```

Column subscripts in ORDER BY expressions are also converted and produce an error with Microsoft™ SQL SERVER:

```
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to:

```
SELECT ... FROM tab1 ORDER BY SUBSTRING(col1,1,3-1+1)
```

String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'string'`

Informix® allows double quotes as string delimiters, but SQL SERVER doesn't. This is important, since many BDL programs use that character to delimit the strings in SQL commands.

> **Note:** This problem concerns only double quotes within SQL statements. Double quotes used in BDL string expressions are not subject of SQL compatibility problems.

National character strings:

With SQL SERVER, all UNICODE strings must be prefaced with an N character:

```
UPDATE cust SET cust_name =N'###' WHERE cust_id=123
```

If you don't specify the N prefix, SQL SERVER will convert the characters from the current system locale to the database locale. If the string is prefixed with N, the server can recognize a UNICODE string and use it as is to insert into NCHAR or NVARCHAR columns.

Solution

The SQL SERVER database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: '''
'This is a single quote: \''
"This is a double quote: """
"This is a double quote: \""
```

> **Important:** Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string !

For example, if the program executes the SQL statement:

`WHERE "tabname"."colname" = "string"`

replacing all double quotes by single quotes would produce:

`WHERE 'tabname'.'colname' = 'string'`

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

National character strings

When using the **SNC** database driver, all string literals of an SQL statement are automatically changed to get the N prefix. Thus, you don't need to add the N prefix by hand in all of your programs. This solution makes by the way your Genero code portable to other databases.

With the **SNC** database driver, character string data is converted from the current Genero BDL locale to Wide Char (Unicode UCS-2), before is it used in an ODBC call such as SQLPrepareW or SQLBindParameter(SQL_C_WCHAR). When fetching character data, the **SNC** database driver converts from Wide Char to the current Genero BDL locale. The current Genero BDL locale is defined by LANG, and if LANG is not defined, the default is the ANSI Code Page of the Windows™ operating system. See CHARACTER data types for more details.

When using the **FTM** (FreeTDS) or the **ESM** (EasySoft) database driver on UNIX™, string literals get the N prefix if the current locale is a multibyte encoding like BIG5, EUC-JP or UTF-8. If the current locale is a single-byte encoding like ISO-8859-1, no prefix will be added to the string literals.

Getting one row with SELECT

With Informix®, you must use the system table with a condition on the table id:

```
SELECT user FROM systables WHERE tabid=1
```

With SQL SERVER, you can omit the FROM clause to generate one row only:

```
SELECT user
```

Solution

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements, while Microsoft™ SQL SERVER supports the LIKE statement only.

The MATCHES operator of Informix® uses the star (*), question mark (?) and square braces ([ ]) wildcard characters. The LIKE operator of SQL SERVER offers the percent (%), underscore (_) and square braces ([ ]) wildcard characters:

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x[a-z]' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x[a-z]' )
```

The LIKE operator of SQL Server does not evaluate to true with CHAR/NCHAR columns, if the LIKE pattern is provided as a UNICODE string literal (with the N prefix) and the search pattern matches the value in the column (without an ending % wildcard for example). See the following test:

```
CREATE TABLE mytable ( k INT, nc NCHAR(20) )
INSERT INTO mytable VALUES ( 1, N'abc' )
SELECT * FROM mytable WHERE nc = 'abc' -- one row is returned
SELECT * FROM mytable WHERE nc = N'abc' -- one row is returned
SELECT * FROM mytable WHERE nc LIKE 'abc' -- one row is returned
```

```
SELECT * FROM mytable WHERE nc LIKE N'abc' -- no rows are found
SELECT * FROM mytable WHERE nc LIKE N'abc%' -- one row is returned
```

This can be an issue because the SQL Server driver will by default automatically add an N prefix before all string literals in SQL statements. See Microsoft™ SQL Server documentation for more details about the LIKE semantics regarding blank padding and see also CHARACTER data types for the N prefix usage: You might consider setting the **snc.widechar** FGLPROFILE parameter to false if you are using CHAR/VARCHAR types.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no `[ ]` bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

Pay attention to UNICODE string prefixes N'...' in the LIKE expressions when used with CHAR/NCHAR columns. You might want to always add a % wildcard at the end of the LIKE condition, or use the equal operator when doing a query with exact values.

See also: MATCHES and LIKE operators on page 438.

Querying system catalog tables

As in Informix®, Microsoft™ SQL SERVER provides system catalog tables (sysobjects, syscolumns, etc.) in each database, but the table names and their structure are quite different.

Solution

> **Note:** No automatic conversion of Informix® system tables is provided by the database interface.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

`UPDATE` *`table`* `SET ( col-list ) = ( val-list )`

or

`UPDATE table SET` *`table`*`.* = myrecord.*`

`UPDATE table SET * =` *`myrecord`*`.*`

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

`UPDATE` *`table`* `SET` *`column = value`* `[,...]`

The LENGTH() function

Informix® provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Microsoft™ SQL SERVER has a equivalent function called LEN().

Do not confuse LEN() with DATALEN(), which returns the data size used for storage (number of bytes).

Both Informix® and SQL SERVER ignore trailing blanks when computing the length of a string.

Solution

You must adapt the SQL statements using LENGTH() and use the LEN() function.

Note:

If you create a user function in SQL SERVER as follows:

```
create function length(@s varchar(8000))
 returns integer
as
begin
   return len(@s)
end
```

You must qualify the function with the owner name:

```
SELECT dbo.length(col1) FROM table
```

String concatenation operator

The Informix® concatenation operator is the double pipe (||):

```
SELECT firstname || ' ' || lastname FROM employee
```

The Microsoft™ SQL SERVER concatenation operator is the plus sign:

```
SELECT firstname + ' ' + lastname FROM employee
```

Solution

The database interface detects double-pipe operators in SQL statements and converts them to a plus sign automatically.

BDL programming
Microsoft™ SQL Server related programming topics.
Executing SQL statements

The database driver for Microsoft™ SQL SERVER is based on ODBC. The ODBC driver implementation provided with SQL SERVER uses system stored procedures to prepare and execute SQL statements (You can see this with the Profiler).

Some Transact-SQL statements like SET DATEFORMAT have a local execution context effect (for example, when executed in a stored procedure, it is reset to the previous values when procedure execution is finished).

To support such statements in BDL programs, the database driver uses the SQLExecDirect() ODBC API function when the SQL statement is not a SELECT, INSERT, UPDATE or DELETE. This way the SET statement is executed 'directly', without using the system stored procedures. The result is that the SET statement has the expected effect (i.e. a permanent effect).

However, if the SQL statement uses parameters, the ODBC driver forces the use of system stored procedures to execute the statement.

See the MSDN for more details about system stored procedures used by Microsoft™ APIs.

Informix® specific SQL statements in BDL

The BDL compiler supports several Informix® specific SQL statements that have no meaning when using Microsoft™ SQL SERVER.

Examples:

- CREATE DATABASE dbname **IN dbspace WITH BUFFERED LOG**
- **START DATABASE** (SE only)
- **ROLLFORWARD DATABASE**
- CREATE TABLE ... **IN dbspace WITH LOCK MODE ROW**

Solution

Review your BDL source and remove all static SQL statements that are Informix-specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

Microsoft™ SQL SERVER does not support insert cursors.

Solution

Insert cursors are emulated by the Microsoft™ SQL SERVER database interface.

Cursors WITH HOLD

Informix® automatically closes opened cursors when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction.

Microsoft™ SQL SERVER does not close cursors when a transaction ends. You can change this behavior using the SET CURSOR_CLOSE_ON_COMMIT ON.

Solution

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed by the BDL program.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

- A transaction must be started before opening cursors declared for update.
- The row must be fetched in order to set the lock.
- The lock is released when the transaction ends (if the cursor is not declared "WITH HOLD") or when the cursor is closed.

Microsoft™ SQL SERVER allows individual and exclusive row locking by using the (UPDLOCK) hint after the table names in the FROM clause:

```
SELECT ... FROM tab1 WITH (UPDLOCK) WHERE ...
```

The FOR UPDATE clause is not mandatory; the (UPDLOCK) hint is important.

- Individual locks are acquired when fetching the rows.

- When the cursor (WITH HOLD) is opened outside a transaction, locks are released when the cursor is closed.
- When the cursor is opened inside a transaction, locks are released when the transaction ends.

SQL SERVER's locking granularity is at the row level, page level or table level (the level is automatically selected by the engine for optimization).

To control the behavior of the program when locking rows, Informix® provides a specific instruction to set the wait mode:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This as an Informix® specific SQL statement.

Solution

The SQL SERVER database driver for MS SQL SERVER uses the SCROLL LOCKS concurrency options for cursors (SQL_ATTR_CONCURRENCY = SQL_CONCUR_LOCK).

This option implements pessimistic concurrency control, in which the application attempts to lock the underlying database rows at the time they are read into the cursor result set.

When using server cursors, an update lock is placed on the row when it is read into the cursor. If the cursor is opened within a transaction, the transaction update lock is held until the transaction is either committed or rolled back; the cursor lock is dropped when the next row is fetched.

If the cursor has been opened outside a transaction, the lock is dropped when the next row is fetched.

Therefore, a cursor should be opened in a transaction whenever the user wants full pessimistic concurrency control.

An update lock prevents any other task from acquiring an update or exclusive lock, which prevents any other task from updating the row.

An update lock, however, does not block a shared lock, so it does not prevent other tasks from reading the row unless the second task is also requesting a read with an update lock.

SELECT FOR UPDATE statements are well supported in BDL as long as they are used inside a transaction. Avoid cursors declared WITH HOLD.

> **Note:** The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

The SELECT FOR UPDATE statement cannot contain an ORDER BY clause if you want to perform positioned updates/deletes with WHERE CURRENT OF.

Cursors declared with SELECT ... FOR UPDATE using the "WITH HOLD" clause cannot be supported with SQL SERVER.

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by SQL SERVER.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instruction inserts rows from a text file into a database table.

Microsoft™ SQL SERVER has LOAD and UNLOAD instructions, but those commands are related to database backup and recovery. Do not confuse with Informix® commands.

Solution

LOAD and UNLOAD instructions are supported; note the following;

- The LOAD instruction does not work with tables using emulated SERIAL columns because the generated INSERT statement holds the "SERIAL" column which is actually a IDENTITY column in SQL SERVER. See the limitations of INSERT statements when using SERIALs.
- With Microsoft™ SQL SERVER versions prior to 2008, Informix® DATE data is stored in DATETIME columns, but DATETIME columns are similar to Informix® DATETIME YEAR TO FRACTION(3) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fff". However, since SQL SERVER 2008, Informix® DATE data is stored in SQL SERVER DATE columns, so the result of a LOAD or UNLOAD statement is equivalent when using a DATE column with SQL SERVER 2008.
- With Microsoft™ SQL SERVER versions prior to 2008, Informix® DATETIME data is stored in DATETIME columns, but DATETIME columns are similar to Informix® DATETIME YEAR TO FRACTION(3) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fff". With SQL SERVER 2008, Informix® DATETIME data is stored in SQL SERVER DATETIME2(n<=5) or TIME(n<=5) columns. Concerning DATETIME2(n<=5) columns, the result of LOAD and UNLOAD is equivalent to Informix® DATETIME columns, as long as the original Informix® type starts with the YEAR qualifier. The text data will be "YYYY-MM-DD hh:mm:ss.<*fraction-digits*>", where *fraction-digits* depends on the precision (n) of the DATETIME2(n) column. Concerning TIME(n) columns, the type is converted to an Informix® DATETIME HOUR TO SECOND or FRACTION(n). The text data will be "hh:mm:ss.<*fraction-digits*>", where *fraction-digits* depends on the precision (n) of the TIME(n) column.
- When using an Informix® database, simple dates are unloaded with the DBDATE format (ex:"23/12/1998"). Therefore, unloading from an Informix® database for loading into a Microsoft™ SQL SERVER database is not supported.

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

SQL SERVER 2005 supports SQL Interruption in a similar way as Informix®. The db client must issue an SQLCancel() ODBC call to interrupt a query.

Solution

The SQL Server `SNC` and `ESM` database drivers support SQL interruption and return the Informix® error code **-213** if the statement is interrupted.

> **Important:** Make sure you have SQL SERVER 2005 or higher installed and that you use the **SNC** or **ESM** database driver.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

SQL Server supports native scrollable cursors.

Solution

All the SQL SERVER database drivers use the native SQL Server scrollable cursors by setting the ODBC statement attribute SQL_ATTR_CURSOR_SCROLLABLE to SQL_SCROLLABLE.

SQL adaptation guide for Oracle MySQL 5.x, MariaDB 10.x

> **Note:** Genero programs can connect to Oracle MySQL and it's open source equivalent MariaDB, by using the same database driver (dbmmys).

Installation (Runtime Configuration)
Oracle MySQL related installation topics.
Install MySQL/MariaDB and create a database - database configuration/design tasks

1. Install the MySQL Server (or MariaDB) on your computer.

2. Configure the server with the appropriate storage engine.

   In order to have transaction support by default, you must use a storage engine that supports transactional tables, such as INNODB. In recent versions of MySQL, this is the default storage engine.
3. Consider setting the `sql-mode` configuration parameter to get the appropriate behavior of the MySQL server:
   a) When the STRICT_TRANS_TABLES mode is used, you will get a -1406 error (data too long) when inserting a character string that is too large for the target column.

      If you don't use the STRICT_TRANS_TABLES mode, you get a -1265 warning (data truncated) when the value is too large.
   b) Blank padding of fetched CHAR data can be controlled with the PAD_CHAR_TO_FULL_LENGTH.

      You can use this parameter to get CHAR values padded with blanks to their full length, but the result of the SQL LENGTH() function will be different since trailing blanks are significant for that function in MySQL.
4. The **mysqld** process must be started to listen to database client connections. See MySQL documentation for more details about starting the database server process.
5. Create a database user dedicated to your application, the application administrator.

   Connect as the MySQL root user and GRANT all privileges to this user:

```
$ mysql -u root
...
mysql> grant all privileges on *.*
            to 'myuser'@'localhost'
            identified by 'password'
...
```

6. Connect as the application administrator and create a MySQL database with the CREATE DATABASE statement, and specify the character set to be used for this database:

```
$ mysql -u mysuser
...
mysql> create database mydatabase
             default character set utf8;
```

7. Create the application tables.

   Do not forget to convert Informix® data types to MySQL data types. See Data type conversion table: Informix to MySQL on page 636 for more details.

Prepare the runtime environment - connecting to the database

1. In order to connect to MySQL, you must have a MySQL database driver "`dbmmys`" in `FGLDIR/dbdrivers`.
2. The MySQL client software is required to connect to a database server.

   Check if the MySQL client library (`libmysqlclient.*`) is installed on the system. The shared library version of the MySQL client library must match the libmysqlclient library version linked to the dbmmys.so ODI driver.
3. Make sure that the MySQL client environment variables are properly set.

   Check for example MYSQL_HOME (the path to the installation directory), DATADIR (the path to the data files directory), etc. See MySQL documentation for more details about client environment variables to be set.
4. Check the MySQL client configuration options in the my.cnf file. The driver will read the options defined in the [client] group. Note that your can specify a particular configuration file with the `dbi.database.`*`dbname`*`.mys.optionsfile` FGLPROFILE configuration parameter.
5. Check the database client locale settings (`default-character-set` option in the my.cnf configuration file).

The database client locale must match the locale used by the runtime system (LC_ALL, LANG).

6. Verify the environment variable defining the search path for the database client shared library (libmysqlclient.so on UNIX™, LIBMYSQL.dll on Windows™).

   Table 188: Shared library environment setting for MySQL

MySQL version	Shared library environment setting
MySQL 5.1 and higher	*UNIX™* : Add $MYSQL_HOME/lib to LD_LIBRARY_PATH (or its equivalent). *Windows™ :* Add %MYSQL_HOME%\bin to PATH.

7. To verify if the MySQL client environment is correct, you can start the MySQL command interpreter:

```
$ mysql dbname -u appadmin -p
```

8. Set up the fglprofile entries for database connections.
   a) Define the MySQL database driver:

```
dbi.database.dbname.driver = "dbmmys"
```

   b) The "source" parameter defines the name of the MySQL database.

```
dbi.database.dbname.source = "test1"
```

Database concepts
Oracle MySQL related database concepts topics.
Database concepts

Like Informix® servers, MySQL can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

Solution

Create a MySQL database for each Informix® database.

Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from Informix® to MySQL. Most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be reused for the MySQL database.

Storage concepts are quite similar in Informix® and in MySQL, but the names are different.

Data consistency and concurrency

Data consistency involves readers which want to access data currently modified by writers and *concurrency data access* involves several writers accessing the same data for modification. *Locking granularity* defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same

row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is per page.

MySQL

When data is modified, **exclusive locks** are set and held until the end of the transaction. For data consistency, MySQL uses a **locking mechanism**. Readers must wait for writers as in Informix®.

Control:

- No lock wait mode control is provided.
- Isolation level: SET TRANSACTION ISOLATION LEVEL ...
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed.
- The default locking granularity is per table (pre page when using BDB tables).

Solution

The SET ISOLATION TO ... Informix® syntax is replaced by SET SESSION TRANSACTION ISOLATION LEVEL ... in MySQL. The next table shows the isolation level mappings done by the MySQL database driver:

Table 189: Isolation level mappings done by the MySQL database driver

SET ISOLATION instruction in program	Native SQL command
SET ISOLATION TO DIRTY READ	SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
SET ISOLATION TO CURSOR STABILITY	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED
SET ISOLATION TO REPEATABLE READ	SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ

For portability, it is recommended that you work with Informix® in the read committed isolation level, make processes wait for each other (lock mode wait), and create tables with the "lock mode row" option.

See Informix® and MySQL documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Informix® and MySQL handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

MySQL:

- Transactions are started with START TRANSACTION.
- Transactions are validated with COMMIT [WORK].
- Transactions are canceled with ROLLBACK [WORK].
- Savepoints can be placed with SAVEPOINT *name*.
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO [SAVEPOINT] *name*.
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Solution

Informix® transaction handling commands are automatically converted to MySQL instructions to start, validate or cancel transactions.

MySQL does not support transactions by default. You must set the server system parameter **table_type=InnoDB**.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with MySQL, as long as you have a transaction manager installed with MySQL.

If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Database users

Until version 11.70.xC2, Informix® database users must be created at the operating system level and be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as most other db servers. Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

MySQL users must be registered in the database. They are created with the **GRANT** SQL instruction:

```
$ mysql -u root -pmanager --host orion test
mysql>  GRANT ALL PRIVILEGES ON * TO mike IDENTIFIED BY 'pswd';
```

Solution

According to the application logic (is it a multiuser application?), you have to create one or several MySQL users.

Data dictionary
Oracle MySQL related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way: As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

MySQL supports the BOOLEAN data type and stores 1 or 0 integer values for TRUE and FALSE.

Solution

The MySQL database interface supports the BOOLEAN data type and stores 1 or 0 values in the column..

CHARACTER data types

Informix® supports following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases)

MySQL supports the following character data types:

- CHAR(N) with N<= 255 characters
- VARCHAR(N) with N<= 65535 characters
- NCHAR(N) with N<= 255 characters
- NVARCHAR(N) with N<= 65535 characters
- TEXT (a LOB data type)

With MySQL version 4, CHAR/VARCHAR with a size exceeding 255 characters are silently converted to TEXT columns. With later versions, you now get an SQL error when trying to define a CHAR or VARCHAR column with a size greater than the limit. Also, before version MySQL 5.0.3, VARCHAR limit was 255 characters, starting with 5.0.3 the limit is 65535 characters.

MySQL uses character length semantics to define the size of CHAR/VARCHAR columns, while Informix® and Genero use Byte Length Semantics.

MySQL can support multiple character sets, you can run the SHOW CHARACTER SET statement to list supported encodings. There are different configuration levels to define the character set used by MySQL to store data. The server character set defines the default for database character sets if not specified in the CREATE DATABASE command. You can even define a specific character set at the table and column level, but this is not recommended with Genero applications. The database character set is used to store CHAR and VARCHAR columns. The NCHAR and NATIONAL VARCHAR types use a predefined character set which can be different from the database character set. In MySQL the national character set is UTF-8.

MySQL can automatically convert from/to the client and server characters sets. In the client applications, you define the character set with the SET NAMES instruction.

Note that by default, when fetching CHAR columns from MySQL, trailing blanks are trimmed. This does not matter as long as you fetch CHAR columns into CHAR variables, but this non-standard behavior will

impact CHAR fetch into VARCHAR, or other SQL areas such as string concatenation for example. You can control the behavior of CHAR trailing blanks trimming with the PAD_CHAR_TO_FULL_LENGTH sql-mode parameter. But when this mode is used, the result of the SQL LENGTH() function will be different since trailing blanks are significant for that function in MySQL.

Solution

Informix® CHAR(N) types must be mapped to MySQL CHAR(N) types. Informix® VARCHAR(N) or LVARCHAR(N) columns must be mapped to MySQL VARCHAR(N).

You can store single-byte or multibyte character strings in MySQL CHAR, VARCHAR and TEXT columns.

MySQL uses character length semantics: When you define a CHAR(20) and the database character set is multibyte, the column can hold more bytes/characters than the Informix® CHAR(20) type, when using byte length semantics. When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a MySQL database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in MySQL database using the UTF8 character set, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

Review your database schema when using CHAR columns with a size exceeding the MySQL limits: If you need to store CHAR character strings larger as the MySQL CHAR limit, you can use the MySQL TEXT type. However, as of MySQL version 5.0.3 (supporting large VARCHAR sizes), as long as you use short sizes for CHAR (<100c), the character types can be used as is in MySQL.

The CHAR(N>255) types are converted by the SQL Translator to a MySQL TEXT type, because MySQL CHAR type has a limit of 255 characters. When designing a database, you should consider to use CHAR only for short character string data storage (<50c), and use VARCHAR for larger character string data storage (name, address, comments).

> **Note:** For each TEXT column fetched from MySQL, the MySQL database driver needs to allocate a temporary string buffer of 65535 bytes. The memory used by this temporary buffer is freed when freeing the cursor.

When using VARCHAR types, the SQL Translator leaves the type definition as is, even for N > 255, assuming that the target MySQL server version is at least 5.0.3 (supporting VARCHAR(N) up to 65535 characters).

See also the section about Localization.

NUMERIC data types

Informix® supports several data types to store numbers:

Table 190: Informix® numeric data types

Informix® data type	Description
SMALLINT	16 bit signed integer
INT / INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
INT8	64 bit signed integer (replaced by BIGINT)
DEC / DECIMAL	Equivalent to DECIMAL(16)

Informix® data type	**Description**
DEC(p) / DECIMAL(p)	Floating-point decimal number
DEC(p,s) / DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL / SMALLFLOAT	32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)]	64-bit floating point decimal (C double)

Solution

MySQL supports the following data types to store numbers:

Table 191: MySQL numeric data types

MySQL data type	**Description**
DECIMAL(p)	Stores whole numeric numbers up to p digits (not floating point)
DECIMAL(p,s)	Maximum precision depends on MySQL Version, see documentation.
FLOAT[(M,D)]	4 bytes variable precision
DOUBLE[(M,D)]	8 bytes variable precision
SMALLINT	16 bit signed integer
INTEGER	32 bit signed integer
BIGINT	64 bit signed integer

> **Note:** Before MySQL 5.0.3, the maximum range of DECIMAL values is the same as for DOUBLE. Since MySQL 5.0.3, DECIMAL can store real precision numbers as in Informix®. However, the maximum number of digits depends on the version of MySQL, see documentation for more details. We strongly recommend that you make tests (INSERT+SELECT) to check whether large decimals are properly inserted and fetched back.

DATE and DATETIME data types

Informix® provides two data types to store dates and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

MySQL provides the following data type to store dates:

- DATE = for year, month, day storage.
- TIME[(N)] = for hour, minute, second and fraction of second storage.
- DATETIME[(N)] = for year, month, day, hour, minute, second and fraction of second storage.
- TIMESTAMP = Lilke DATETIME, but can automatically updated when row is touched.

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in Informix®, MySQL can convert quoted strings to datetime data according the ISO datetime format (YYYY-MM-DD hh:mm:ss').

Date arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- In MySQL, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. MySQL does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix® specific and have no equivalent in MySQL.

Solution

MySQL has the same DATE data type as Informix® (year, month, day). So you can use MySQL DATE data type for Informix® DATE columns.

The SQL Translator of the MySQL driver makes the following conversions automatically for the DATETIME types:

- DATETIME HOUR TO MINUTE is converted to MySQL TIME (seconds set to 00).
- DATETIME HOUR TO SECOND is converted to MySQL TIME.
- DATETIME HOUR TO FRACTION(N) is converted to MySQL TIME(N).
- DATETIME YEAR TO MINUTE is converted to MySQL DATETIME (seconds set to 00).
- DATETIME YEAR TO SECOND is converted to MySQL DATETIME.
- DATETIME YEAR TO FRACTION(N) is converted to MySQL DATETIME(N).

Other DATETIME types will be mapped to MySQL DATETIME(N) columns. Missing date or time parts default to 1900-01-01 00:00:00.

> **Important:** MySQL version older than 5.6.4 and MariaDB versions older than 5.3.0 do not support fractional part of DATETIME. If you try to store a DATETIME x TO FRACTION(P) with such old server version, the fractional part is lost.

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

Informix® INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals .

MySQL provides an INTERVAL data type, but it is totally different from the Informix® INTERVAL type. For example, you specify an INTERVAL literal as follows:

```
25 years 2 months 23 days
```

Solution

The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

SERIAL data type

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

MySQL supports the AUTO_INCREMENT column definition option as well as the SERIAL keyword:

- In CREATE TABLE, you specify a auto-incremented column with the AUTO_INCREMENT attribute
- Auto-incremented columns have the same behavior as Informix® SERIAL columns
- You define a start value with ALTER TABLE tabname AUTO_INCREMENT = value
- The column must be the primary key.
- When using the InnoDB engine, AUTO_INCREMENTED columns might reuse unused sequences after a server restart. Actually, when the server restarts, it issues a SELECT MAX(auto_increment_column) on each table with such as column to identify the next sequence to be generated. If you insert rows that generate the numbers 101, 102 and 103, then you delete rows 102 and 103; When the server is restarted next generated number will be 101 + 1 = 102.
- SERIAL is a synonym for BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE.

Solution

The Informix® SERIAL data type is emulated with MySQL AUTO_INCREMENT option. After an insert, SQLCA.SQLERRD[2] holds the last generated serial value. However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT auto incremented columns. If you are using BIGINT auto incremented columns, you must use the LAST_INSERT_ID() SQL function.

AUTO_INCREMENT columns must be primary keys. This is handled automatically when you create a table in a BDL program.

Like Informix®, MySQL allows to specify a zero for auto-incremented columns, however, for SQL portability, INSERT statements should be reviewed to remove the SERIAL column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES ( 0, p_value)
```

can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.* ) -- will use the serial column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

ROWIDs

When creating a table, Informix® automatically adds a "ROWID" integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

MySQL does not have an equivalent for the Informix® ROWID pseudo-column.

Solution

ROWIDs are not supported. You must review the code using ROWIDs and use primary key columns instead.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

MySQL provides TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT, TINYBLOB, BLOB, MEDIUMBLOB and LONGBLOB data types.

Solution

Starting with MySQL version 5.0, the database interface can convert BDL TEXT data to LONGTEXT and BYTE data to LONG BLOB.

Genero TEXT/BYTE program variables have a limit of 2 gigabytes, make sure that the large object data does not exceed this limit.

Because MySQL CHAR and VARCHAR cannot exceed 255 bytes, we recommend that you use the MySQL TEXT type to store CHAR/VARCHAR values with a size larger than 255 bytes. When fetching TEXT columns from a MySQL database, these will be treated as CHAR/VARCHAR types by the MySQL database driver. See CHAR/VARCHAR types for more details.

Constraints

Constraint naming syntax

Both Informix® and MySQL support primary key, unique, foreign key and default, but the constraint naming syntax is different: MySQL expects the "CONSTRAINT" keyword **before** the constraint specification and Informix® expects it **after**.

UNIQUE constraint example

Table 192: UNIQUE constraint example (Informix® vs. MySQL)

Informix®	MySQL

`CREATE TABLE emp(`
`  ...`
`  emp_code CHAR(10) UNIQUE`
`  CONSTRAINT pk_emp,`
	`CREATE TABLE emp (`
	`  ...`
	`  emp_code CHAR(10)`
	`  CONSTRAINT pk_emp UNIQUE,`

Informix®	MySQL
`...`	`...`

Primary keys

Like Informix®, MySQL creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using "CREATE UNIQUE INDEX" to define unique constraints is obsolete (use primary keys or a secondary key instead).

In MySQL, the name of a PRIMARY KEY is PRIMARY.

Unique constraints

Like Informix®, MySQL creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

When using a unique constraint, Informix® allows only one row with a NULL value, while MySQL allows several rows with NULL! Using CREATE UNIQUE INDEX is obsolete.

Foreign keys

Both Informix® and MySQL support the ON DELETE CASCADE option. In MySQL, foreign key constraints are checked immediately, so NO ACTION and RESTRICT are the same.

Check constraints

Check constraints are not yet supported in MySQL.

Solution

Constraint naming syntax

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for MySQL.

Name resolution of SQL objects

Informix® uses the following form to identify a SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

With MySQL, an object name takes the following form:

```
[database.]identifier
```

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

Data type conversion table: Informix to MySQL

Table 193: Data type conversion table (Informix to MySQL)

Informix® data types	MySQL data types
CHAR(n)	CHAR(n) or TEXT (see note 1)
VARCHAR(n[,m])	VARCHAR(n)

Informix® data types	**MySQL data types**
LVARCHAR(n)	VARCHAR(n)
NCHAR(n)	NCHAR(n)
NVARCHAR(n[,m])	NVARCHAR(n)
BOOLEAN	BOOLEAN
SMALLINT	SMALLINT
INT / INTEGER	INTEGER
BIGINT	BIGINT
INT8	BIGINT
SERIAL[(start)]	INTEGER (see note 2)
BIGSERIAL[(start)]	BIGINT (see note 2)
SERIAL8[(start)]	BIGINT (see note 2)
DOUBLE PRECISION / FLOAT[(n)]	DOUBLE
REAL / SMALLFLOAT	FLOAT
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL(p) with p<=15	DECIMAL(p*2,p)
NUMERIC / DEC / DECIMAL(p) with >15	N/A
NUMERIC / DEC / DECIMAL	DECIMAL(32,16) (unsupported!)
MONEY(p,s)	DECIMAL(p,s)
MONEY(p)	DECIMAL(p,2)
MONEY	DECIMAL(16,2)
DATE	DATE
DATETIME HOUR TO MINUTE	TIME
DATETIME HOUR TO SECOND	TIME
DATETIME HOUR TO FRACTION(p)	TIME(p) (see note 3)
DATETIME YEAR TO MINUTE	DATETIME
DATETIME YEAR TO SECOND	DATETIME
DATETIME YEAR TO FRACTION(p)	DATETIME(p) (see note 3)
DATETIME q1 TO q2 (others than above)	DATETIME(p) (see note 3)
INTERVAL q1 TO q2	CHAR(50)
TEXT	MEDIUMTEXT / LONGTEXT (using <= 2Gb!)
BYTE	MEDIUMBLOB / LONGBLOB (using <= 2Gb!)

Notes:

1. The CHAR types with a size > 255 are converted TEXT types. For more details, see CHARACTER data types on page 630.
2. For more details about serial emulation, see SERIAL data type on page 634.

3. Only with MySQL >= 5.6.4 and MariaDB >= 5.3.0, for older versions DATETIME cannot use a frational part.

Data manipulation
Oracle MySQL related data manipulation topics.
Reserved words

SQL object names like table and column names cannot be SQL reserved words in MySQL.

Solution

Table or column names which are MySQL reserved words must be renamed.

Outer joins

In Informix® SQL, outer tables can be defined in the **FROM** clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER(b)
 WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
 WHERE a.key = b.akey    AND b.key1 = c.bkey1
   AND b.key2 = c.bkey2
```

MySQL 3.23 supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                     ON cust.key = order.custno
SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
             ON cust.key = order.custno
 WHERE order.cdate > current date
```

See the MySQL reference for a complete description of the syntax.

Solution

For better SQL portability, you should use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The MySQL interface can convert most Informix® OUTER specifications to ANSI outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must figure from left to right in the order of outer levels.

   Example which does not work: "FROM OUTER(tab2), tab1".
2. The outer join in the WHERE part must use the table name as prefix.

   Example: "WHERE tab1.col1 = tab2.col2".

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   Example: "... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10".
2. Statements composed by 2 or more SELECT instructions using OUTERs are not supported.

   Example: "SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"

   Note:

1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias: "OUTER(tab1 alias1)".
2. In the outer join, <outer table>.<col> can be placed on both right or left side of the equal sign.

   OUTER join example with table on the left: "WHERE outertab.col1 = maintab.col2 ".
3. Table names detection is not case-sensitive.

   Example: "SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2".
4. Temporary tables

   are supported in OUTER specifications.

Transactions handling

Informix® and MySQL handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

MySQL:

- Transactions are started with START TRANSACTION.
- Transactions are validated with COMMIT [WORK].
- Transactions are canceled with ROLLBACK [WORK].
- Savepoints can be placed with SAVEPOINT *name*.
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO [SAVEPOINT] *name*.
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Solution

Informix® transaction handling commands are automatically converted to MySQL instructions to start, validate or cancel transactions.

MySQL does not support transactions by default. You must set the server system parameter **table_type=InnoDB**.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with MySQL, as long as you have a transaction manager installed with MySQL.

If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Temporary tables

Informix® temporary tables are created through the **CREATE TEMP TABLE** DDL instruction or through a **SELECT ... INTO TEMP** statement. Temporary tables are automatically dropped when the SQL session ends, but they can be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

MySQL support temporary tables with the following syntax:

```
CREATE TEMPORARY TABLE tablename ( coldef [,...] )
```

and:

```
CREATE TEMPORARY TABLE tablename LIKE other-table
```

Solution

In BDL, Informix® temporary tables instructions are converted to generate native SQL Server temporary tables.

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3]= 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

MySQL provides the SUBSTRING() function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1,2,3) = 'RO'
SELECT SUBSTRING('Some text',6,3) ... -- Gives 'tex'
```

Solution

You must replace all Informix® col[x,y] expressions by SUBSTRING(col,x,y-x+1).

In UPDATE instructions, setting column values through subscripts will produce an error with MySQL:

```
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
```

is converted to:

```
UPDATE tab1 SET SUBSTRING(col1,2,(3-2+1)) = 'RO' WHERE ...
```

Column subscripts in ORDER BY expressions are also converted and produce an error with MySQL:

```
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

is converted to:

```
SELECT ... FROM tab1 ORDER BY SUBSTRING(col1,1,(3-1+1))
```

Database object name delimiters

Informix® identifies database object names with double quotes, while MySQL does not use the double quotes as database object identifiers.

Solution

Check your programs for database object names having double quotes:

```
WHERE "tabname"."colname" = "string"
```

should be written as follows:

```
WHERE tabname.colname = 'string'
```

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements. MySQL supports the LIKE statement as in Informix®, plus the ~ operators that are similar but different from the Informix® MATCHES operator.

MATCHES requires * and ? wildcard characters, and LIKE uses the % and _ wildcards was equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES allows brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

The MySQL LIKE operator has no operator for [ ] brackets character ranges.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( col-list ) = ( val-list )
```

or

```
UPDATE table SET table.* = myrecord.*
```

```
UPDATE table SET * = myrecord.*
```

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column = value [,...]
```

BDL programming

Oracle MySQL related programming topics.

Informix-specific SQL statements in BDL

The BDL compiler supports several Informix® specific SQL statements that have no meaning when using MySQL:

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution

Review your BDL source and remove all static SQL statements that are Informix-specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

MySQL does not support insert cursors.

Solution

Insert cursors are emulated by the MySQL database interface.

Cursors WITH HOLD

Informix® closes opened cursors automatically when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction. In MySQL, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a transaction ends. Actually, all MySQL cursors are 'WITH HOLD' cursors unless the FOR UPDATE clause is used in the SELECT statement.

Cursors declared FOR UPDATE and using the WITH HOLD option cannot be supported with MySQL because FOR UPDATE cursors are automatically closed by MySQL when the transaction ends.

Solution

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed.

Since MySQL automatically closes FOR UPDATE cursors when the transaction ends, opening cursors declared FOR UPDATE and WITH HOLD option results in an SQL error; in the same conditions, this does not normally appear with Informix®. Review the program logic in order to find another way to set locks.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

MySQL locking mechanism depends upon the transaction manager. The default locking granularity is per table when you use the default non-transactional configuration. You must use the InnoDB Storage Engine to get transactions and locking mechanisms.

SELECT ... FOR UPDATE is only supported since MySQL version 6.0. Locks are released at the end of the transaction.

Solution

Check if the MySQL storage engine supports SELECT FOR UPDATE, otherwise review the program logic.

UPDATE/DELETE WHERE CURRENT OF

Informix® allows positioned UPDATEs and DELETEs with the "`WHERE CURRENT OF` *`cursor`*" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

Solution

WHERE CURRENT OF is not supported by MySQL; review your code for occurrences.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

MySQL does not provide LOAD and UNLOAD instructions.

Solution

LOAD and UNLOAD instructions are supported.

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

MySQL provides the KILL QUERY command to interrupt a running query on the server, but the client program must open a second connection to execyte this statement.

Solution

SQL interruption is supported with MySQL. The database driver opens a second connection to the server and sends a KILL QUERY command, with the MySQL process id of the current connection.

> **Important:** Opening a second connection does not work when using Unix sockets, connect to MySQL with a host name and TCP port.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

MySQL 6.0 does not support native scrollable cursors.

Solution

The MySQL database driver emulates scrollable cursors with temporary files.

See Scrollable cursors on page 422 for more details about scroll cursor emulation.

SQL adaptation guide for Oracle Database 11, 12

Installation (Runtime Configuration)
Oracle Database related installation topics.
Install Oracle and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install the ORACLE database software on your computer.
2. Create and setup the Oracle instance and database. Consider creating a multitenant database when using Oracle 12c and higher, to create several pluggable databases (PDB) in the same Oracle instance.

   Specify the database character set when creating the database instance. If you plan to create a database a multi-byte character set like UTF-8, consider using character length semantics.
3. Create a database context dedicated to your application.

   According the Oracle version, define a db user / schema to hold application tables, or create a pluggable database (starting with Oracle 12c).

   a) With Oracle version 11g and lower, group application tables in a schema by creating a dedicated database user.

Connect as system user with:

```
$ sqlplus / AS SYSDBA
```

and execute the following SQL command to create the db user:

```
CREATE USER appadmin IDENTIFIED BY password;
```

Grant privileges to the application administrator user:

```
GRANT CONNECT, RESOURCE, UNLIMITED TABLESPACE TO appadmin;
```

b) With Oracle version 12c and higher, group application tables in a pluggable database (PDB).

Connect as system user with:

```
$ sqlplus / AS SYSDBA
```

and create a pluggable database and its PDB administrator user. This is a basic PDB creation example using Oracle Managed Files, consider planing the PDB creation with the person in charge of Oracle database administration:

```
CREATE PLUGGABLE DATABASE mypdb
  ADMIN USER pdbadmin IDENTIFIED BY password ROLES = (DBA)
  DEFAULT TABLESPACE mypdb_01
     DATAFILE 'path_01' SIZE 250M AUTOEXTEND ON ;
```

For now the PDB is only mounted, it must be opened for regular usage:

```
ALTER PLUGGABLE DATABASE mypdb OPEN;
```

PDBs must be identified as separate database services (i.e. different from the CDB service). By default Oracle creates a database service with the same name as the PDB. To access the PDB through TNS, create the *mypdb* record in TNSNAMES.ORA file in addition to the default database service (`ORC*`):

```
tnsname =
 (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP)(HOST = myhost)(PORT = 1521))
    (CONNECT_DATA =
       (SERVER = DEDICATED)
       (SERVICE_NAME = mypdb)
    )
 )
```

By default when Oracle starts, the PDBs are mounted but are not open for regular usage. With Oracle 12c, you can create a database trigger to open all PDBs automatically:

```
CREATE OR REPLACE TRIGGER open_pdbs
  AFTER STARTUP ON DATABASE
BEGIN
    EXECUTE IMMEDIATE 'ALTER PLUGGABLE DATABASE ALL OPEN';
END open_pdbs;
/
```

Re-connect as PDB administrator and create a user dedicated to application tables administration:

```
CONNECT pdbadmin/password@localhost/mypdb
CREATE USER appadmin IDENTIFIED BY password;
```

```
GRANT CONNECT, RESOURCE, UNLIMITED TABLESPACE TO appadmin;
```

4. If programs create temporary tables, you must define the type of temporary table emulation to be used.
   a) If you plan to use the default temporary table emulation, setup your database for the TEMPTABS tablespace usage.

      See Using the default temporary table emulation on page 670 for more details.
   b) If you plan to use the temporary table emulation based on Oracle global temporary tables, setup the database to use the TEMPTABS schema/user.

      See Using the global temporary table emulation on page 672 for more details.

   For more details about temporay table emulations, see Temporary tables on page 670.
5. Create the application tables by connecting to the database context as the application administrator:

   ```
   $ sqlplus appadmin/password@tnsname
   ```

   Convert Informix® data types to Oracle data types. See issue data type Conversion Tables for more details.
6. If you plan to use SERIAL emulation, you must choose a serial emulation method.

   Select the best emulation technique that matches your needs. You need to prepare the database according to the emulation type. For more details, see SERIAL data types on page 658.

Prepare the runtime environment - connecting to the database

1. In order to connect to ORACLE, you must have a database driver "dbmora" in FGLDIR/dbdrivers.
2. If you want to connect to a remote Oracle server from an application server, you must install the ORACLE Client Software on your application server and configure this.
3. Make sure that the ORACLE client environment variables are properly set.

   Check variables such as ORACLE_HOME (the path to the installation directory), ORACLE_SID (the server identifier when connecting locally), etc. See the Oracle documentation for more details.
4. Verify the environment variable defining the search path for database client shared libraries (libclntsh.so on UNIX™, OCI.DLL on Windows™)

ORACLE version	Shared library environment setting
Oracle 10g and higher	*UNIX*™: Add $ORACLE_HOME/lib to LD_LIBRARY_PATH (or its equivalent). *Windows*™: Add %ORACLE_HOME%\bin to PATH.

5. Check the database client locale settings (NLS_LANG, NLS_DATE_FORMAT, etc).

   The database client locale must match the locale used by the runtime system (LC_ALL, LANG).
6. If you are using the TNS protocol, verify if the ORACLE listener is started on the server.
7. To test the client environment settings, you can try to connect to the ORACLE server with the SQL*Plus tool:

   ```
   $ sqlplus username/password@service
   ```

8. Set up the fglprofile entries for database connections.
   a) Set up fglprofile for the SERIAL emulation method.

      The following entry defines the SERIAL emulation method. You can use the SEQUENCE based trigger or the SERIALREG based trigger method:

      ```
      dbi.database.dbname.ifxemul.datatype.serial.emulation = "(native|
      regtable)"
      ```

The value 'native' selects the SEQUENCE based method, and the value 'regtable' selects the SERIALREG based method. This entry has no effect if `dbi.database.`*`dbname`*`.ifxemul.datatype.serial` is set to 'false'.

The default is SERIAL emulation enabled with native method (SEQUENCE-based). See issue SERIAL data types on page 658 for more details.

b) The "`source`" parameter defines the TNS name of the Oracle database.

```
dbi.database.dbname.source = "stock"
```

c) Define the database schema selection if needed.

The following entry defines the database schema to be used by the application. The database interface automatically executes an "`ALTER SESSION SET CURRENT_SCHEMA` *`owner`*" instruction to switch to a specific schema:

```
dbi.database.dbname.ora.schema = "name"
```

Here *dbname* identifies the database name used in the BDL program (DATABASE *dbname*) and *name* is the schema name to be used in the ALTER SESSION instruction. If this entry is not defined, no ALTER SESSION instruction is executed and the current schema defaults to the user's name.

d) Define pre-fetch parameters.

Oracle offers high performance by pre-fetching rows in memory. The pre-fetching parameters can be tuned with the following entries:

```
dbi.database.dbname.ora.prefetch.rows = integer
dbi.database.dbname.ora.prefetch.memory = integer # in bytes
```

These values will be applied to all application cursors.

The interface pre-fetches rows up to the prefetch.rows limit unless the prefetch.memory limit is reached, in which case the interface returns as many rows as will fit in a buffer of size prefetch.memory. By default, pre-fetching is on and defaults to 10 rows; the memory parameter is set to zero, so the memory size is not included in computing the number of rows to prefetch.

e) If needed, define a specific command to generate session identifiers with this FGLPROFILE setting:

```
dbi.database.dbname.ora.sid.command = "SELECT ..."
```

This unique session identifier will be used to create table names for temporary table emulation.

By default, the database driver will use "`SELECT USERENV('SESSIONID') FROM DUAL`".

f) If needed, define a specific command to generate session identifiers with this FGLPROFILE setting:

```
dbi.database.dbname.ora.sid.command = "SELECT ..."
```

This unique session identifier will be used to create table names for temporary table emulation.

By default, the database driver will use "`SELECT USERENV('SESSIONID') FROM DUAL`".

g) The default temporary table emulation uses regular permanent tables.

If this does not fit your needs, you can use GLOBAL TEMPORARY TABLES with this FGLPROFILE setting:

```
dbi.database.dbname.ifxemul.temptables.emulation = "global"
```

Database concepts
Oracle Database related database concepts topics.
Database concepts

Informix® servers can handle multiple database entities. By default an ORACLE instance can only handle one database entity. Starting with Oracle 12c, you can use a multi-tenant database to define several pluggable databases.

ORACLE can manage multiple schemas, but by default other users must give the owner name as prefix to the table name:

```
SELECT * FROM stores.customer
```

Solution 1: With Oracle 12c and higher

Oracle 12c introduced the multi-tenant database concept, where you can create several pluggable databases in a root container. Consider using this feature, if you need to create several copies of the same database entity, that can be accessed/seen as individual data sources.

Solution 2: With Oracle 10g and 11g

In an Oracle database, each user can manage his own database schema. You can dedicate a database user to administer each occurrence of the application database.

Any user can select the current database schema with the following SQL command:

```
ALTER SESSION SET CURRENT_SCHEMA = "schema"
```

Using this instruction, any user can access the tables without giving the owner prefix as long as the table owner has granted the privileges to access the tables.

You can make the database interface select the current schema automatically with the following fglprofile entry:

```
dbi.database.dbname.schema = "schema"
```

When using multiple database schemas, it is recommended that you create them in separated tablespaces to enable independent backups and keep logical sets of tables together. The simplest way is to define a default tablespace when creating the schema owner:

```
CREATE USER user IDENTIFIED BY password
   DEFAULT TABLESPACE deftablespace
    TEMPORARY TABLESPACE tmptablespace
```

Data storage concepts

An attempt should be made to preserve as much of the storage specification as possible when converting from Informix® to ORACLE. Most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be reused for the ORACLE database.

Storage concepts are quite similar in Informix® and in ORACLE, but the names are different.

This table compares Informix® storage concepts to ORACLE storage concepts:

Table 194: Physical units of storage

Informix®	**ORACLE**
The largest unit of physical disk space is a "**chunk**", which can be allocated either as a cooked file (I/O is controlled by the OS) or as raw device (=UNIX partition, I/O is controlled by the database engine). A "dbspace" uses at least one "chunk" for storage.	One or more "**data files**" are created for each "tablespace" to physically store the data of all logical structures. Like Informix® "chunks", a "data file" can be an OS file or a raw device.

Informix®	**ORACLE**
You must add "chunks" to "dbspaces" in order to increase the size of the logical unit of storage.	You can add "data files" to a "tablespace" in order to increase the size of the logical unit of storage or you can use the AUTOEXTEND option when using OS files.
A "**page**" is the smallest physical unit of disk storage that the engine uses to read from and write to databases. A "chunk" contains a certain number of "pages". The size of a "page" must be equal to the operating system's block size.	At the finest level of granularity, ORACLE stores data in "**data blocks**" which size corresponds to a multiple of the operating system's block size. You set the "data block" size when creating the database.
An "**extent**" consists of a collection of contiguous "pages" that the engine uses to allocate both initial and subsequent storage space for database tables. When creating a table, you can specify the first extent size and the size of future extents with the EXTENT SIZE and NEXT EXTENT options. For a single table, "extents" can be located in different "chunks" of the same "dbspace".	An "**extent**" is a specific number of contiguous "data blocks", obtained in a single allocation. When creating a table, you can specify the first extent size and the size of future extents with the STORAGE() option. For a single table, "extents" can be located in different "data files" of the same "tablespace".

Table 195: Logical units of storage

Informix®	**ORACLE**
A "**table**" is a logical unit of storage that contains rows of data values.	Same concept as Informix®.
A "**database**" is a logical unit of storage that contains table and index data. Each database also contains a system catalog that tracks information about database elements like tables, indexes, stored procedures, integrity constraints and user privileges.	Same concept as Informix®, but one ORACLE instance can manage only one database, in the meaning of Informix®.
Database tables are created in a specific "**dbspace**", which defines a logical place to store data. If no dbspace is given when creating the table, Informix® defaults to the current database dbspace.	Database tables are created in a specific "**tablespace**", which defines a logical place to store data. If no tablespace is given when creating the table, ORACLE defaults to the user's default tablespace.
The total disk space allocated for a table is the "**tblspace** ", which includes "pages" allocated for data, indexes, blobs, tracking page usage within table extents. Do not confuse the Informix® "tblspace" concept and ORACLE "tablespaces".	A "**segment**" is a set of "extents" allocated for a certain logical structure. There are four different types of segments, including data segments, index segments, rollback segments and temporary segments.

Table 196: Other concepts related to storage

Informix®	**ORACLE**
When initializing an Informix® engine, a "**root dbspace**" is created to store information about all databases, including storages information (chunks used, other dbspaces, etc.)	Each ORACLE database has a "**control file**" that records the physical structure of the database, like the database name, location and names of "data files" and "redo log" files, and time stamp of database creation.
The "**physical log**" is a set of continuous disk pages where the engine stores "before-images" of data that has been modified during processing. The "**logical log**" is a set of "**logical-log files**" used to record logical operations during online processing. All transaction information is stored in the logical log files if a database has been created with transaction log. Informix® combines "physical log" and "logical log" information when doing fast recovery. Saved "logical logs" can be used to restore a database from tape.	A "**rollback segment**" records the actions of SQL transactions that could be rolled back, and it records the data as it existed before an operation in a transaction. The "**redo log files**" hold all changes made to the database, in case the database experiences an instance failure. Each database has at least two "redo log files". Redo entries record data that can be used to reconstruct all changes made to the database, including the rollback segments stored in the database buffers of the SGA. Therefore, the online redo log also protects rollback data.

Data consistency and concurrency

Data consistency involves readers that want to access data currently modified by writers, and *concurrency data access* involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification, or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is page.

ORACLE

When data is modified, **exclusive locks** are set and held until the end of the transaction. For data consistency, ORACLE uses a **multi-version consistency model**: a copy of the original row is kept for readers before performing writer modifications. Readers do not have to wait for writers as in Informix®. The simplest way to think of Oracle's implementation of read consistency is to imagine each user accessing a private copy of the database, hence the multi-version consistency model. The **lock wait mode** cannot be changed session wide as in Informix®; the waiting behavior can be controlled with a SELECT FOR UPDATE NOWAIT only. Locks are set at the **row level** in ORACLE, and this cannot be changed.

Control:

- Lock wait mode (on SELECT only): SELECT ... FOR UPDATE NOWAIT
- Isolation level: SET TRANSACTION ISOLATION LEVEL TO ...
- Explicit exclusive lock: SELECT ... FOR UPDATE [NOWAIT]

Defaults:

- The default isolation level is Read Committed (readers cannot see uncommitted data, no shared lock is set when reading data).

The main difference between Informix® and ORACLE is that readers do not have to wait for writers in ORACLE.

Solution

The SET ISOLATION TO ... Informix® syntax is replaced by ALTER SESSION SET ISOLATION_LEVEL ... in Oracle. The next table shows the isolation level mappings done by the database driver:

Table 197: Isolation level mappings done by the Oracle database driver

SET ISOLATION instruction in program	Native SQL command
SET ISOLATION TO DIRTY READ	ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED
SET ISOLATION TO CURSOR STABILITY	ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED
SET ISOLATION TO REPEATABLE READ	ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE

ORACLE does not provide a dirty read mode, the (session wide) lock wait mode cannot be changed and the locking precision is always at the row level. Based on this, it is recommended that you work with Informix® in the read committed isolation level (default), make processes wait for each other (lock mode wait), and use the default page-level locking granularity.

See the Informix® and ORACLE documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Informix® and ORACLE handle transactions differently. The differences in the transactional models can affect the program logic.

Informix® native mode (non ANSI):

- DDL statements can be executed (and canceled) in transactions.

- Transactions must be started with BEGIN WORK. Statements executed outside of a transaction are automatically committed.

ORACLE:

- Beginnings of transactions are implicit; two transactions are delimited by COMMIT or ROLLBACK.
- The current transaction is automatically committed when a DDL statement is executed.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, in accordance with the transaction model.

Informix® version 11.50 introduces savepoints with the following instructions:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] ]
RELEASE SAVEPOINT name
```

ORACLE supports savepoints too. However, there are differences:

1. Savepoints cannot be declared as UNIQUE
2. Rollback must always specify the savepoint name
3. You cannot release savepoints (RELEASE SAVEPOINT)

Solution

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with ORACLE. The Informix® behavior is simulated with an autocommit mode in the ORACLE interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program.

When executing a DDL statement inside a transaction, ORACLE automatically commits the transaction. Therefore, you must extract the DDL statements from transaction blocks.

If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

See also SELECT FOR UPDATE

Database users

Until version 11.70.xC2, Informix® database users must be created at the operating system level and must be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers. Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

Oracle users can be authenticated in different manner: as database users, as operating system users or by delegating authentication to another service, like Kerberos or LDAP.

Oracle users must be created in the database with a CREATE USER command, to create a user authenticated by the database server:

`CREATE USER username IDENTIFIED BY password`

Oracle users can also be created with the "IDENTIFIED EXTERNALLY" clause:

`CREATE USER username IDENTIFIED EXTERNALLY`

In this case, ORACLE trusts the operating system to authenticate the user. See the Oracle documentation for OS user authentication configuration, especially the `OS_AUTHENT_PREFIX` (empty string) and `REMOTE_OS_AUTHENT` (true) server parameters. Note also that the Oracle user name needs to be specified in uppercase in the CREATE USER instruction, and gets an additional prefix, according to the operating system (domain name on Windows platforms)

In ORACLE, is it also possible to define users that are defined in a central LDAP directory, with the "IDENTIFIED GLOBALLY" clause:

`CREATE USER` *`username`* `IDENTIFIED GLOBALLY AS '`*`distinguished_name`*`'`

Global users are registered and managed by an external LDAP service, and are identified by the distinguished name (DN).

Solution

Based on the application logic, you must create one or several ORACLE users. Use RDBMS or external authentication according to your needs. If you want to keep the same Informix® OS users, you must configure Oracle for OS authentication, and create users with the IDENTIFIED EXTERNALLY option. Consider however to use real RDBMS users instead, and ask for login/password when connecting a program to Oracle.

To connect to an Oracle server from a program, use the `CONNECT TO` instruction. When the `USER/USING` clause is not specified, external authentication takes place. You can check if external or rdbms authentication takes place with the FGLSQLDEBUG output (check the line containing "Credential flag").

Tester with Oracle 11.2 on a Linux system (the Linux user login name is "sf" in lowercase):

```
$ sqlplys / as sysdba

SQL> show parameter os_authent_prefix;
NAME                 TYPE            VALUE
---------------------------------------------
os_authent_prefix    string

SQL> show parameter remote_os_authent;
NAME                 TYPE            VALUE
---------------------------------------------
remote_os_authent    boolean         TRUE

SQL> create user "SF" identified externally;
User created.

SQL> grant connect, resource to "SF";
Grant succeeded.
```

To connect to Oracle as an external user declared with IDENTIFIED EXTERNALLY (authenticated by the operating system), do not specify any login/password. For example, omit the `USER/USING` clause in the `CONNECT TO` instruction:

```
CONNECT TO "orc1fox+driver='dbmora'"
```

If no db login is specified, the Oracle driver will open a database session with the OCI_CRED_EXT credentials.

An Oracle connection can also be established as SYSDBA or SYSOPER users. This is possible by specifying the following strings after the user name in the `USER` clause of the `CONNECT TO` instruction:

Table 198: Oracle connection as SYSDBA or SYSOPER

String passed to USER clause after user name	Effect as Oracle connection
`/SYSDBA`	Connection will be established as SYSDBA user.
`/SYSOPER`	Connection will be established as SYSOPER user.

Specify the user login before the /SYSDBA or /SYSOPER strings:

```
CONNECT TO "orc1fox+driver='dbmora'"
    USER "orauser/SYSDBA" USING "fourjs"
```

Setting privileges

Informix® and ORACLE user privileges management are quite similar.

ORACLE provides roles to group privileges which then can be assigned to users. Starting with version 7.20, Informix® provides roles also. But users must execute the SET ROLE statement in order to enable a role. ORACLE users do not have to explicitly set a role, they are assigned to a default privilege domain (set of roles). More than one role can be enabled at a time with ORACLE.

Informix® users must have at least the CONNECT privilege to access the database:

`GRANT CONNECT TO (PUBLIC|`*`username`*`)`

ORACLE users must have at least the CREATE SESSION privilege to access the database. This privilege is part of the CONNECT role.

`GRANT CONNECT TO (PUBLIC|`*`username`*`)`

Informix® database privileges do NOT correspond exactly to ORACLE CONNECT, RESOURCE and DBA roles. However, roles can be created with equivalent privileges.

Solution

Create a role which groups Informix® CONNECT privileges, and assign this role to the application users:

```
CREATE ROLE ifx_connect IDENTIFIED BY oracle;
GRANT CREATE SESSION, ALTER SESSION, CREATE ANY VIEW, ... TO ifx_connect;
GRANT ifx_connect TO user1;
```

Data dictionary
Oracle Database related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way: As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

Oracle does not implement a native BOOLEAN type in SQL types. However, a BOOLEAN type exists in the PL/SQL language.

Solution
The Oracle database interface converts the BOOLEAN type to CHAR(1) columns and stores '1' or '0' values in the column.
CHARACTER data types

Informix® supports the following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/

NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics; the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases.

ORACLE provides the following types to store character strings:

- CHAR(N) with N specified in bytes or characters according to the length semantics (max size is 2000 bytes)
- VARCHAR2(N) with N specified in bytes or characters according to the length semantics (max size is 4000 bytes - standard type)
- NCHAR(N) with N specified in characters (max size is 2000 bytes)
- NVARCHAR2(N) with N specified characters (max size is 4000 bytes - standard type)

  Note: Oracle 12c introduced extended character types with the MAX_STRING_SIZE=EXTENDED server parameter. You can use VARCHAR2 types with a size up to 32Kb when MAX_STRING_SIZE=EXTENDED is set. (You need to close/upgrade/alter/reopen your database, see Oracle documentation for details). However, the storage technique used by Oracle 12c for such a large string type is different from the native/standard VARCHAR2(4000) type. Large character strings will be stored as LOBs. Extended character types are not supported by Genero's Oracle database driver.

In ORACLE CHAR(N)/VARCHAR2(N) types, the size N can be specified in character or byte units, according to length semantics settings.

When comparing CHAR and VARCHAR2 values in ORACLE, the trailing blanks are significant; this is not the case when using Informix® VARCHARs. However, before comparing string values, ORACLE blank-pads CHAR(N) data to the maximum length of both operands. As result, it looks like trailing blanks are no significant in CHAR(N) comparison. For example, a column defined as CHAR(5) with the value `'abc '` (with 2 trailing blanks) will not be equal to `'abc'`, but when comparing `(col = 'abc')`, ORACLE will add 2 blanks to the right operand and values will match. Blank padding does not occur for VARCHAR2() data, as result, the expression `(col = 'abc')` will be false, if col VARCHAR2 does not exactly contain the value `'abc'`. For more details, see blank-padded and non-padded comparison semantics in ORACLE documentation.

ORACLE treats empty strings like NULL values; Informix® doesn't. See issue Empty Character Strings for more details.

With ORACLE, you can define a Database Character Set and a National Character Set: ORACLE uses the Database Character Set to store string data in the CHAR/VARCHAR2 columns, and uses the National Character Set for NCHAR/NVARCHAR2 columns.

Solution

Informix® CHAR(N) types must be mapped to ORACLE CHAR(N) types, and Informix® VARCHAR(N) or LVARCHAR(N) columns must be mapped to ORACLE VARCHAR2(N).

Check that your database tables does not use CHAR, VARCHAR or LVARCHAR types with a length exceeding the ORACLE limits of CHAR/VARCHAR2.

When using a multibyte character set (such as UTF-8), configure ORACLE to use character length semantics, define CHAR/VARCHAR2 database columns with a size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from an ORACLE database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in the database, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

The ORACLE client character set must correspond to the Genero runtime system locale (LANG/LC_ALL). You can define the ORACLE client character set with the NLS_LANG environment variable.

See also the section about Localization.

NUMERIC data types

Informix® supports several data types to store numbers:

Table 199: Informix® numeric data types

Informix® data type	Description
SMALLINT	16 bit signed integer
INT / INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
INT8	64 bit signed integer (replaced by BIGINT)
DEC / DECIMAL	Equivalent to DECIMAL(16)
DEC / DECIMAL(p)	Floating-point decimal number
DEC / DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL / SMALLFLOAT	32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)]	64-bit floating point decimal (C double)

ORACLE supports only one data type to store numbers:

Table 200: Oracle numeric data types

ORACLE data type	Description
NUMBER(p,s) (1<=p<= 38, -84<=s<=127)	Fixed point decimal numbers.
NUMBER(p) (1<=p<= 38)	Integer numbers with a precision of p.
NUMBER(*,s)	Fixed point decimal numbers with a precision of 38 digits.
NUMBER	Floating point decimals with a precision of 38 digits.
FLOAT(b) (1<=b<= 126)	Floating point numbers with a binary precision b. This is a sub-type of NUMBER.
BINARY_FLOAT (since Oracle 10g)	32-bit floating point number.
BINARY_DOUBLE (since Oracle 10g)	64-bit floating point number.

ANSI types like SMALLINT, INTEGER are supported by ORACLE but will be converted to the native NUMBER type.

When dividing INTEGERs or SMALLINTs, Informix® rounds the result (7 / 2 = 3), while ORACLE doesn't, because it does not have a native integer data type (7 / 2 = 3.5)

Solution

We recommend that you use the following conversion rules:

Table 201: Conversion rules (Informix® vs. Oracle)

Informix® data type	ORACLE data type (before 10g)	ORACLE data type (since 10g)
DECIMAL(p,s), MONEY(p,s)	NUMBER(p,s)	NUMBER(p,s)
DECIMAL(p)	FLOAT(p * 3.32193)	FLOAT(p * 3.32193)
DECIMAL (not recommended)	FLOAT	FLOAT
SMALLINT	NUMBER(5,0)	NUMBER(5,0)
INTEGER	NUMBER(10,0)	NUMBER(10,0)
BIGINT	NUMBER(20,0)	NUMBER(20,0)
INT8	NUMBER(20,0)	NUMBER(20,0)
SMALLFLOAT	NUMBER	BINARY_FLOAT
FLOAT[(p)]	NUMBER	BINARY_DOUBLE

Avoid dividing integers in SQL statements. If you do divide an integer, use the TRUNC() function with ORACLE.

When creating a table directly in Oracle's sqlplus, using ANSI data types INTEGER, SMALLINT, you do actually create columns with the NUMBER type, which has a precision of 38 digits. As result, it is not possible to distinguish the original types used in CREATE TABLE, nor can it be possible to distinguish the columns created explicitely with the native NUMBER type, in the next example, all column with be of type NUMBER:

```
$ sqlplus ...
sql> CREATE TABLE mytab (
   col1 INTEGER,
   col2 SMALLINT,
   col3 NUMBER,
   ...
```

When extracting the database schema with fgldbsch, NUMBER, NUMBER(p>32) and NUMBER(p>32,s) types will by default give an extraction error. However, these types can be converted to DECIMAL(32) and DECIMAL(32,s) with the `-cv` option, by using the "B" character at positions 22 (for NUMBER) and 23 (for NUMBER(p>32[,s])).

> **Note:** When fetching a NUMBER[(p>32,s)] into a BDL DECIMAL(32[,s]) type, if the value stored in the NUMBER column has more than 32 digits, it will be rounded to fit into a DECIMAL(32), or the overflow error -1226 will occur when fetching into a DECIMAL(32,s). Note that it must be allowed to fetch numeric expressions such as 1/3 (=0.333333333333....) into a DECIMAL(p,s), even if such expression will produce more than 32 digits with Oracle.

When creating a table in a BDL program with DECIMAL(p), this type is converted to native Oracle FLOAT(p*3.32193). When creating a table in a BDL program with DECIMAL (without precision) this type is converted to native Oracle FLOAT. The native Oracle FLOAT[(p)] type can be extracted by fgldbsch, but Oracle's FLOAT has a higher precision than the BDL DECIMAL type, which can lead to value rounding when fetching rows.

With Oracle versions older than 10g, when creating tables in a BDL program with SMALLFLOAT or FLOAT types, these types are mapped to NUMBER (The native Oracle FLOAT(b) type could have been used, but this type is reserved to map DECIMAL(p) types). Starting with Oracle 10g, SMALLFLOAT or FLOAT types

will respectively be converted to BINARY_FLOAT and BINARY_DOUBLE native Oracle types, which can be extracted by fgldbsch and mapped back to BDL SMALLFLOAT and FLOAT respectively in the .sch file.

> **Note:** As a general recommendation, do not use DECIMAL[(p)] or SMALLFLOAT/FLOAT floating point types in business applications. These types should only be used for scientific data storage.

DATE and DATETIME data types

Informix® provides two data types to store date and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

ORACLE provides only the following data types to store date and time data:

- DATE = for year, month, day, hour, min, second storage.
- TIMESTAMP= for year, month, day, hour, min, second, fraction storage.

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string contains matching environment parameters (i.e. DBDATE, GL_DATETIME).

As in Informix®, ORACLE can convert quoted strings to DATE or TIMESTAMP data if the contents of the string matches the NLS date format parameters (NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT). The TO_DATE() and TO_TIMESTAMP() SQL functions convert strings to dates or timestamps, according to a given format. The TO_CHAR() SQL function allows you to convert dates or timestamps to strings, according to a given format.

Date arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used, and an INTERVAL value if a DATETIME is used in the expression. In ORACLE, the result of an arithmetic expression involving DATE values is a NUMBER of days; the decimal part is the fraction of the day (0.5 = 12H00, 2.00694444 = (2 + (10/1440)) = 2 days and 10 minutes). The result of an expression involving Oracle TIMESTAMP data is of type INTERVAL. See the Oracle documentation for more details.
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. ORACLE does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix-specific and have no equivalent in ORACLE.
- To compare dates that have time data in ORACLE, you can use the ROUND() or TRUNC() SQL functions.

Solution

Storing BDL DATE values

The ORACLE DATE type is used to store Genero BDL DATE values. However, keep in mind that the ORACLE DATE type stores also time (hh:mm:ss) information. The database interface automatically sets the time part to midnight (00:00:00) during input/output operations.

You must be very careful since manual modifications of the database might set the time part, for example:

UPDATE table SET date_col = SYSDATE

(SYSDATE is equivalent to CURRENT YEAR TO SECOND in Informix®).

After this type of update, when columns have date values with a time part different from midnight, some SELECT statements might not return all the expected rows.

When fetching ORACLE DATE values into Genero BDL DATE or DATETIME variables, the date and time information is directly set for the individual date/time parts and the conversion is straight forward. But when fetching an ORACLE DATE into a CHAR or VARCHAR variable, date to string conversion occurs. Since ORACLE DATEs are equivalent of Informix® DATETIME YEAR TO SECOND, the values are by default converted with the ISO format (YYYY-MM-DD hh:mm:ss), which is not the typical Informix® behavior where DATEs are formatted according to the DBDATE environment variable. If your application fetches DATE values into CHAR/VARCHAR and you want to get the DBDATE conversion, you must set the following FGLPROFILE entry:

```
dbi.database.dbname.ora.date.ifxfetch = true
```

Storing BDL DATETIME values

Informix® DATETIME data with any precision from YEAR to SECOND is stored in ORACLE DATE columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the ORACLE DATE value will be "1900-01-01 11:45:00".

Informix® DATETIME YEAR TO FRACTION(n) data is stored in ORACLE TIMESTAMP columns. The TIMESTAMP data type can store up to 9 digits in the fractional part, and therefore can store all precisions of Informix® DATETIME.

Important:

- Most arithmetic expressions involving dates (for example, to add or remove a number of days from a date) will produce the same result with ORACLE. But keep in mind that ORACLE evaluates date arithmetic expressions to NUMBERs (*days*.*fraction*) while Informix® evaluates to INTEGERs when only DATEs are used in the expression, or to INTERVALs if at least one DATETIME is used in the expression.
- Even if a configuration parameter exists to get the Informix® behavior, avoid to fetch date values into CHAR or VARCHAR, to bypass the DBDATE / ISO format conversion difference with ORACLE.

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

Informix's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

ORACLE supports the INTERVAL data type similar to Informix®, with two classes (YEAR TO MONTH and DAY TO SECOND), but Oracle's INTERVAL cannot be defined with a precision different from these two classes (for example, you cannot define an INTERVAL HOUR TO MINUTE in Oracle). The class DAY TO SECOND(n) is equivalent to the Informix® INTERVAL class DAY TO FRACTION(n).

Solution

Informix® INTERVAL YEAR(n) TO MONTH data is stored in Oracle INTERVAL YEAR(n) TO MONTH columns. These data types are equivalent.

Informix® INTERVAL DAY(n) TO FRACTION(p) data is stored in Oracle INTERVAL DAY(n) TO SECOND(p) columns. These data types are equivalent.

Other Informix® INTERVAL types must be stored in CHAR() columns as with Oracle 8i, because the high qualifier precision cannot be specified with Oracle INTERVALs. For example, Informix® INTERVAL HOUR(5) TO MINUTE has no native equivalent in Oracle.

SERIAL data types

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

ORACLE provides several solutions to implement auto-incremented columns:

1. Sequence objects can be created to generate numbers (CREATE SEQUENCE, *seqname*.currval).
2. Since ORACLE 12c, it is possible to reference a sequence in DEFAULT ON NULL column clauses.
3. Since ORACLE 12c, you can define columns with the GENERATE ... AS IDENTITY clause.

Details about ORACLE sequences:

- Sequences are totally detached from tables.
- The purpose of sequences is to provide unique integer numbers.
- Sequences are identified by a sequence name.
- To create a sequence, you must use the CREATE SEQUENCE statement. Once a sequence is created, it is permanent (like a table).
- To get a new sequence value, you must use the `nextval` keyword, preceded by the name of the sequence. The *`seqname`*`.nextval` expression can be used in INSERT statements:

  ```
  INSERT INTO tab1 VALUES ( tab1_seq.nextval, ... )
  ```
- To get the last generated number, ORACLE provides the `currval` keyword:

  ```
  SELECT seqname.currval FROM DUAL
  ```
- In order to improve performance, ORACLE can handle a set of sequences in the cache (See CREATE SEQUENCE syntax in the ORACLE documentation).

Solution

The SERIAL data type can be emulated with sequences used in INSERT triggers or with the DEFAULT ON NULL clause.

The method used to emulate SERIAL types is defined by the `ifxemul.datatype.serial.emulation` FGLPROFILE parameter:

```
dbi.database.dbname.ifxemul.datatype.serial.emulation =
{"native"|"native2"|"regtable"}
```

- native: uses insert triggers with sequences.
- native2: uses DEFAULT ON NULL column clause with sequences.
- regtable: uses insert triggers with the SERIALREG table.

The default emulation technique is "`native`".

This entry must be used in conjunction with:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial = {true|false}`

If the `datatype.serial` entry is set to false, the emulation method is ignored.

> **Important:** The "`regtable`" emulation based on the SERIALREG table is provided to simplify the migration from Informix. We strongly recommend that you use the `native` or `native2` method instead. The "`native2`" method is the fastest solution when inserting a large number of rows in the database.

Notes common to all serial emulation modes

When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the Oracle interface automatically creates the additional SQL objects (column clauses, sequences or triggers) to generate numbers when an INSERT statement is performed.

Users executing programs which create tables with SERIAL columns must have the CONNECT and RESOURCE roles assigned to create triggers and sequences.

SERIAL[(n)] data types are converted to NUMBER(10,0), while SERIAL8[(n)] and BIGSERIAL[(n)] are replaced by NUMBER(20,0).

For SERIAL types, the SQLCA.SQLERRD[2] register is filled as expected with the last generated serial value. However, since SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGSERIAL (NUMBER(20,0)) auto-incremented columns. If you are using BIGSERIAL columns, you must the fetch the sequence pseudo-column CURR_VAL or fetch the LASTSERIAL column from the SERIALREG table, if used.

Check whether your application uses tables with a SERIAL column that can contain a NULL value: INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements that do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

For SQL portability, INSERT statements should be reviewed to remove the SERIAL column from the list. For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES (0, p_value)
```

can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.* INSERT INTO tab VALUES (rec.*) -- will use the serial
column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

When using the Static SQL INSERT or UPDATE syntax using record.* without braces, make sure that you database schema files contain information about serials: This information can be lost when extracting the schema from an Oracle database. See Database Schema for more details about the serial flag in column type encoding (data type code must be 6).

If the "`native`" or "`regtable`" emulation is used, inserting rows with ORACLE tools like SQL*Plus or SQL*Loader will execute the INSERT triggers. When loading big tables, you can disable triggers with `ALTER TRIGGER [ENABLE | DISABLE]` (see ORACLE documentation for more details). After reactivation of the serial triggers, the SERIAL sequences must be re-initialized (use `serialpkg.create_sequence('tab','col')`) or re-execute the PL/SQL script containing the sequence and trigger creation.

Using the `native` serial emulation

Each table having a SERIAL column needs an INSERT TRIGGER and a SEQUENCE dedicated to SERIAL generation.

To know how to write those sequences and triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native SQL commands to create the sequence and the trigger.

For temporary tables, the trigger and the sequence are dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

Using the `native2` serial emulation

With this emulation, a SERIAL type is converted to a DEFAULT ON NULL clause using a sequence is created automatically by the database driver, for example:

```
CREATE TABLE t1 ( mykey SERIAL(100), .... )
```

is converted to:

```
CREATE SEQUENCE t1_srl INCREMENT BY 1 START WITH 100
```

```
CREATE TABLE t1 (mykey NUMBER(10,0) DEFAULT ON NULL t1_srl.nextval , ...
```

For temporary tables, the sequence is dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

> **Note:** The `native2` serial emulation uses the DEFAULT ON NULL clause, supported by Oracle, starting from version 12.1.

Using the `regtable` serial emulation

Each table having a SERIAL column needs an INSERT TRIGGER which uses the SERIALREG table dedicated to SERIAL registration.

First, you must prepare the database and create the SERIALREG table as follows:

```
CREATE TABLE serialreg (
    tablename VARCHAR2(50) NOT NULL,
    lastserial NUMBER(20,0) NOT NULL,
    PRIMARY KEY ( tablename )
)
```

> **Important:** This table must exist in the database before creating the serial triggers.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. SERIAL8/BIGSERIAL columns must be converted to NUMBER(20,0). To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at start values and you might get duplicated values.

For temporary tables, the trigger is dropped automatically after a "DROP TABLE temptab" or when the program disconnects from the database.

ROWIDs

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

ORACLE supports ROWIDs, but the data type is different from Informix® ROWIDs: ORACLE rowids are CHAR(18).

For example: `AAAA8mAALAAAAQkAAA`

Since ORACLE rowids are physical addresses, they cannot be used as permanent row identifiers (After a DELETE, an INSERT statement might reuse the physical place of the deleted row, to store the new row).

With Informix®, SQLCA.SQLERRD[6] contains the ROWID of the last INSERTed or UPDATEd row. This is not supported with ORACLE because ORACLE ROWID are not INTEGERs.

Solution

If the BDL application uses Informix® rowids as primary keys, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported) or ORACLE rowids as CHAR(18) (Informix® rowids will fit in this char data type).

If you cannot avoid the use of rowids, you must change the type of the variables which hold ROWID values. Instead of using INTEGER, you must use CHAR(18). Informix® rowids (INTEGERs) will automatically fit into a CHAR(18) variable.

All references to SQLCA.SQLERRD[6] must be removed because this variable will not contain the ROWID of the last INSERTed or UPDATEd row when using the ORACLE interface.

The RAW data type

ORACLE supports the RAW data type to hold binary data. This data type is for example used to return values from the `SYS_GUID()` SQL function.

Solution

The ORACLE RAW values can be converted to a character string in the hexadecimal notation.

When fetching rows from the database, the database driver will automatically convert ORACLE RAW values to hexadecimal. On the other hand, when using SQL parameters, the database driver will convert hexadecimal VARCHAR strings to binary data.

Since each byte is represented with two characters in the hexadecimal notation, you must define a VARCHAR(N*2) variable to hold the values of a native RAW(N) column.

When extracting a database schema with the `fgldbsch` tool, the ORACLE RAW(N) type is converted to VARCHAR2(N*2).

Large OBject (LOB) types

Informix® uses the TEXT and BYTE data types to store very large texts or images. ORACLE 8 provides CLOB, BLOB, and BFILE data types. Columns of these types store a kind of pointer (lob locator). This technique allows you to use more than one CLOB / BLOB / BFILE column per a table.

Solution

The ORACLE database interface can convert BDL TEXT data to CLOB and BYTE data to BLOB.

Genero TEXT/BYTE program variables have a limit of 2 gigabytes, make sure that the large object data does not exceed this limit.

ORACLE BFILEs are not supported.

The ALTER TABLE instruction

Informix® and ORACLE have different implementations of the ALTER TABLE instruction. For example, Informix® allows you to use multiple ADD clauses separated by commas. This is not supported by ORACLE:

Informix®:

```
ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

ORACLE:

```
ALTER TABLE customer ADD(col1 INTEGER, col2 CHAR(20))
```

Solution

No automatic conversion is done by the database interface. There is no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

Constraints

Constraint naming syntax

Both Informix® and ORACLE support primary key, unique, foreign key, default and check constraints, but the constraint naming syntax is different: ORACLE expects the "CONSTRAINT" keyword **before** the constraint specification and Informix® expects it **after**.

UNIQUE constraint example

Table 202: UNIQUE constraint example (Informix® vs ORACLE)

Informix®	ORACLE

`CREATE TABLE emp (`
`  ...`
`  emp_code CHAR(10) UNIQUE`
`  CONSTRAINT pk_emp,`
	`CREATE TABLE emp (`
	`  ...`
	`  emp_code CHAR(10)`
	`  CONSTRAINT pk_emp UNIQUE,`
	`  ...`

Primary keys

Like Informix®, ORACLE creates an index to enforce PRIMARY KEY constraints (some RDBMS do not create indexes for constraints). Using "CREATE UNIQUE INDEX" to define unique constraints is obsolete (use primary keys or a secondary key instead).

Unique constraints

Like Informix®, ORACLE creates an index to enforce UNIQUE constraints (some RDBMS do not create indexes for constraints).

When using a unique constraint, Informix® allows only one row with a NULL value, while ORACLE allows several rows with NULL! Using CREATE UNIQUE INDEX is obsolete.

Foreign keys

Both Informix® and ORACLE support the ON DELETE CASCADE option. To defer constraint checking, Informix® provides the SET CONSTRAINT command while ORACLE provides the ENABLE and DISABLE clauses.

Check constraints

The check condition may be any valid expression that can be evaluated to TRUE or FALSE, including functions and literals. You must verify that the expression is not Informix® specific.

Null constraints
Informix® and ORACLE support not null constraints, but Informix® does not allow you to give a name to "NOT NULL" constraints.

Solution

Constraint naming syntax

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for ORACLE.

Triggers

Informix® and ORACLE provide triggers with similar features, but the trigger creation syntax and the programming languages are totally different.

Informix® triggers define the stored procedures to be called when a database event occurs (before | after insert | update | delete ...), while ORACLE triggers can hold a procedural block.

In ORACLE, triggers can be created with 'CREATE OR REPLACE' to keep privileges settings. With Informix®, you must drop and create again.

ORACLE V8 provides an 'INSTEAD OF' option to completely replace the INSERT, UPDATE or DELETE statement. This is provided to implement complex storage operations, for example on views that are usually read-only (you can attach triggers to views).

ORACLE allows you to create multiple triggers on the same table for the same trigger event, but it does not guarantee the execution order.

Solution

Informix® triggers must be converted to ORACLE triggers "by hand".

Stored procedures

Both Informix® and ORACLE support stored procedures, but the programming languages are totally different: **SPL** for Informix® versus **PL/SQL** for ORACLE.

In ORACLE, stored procedures and functions can be implemented in packages (similar to BDL modules). This is a powerful feature which enables structured procedural programming in the database. ORACLE itself implements system tools with packages (dbms_sql, dbms_output, dbms_lock). Procedures, functions and packages can be created with 'CREATE OR REPLACE' to keep privileges settings. With Informix®, you must drop and create again.

ORACLE uses a different privilege context when using dynamic SQL in PL/SQL; roles are not effective. Users must have direct privileges settings in order to perform DDL or DML operations inside dynamic SQL.

Solution

Informix® stored procedures must be converted to ORACLE "by hand".

Try to use ORACLE packages in order to group stored procedures into modules.

See SQL Programming for more details about executing stored procedures with ORACLE.

Name resolution of SQL objects

Informix® uses the following form to identify an SQL object:

```
database[@dbservername]:][{owner|"owner"}.]identifier
```

The ANSI convention is to use double quotes for identifier delimiters (For example: "tabname"."colname").

When using double-quoted identifiers, both Informix® and ORACLE become case sensitive. Unlike Informix®, ORACLE database object names are stored in UPPERCASE in system catalogs. That means that SELECT "col1" FROM "tab1" will produce an error because those objects are identified by "COL1" and "TAB1" in ORACLE system catalogs.

in Informix® ANSI compliant databases:

- The table name must include "owner", unless the connected user is the owner of the database object.
- The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in double quotes.

With ORACLE, an object name takes the following form:

```
[(schema|"schema").](identifier|"identifier")[@database-link]
```

ORACLE has separate namespaces for different classes of objects (tables, views, triggers, indexes, clusters).

Object names are limited to 30 chars in ORACLE.

An ORACLE database schema is owned by a user (usually, the application administrator) and this user must create PUBLIC SYNONYMS to provide a global scope for his table names. PUBLIC SYNONYMS can have the same name as the schema objects they point to.

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

Check that you do not use single-quoted or double-quoted table names or column names in your source. Those quotes must be removed because the database interface automatically converts double quotes to single quotes, and ORACLE does not allow single quotes as database object name delimiters.

See also the issue Database Concepts

NULLs in indexed columns

Oracle btree indexes do not store null values, while Informix® btree indexes do. This means that if you index a single column and select all the rows where that column is null, Informix® will do an indexed read to fetch just those rows, but Oracle will do a sequential scan of all rows to find them. Having an index unusable for "is null" criteria can also completely change the behavior and performance of more complicated selects without causing a sequential scan.

Solution

Declare the indexed columns as NOT NULL with a default value and change the program logic. If you do not want to change the programs, partitioning the table so that the nulls have a partition of their own will reduce the sequential scan to just the nulls (un-indexed) partition, which is relatively fast.

Data type conversion table: Informix to Oracle

Table 203: Data type conversion table (Informix to Oracle)

Informix® data types	**ORACLE data types (Versions 10.x and higher)**
CHAR(n)	CHAR(n) (limit = 2000b!)
VARCHAR(n[,m])	VARCHAR2(n)

Informix® data types	**ORACLE data types (Versions 10.x and higher)**
	(limit = 4000b!)
LVARCHAR(n)	VARCHAR2(n) (limit = 4000b!)
NCHAR(n)	NCHAR(n) (limit = 2000b!)
NVARCHAR(n[,m])	NVARCHAR2(n) (limit = 4000b!)
BOOLEAN	CHAR(1)
SMALLINT	NUMBER(5,0)
INT / INTEGER	NUMBER(10,0)
BIGINT	NUMBER(20,0)
INT8	NUMBER(20,0)
SERIAL[(start)]	NUMBER(10,0) (see note 1)
BIGSERIAL[(start)]	NUMBER(20,0) (see note 1)
SERIAL8[(start)]	NUMBER(20,0) (see note 1)
DOUBLE PRECISION / FLOAT[(n)]	BINARY_DOUBLE
REAL / SMALLFLOAT	BINARY_FLOAT
NUMERIC / DEC / DECIMAL(p,s)	NUMBER(p,s)
NUMERIC / DEC / DECIMAL(p)	FLOAT(p*3.32193)
NUMERIC / DEC / DECIMAL *(not recommended)*	FLOAT
MONEY(p,s)	NUMBER(p,s)
MONEY(p)	NUMBER(p,2)
MONEY	NUMBER(16,2)
TEXT	CLOB (using <= 2Gb!)
BYTE	BLOB (using <= 2Gb!)
DATE	DATE
DATETIME YEAR TO YEAR	DATE
DATETIME YEAR TO MONTH	DATE
DATETIME YEAR TO DAY	DATE
DATETIME YEAR TO HOUR	DATE
DATETIME YEAR TO MINUTE	DATE
DATETIME YEAR TO SECOND	DATE

Informix® data types	**ORACLE data types (Versions 10.x and higher)**
DATETIME YEAR TO FRACTION(n)	TIMESTAMP(n)
DATETIME MONTH TO MONTH	DATE
DATETIME MONTH TO DAY	DATE
DATETIME MONTH TO HOUR	DATE
DATETIME MONTH TO MINUTE	DATE
DATETIME MONTH TO SECOND	DATE
DATETIME MONTH TO FRACTION(n)	TIMESTAMP(n)
DATETIME DAY TO DAY	DATE
DATETIME DAY TO HOUR	DATE
DATETIME DAY TO MINUTE	DATE
DATETIME DAY TO SECOND	DATE
DATETIME DAY TO FRACTION(n)	TIMESTAMP(n)
DATETIME HOUR TO HOUR	DATE
DATETIME HOUR TO MINUTE	DATE
DATETIME HOUR TO SECOND	DATE
DATETIME HOUR TO FRACTION(n)	TIMESTAMP(n)
DATETIME MINUTE TO MINUTE	DATE
DATETIME MINUTE TO SECOND	DATE
DATETIME MINUTE TO FRACTION(n)	TIMESTAMP(n)
DATETIME SECOND TO SECOND	DATE
DATETIME SECOND TO FRACTION(n)	TIMESTAMP(n)
DATETIME FRACTION TO FRACTION(n)	TIMESTAMP(n)
INTERVAL YEAR[(p)] TO MONTH	INTERVAL YEAR[(p)] TO MONTH
INTERVAL MONTH[(p)] TO MONTH	CHAR(50)
INTERVAL DAY[(p)] TO FRACTION(n)	INTERVAL DAY[(p)] TO SECOND(n)
INTERVAL HOUR[(p)] TO HOUR	CHAR(50)
INTERVAL HOUR[(p)] TO MINUTE	CHAR(50)
INTERVAL HOUR[(p)] TO SECOND	CHAR(50)
INTERVAL HOUR[(p)] TO FRACTION(n)	CHAR(50)
INTERVAL MINUTE[(p)] TO MINUTE	CHAR(50)
INTERVAL MINUTE[(p)] TO SECOND	CHAR(50)
INTERVAL MINUTE[(p)] TO FRACTION(n)	CHAR(50)
INTERVAL SECOND[(p)] TO SECOND	CHAR(50)

Informix® data types	**ORACLE data types (Versions 10.x and higher)**
INTERVAL SECOND[(p)] TO FRACTION(n)	CHAR(50)
INTERVAL FRACTION[(p)] TO FRACTION	CHAR(50)

Notes:

1. For more details about serial emulation, see SERIAL data types on page 658.

Data manipulation
Oracle Database related data manipulation topics.
Reserved words

SQL object names like table and column names cannot be SQL reserved words in ORACLE.

An example of a common word which is part of the ORACLE SQL grammar is '**level**'.

Solution

Table or column names which are ORACLE reserved words must be renamed.

ORACLE reserved keywords are listed in the ORACLE documentation, or Oracle 8i provides the V $RESERVED_WORDS view to track Oracle reserved words. All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX™ 'grep' or 'awk' tools. Most modifications can be done automatically with UNIX™ tools like 'sed' or 'awk'.

Outer joins

In Informix® SQL, outer joins can be defined in the **FROM** clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER (b) WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c)) WHERE a.key = b.akey
  AND b.key1 = c.bkey1 AND b.key2 = c.bkey2
```

ORACLE expects the **(+)** operator in the join condition. You must set a (+) after columns of the tables which must have NULL values when no record matches the condition:

```
SELECT ... FROM a, b WHERE a.key = b.key (+)

SELECT ... FROM a, b, c WHERE a.key = b.akey (+)>
   AND b.key1 = c.bkey1 (+)
   AND b.key2 = c.bkey2 (+)
```

When using additional conditions on outer tables, the (+) operator also has to be used. For example:

```
SELECT ... FROM a, OUTER(b) WHERE a.key = b.akey AND b.colx > 10
```

Must be converted to:

```
SELECT ... FROM a, b WHERE a.key = b.akey (+)
   AND b.colx (+) > 10
```

The ORACLE outer joins restriction:

In a query that performs outer joins of more than two pairs of tables, a single table can only be the NULL generated table for one other table. The following case is not allowed: WHERE a.col = b.col (+) AND b.col (+) = c.col

Solution

For better SQL portability, you should use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The Oracle interface can convert most Informix® OUTER specifications to Oracle outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work: `"FROM OUTER(tab2), tab1 "`
2. The outer join in the WHERE clause must use the table name as prefix.

   Example: `"WHERE tab1.col1 = tab2.col2 "`

Restrictions:

1. Statements composed by 2 or more SELECT instructions are not supported.

   Example: `"SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"`

   Note:

   1. Table aliases are detected in OUTER expressions.

      OUTER example with table alias: "OUTER(tab1 alias1)".
   2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.

      OUTER join example with table on the left: `"WHERE outertab.col1 = maintab.col2 "`
   3. Table names detection is not case-sensitive.

      Example: `"SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2"`
   4. Temporary tables are supported in OUTER specifications.

Transactions handling

Informix® and ORACLE handle transactions differently. The differences in the transactional models can affect the program logic.

Informix® native mode (non ANSI):

- DDL statements can be executed (and canceled) in transactions.
- Transactions must be started with BEGIN WORK. Statements executed outside of a transaction are automatically committed.

ORACLE:

- Beginnings of transactions are implicit; two transactions are delimited by COMMIT or ROLLBACK.
- The current transaction is automatically committed when a DDL statement is executed.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, in accordance with the transaction model.

Informix® version 11.50 introduces savepoints with the following instructions:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] ]
RELEASE SAVEPOINT name
```

ORACLE supports savepoints too. However, there are differences:

1. Savepoints cannot be declared as UNIQUE
2. Rollback must always specify the savepoint name
3. You cannot release savepoints (RELEASE SAVEPOINT)

Solution

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with ORACLE. The Informix® behavior is simulated with an autocommit mode in the ORACLE interface. A switch to the explicit commit mode is done when a BEGIN WORK is performed by the BDL program.

When executing a DDL statement inside a transaction, ORACLE automatically commits the transaction. Therefore, you must extract the DDL statements from transaction blocks.

If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

See also SELECT FOR UPDATE

Temporary tables

Informix® temporary tables are created through the **CREATE TEMP TABLE** DDL instruction or through a **SELECT ... INTO TEMP** statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

ORACLE does not support temporary tables as Informix® does. ORACLE 8.1 provides GLOBAL TEMPORARY TABLEs which are shared among processes (only data is temporary and local to a SQL process). Informix® does not shared temp tables among SQL processes; each process can create its own temp table without table name conflicts.

Solution

In accordance with some prerequisites, temporary tables creation in BDL programs can be supported by the database interface.

The temporary table emulation can use regular tables or GLOBAL TEMPORARY tables. The way the driver converts Informix® temp table statements to Oracle regular tables or global temporary tables is driven by the following FGLPROFILE entry:

```
dbi.database.<dbname>.ifxemul.temptables.emulation = {"default" | "global" }
```

By default, the database driver uses regular tables (*default* emulation). This default emulation provides maximum compatibility with Informix® temporary tables, but requires real table creation which can be a significant overhead with Oracle. The *global* emulation uses native Oracle Global Temporary Tables, requiring only one initial table creation and thus making programs run faster. However, the *global* emulation mode has to be used carefully because of some limitations and constraints.

When creating a temporary table, you perform a Data Definition Language statement. Oracle automatically commits the current transaction when executing a DDL statement. Therefore, you must avoid temp table creation/destruction in transactions.

Using the *default* temporary table emulation

How does the *default* emulation work?

- Informix® CREATE TEMP TABLE and SELECT INTO TEMP statements are automatically converted to ORACLE "CREATE TABLE". The name of the temporary table is converted to a unique table name.
- Tables are created in the current schema.

- Temporary tables are created with the option TABLESPACE TEMPTABS so that data is stored in a dedicated tablespace named "**TEMPTABS**". Of course the TEMPTABS tablespace must exist before running programs, otherwise temporary table creation will fail. You create a tablespace with the CREATE TABLESPACE SQL command. Using a specific tablespace for temporary tables allows you to specify storage options, for example to use a physical device which can be different from the disk drive used for real data storage. Additionally, backups of permanent application tables can be performed without the data of temporary tables.
- Starting with Oracle 10g, dropped tables are saved in the recycle bin by default. You may want to avoid the recycle bin feature at the database level or session level with:

  ```
  ALTER SYSTEM SET recyclebin = OFF scope=both
  ```

  or:

  ```
  ALTER SESSION SET recyclebin = OFF
  ```

- Once the temporary table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to the corresponding unique table name.
- When the BDL program disconnects from the database (for example, when it ends or when a CLOSE DATABASE instruction is executed), the tables which have not been removed with an explicit "DROP TABLE" are automatically removed by the database interface. However, if the program crashes, the tables will remain in the database, so you may need to cleanup the database from time to time.

Prerequisites when using the *default* emulation

- Application users must have sufficient **privileges** to create database tables in their own schema (usually, "CONNECT" and "RESOURCE" roles).
- Create a dedicated tablespace named "**TEMPTABS**".

  The TEMPTABS tablespace must be of type "permanent", as it will hold permanent tables used to emulate Informix® temp tables.

  Make sure it is big enough to hold all the data, and check for automatic extension.

  When using a PDB, the TEMPTABS table space must be created in the context of the PDB.

  ```
  CREATE TABLESPACE temptabs
    DATAFILE 'file-path' SIZE 1M AUTOEXTEND ON;
  -- Give privileges on temptabs tablespace to other users
  ALTER USER dbuser QUOTA UNLIMITED ON TEMPTABS;
  ```

  For more details, see "CREATE TABLESPACE" in the Oracle documentation.

Limitations of the *default* emulation

- When using the default emulation, the real name of an emulated temporary table will have the following format:

  tt*number_original_name*

  Where <number> is the Oracle AUDSID session id returned by:

  ```
  SELECT USERENV('SESSIONID') FROM DUAL
  ```

  As Oracle 9i and 10g table names can't exceed 30 characters in length, and since session ids are persistent over server shutdown, you must pay attention to the names of your temporary tables. For example, if you create a temp table with the name TEMP_CUSTOMER_INVOICES (22c) it leaves 30 - (3 + 22) = 5 characters left for the session id, which gives a limit of 99999 sessions.

To workaround this limitation, you can provide your own SQL command to generate a unique session id with the following FGLPROFILE entry:

`dbi.database.`*`dbname`*`.ora.sid.command = "select ..."`

As an example, you can use the SID column value from V$SESSION:

```
SELECT SID FROM V$SESSION WHERE AUDSID = USERENV('SESSIONID')
```

- You are not allowed to use the unique table name format in your own database schema. Make sure you are not using table or column names with the following format:

  `tt`*`number_original_name`*
- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation:

```
CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) )
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) )
SELECT tmp1 FROM tab1 WHERE ...
```

Maintenance of *default* emulation

- If you want to list the tables created by a specific user, do this:

```
SELECT * FROM ALL_TABLES WHERE OWNER = 'user_name'
```

  As with other database object names, the user name is stored in uppercase letters if it has been created without using double quotes (`create user scott ...` = stored name is "SCOTT").

Creating indexes on temporary tables with *default* emulation

- Indexes created on temporary tables must have unique names also. The database interface detects CREATE INDEX statements which are using temporary tables and converts the index name to unique names.
- DROP INDEX statements are also detected to replace the original index name by the real name.

SERIALs in temporary table creation with *default* emulation

- You can use the SERIAL data type when creating a temporary table.

  Sequences and triggers will be created in the current schema.

  See issue about SERIALs for more details.

Using the *global* temporary table emulation

The *global* temporary table emulation is provided to get benefit of the Oracle GLOBAL TEMPORARY TABLES, by sharing the same table structure with multiple SQL sessions, reducing the cost of the CREATE TABLE statement execution. However, this emulation does not provide the same level of Informix® compatibility as the *default* emulation, and must be used carefully.

How does the *global* emulation work?

- Informix® CREATE TEMP TABLE and SELECT INTO TEMP statements are automatically converted to ORACLE "CREATE GLOBAL TEMPORARY TABLE". The original table name is kept, but it gets a "**TEMPTABS**" schema prefix, to share the underlying table structure with other database users.
- The Global Temporary Tables are created with the "ON COMMIT PRESERVE ROWS" option, to keep the rows in the table when a transaction ends.

- The Global Temporary Tables are created in a specific schema called "**TEMPTABS**". If the table exists already, error ORA-00955 will just be ignored by the database driver. This allows to do several CREATE TEMP TABLE statements in your programs with no SQL error, to emulate the Informix® behavior. This works fine as long as the table name is unique for a given structure (column count and data types must match).
- Once the Global Temporary Table has been created, all other SQL statements performed in the current SQL session are parsed to convert the original table name to TEMPTABS.*original-tablename*.
- When doing a DROP TABLE *temp-table* statement in the program, the database driver converts it to a DELETE statement, to remove all data added by the current session. A next CREATE TEMP TABLE or SELECT INTO TEMP will fail with error ORA-00955 but since this error is ignored, it will be transparent for the program. We can't use TRUNCATE TABLE because that would required at least DROP ANY TABLE privileges for all users.
- When the BDL program disconnects from the database (for example, when it ends or when a CLOSE DATABASE instruction is executed), the tables that have not been dropped by the program with an explicit DROP TABLE statement will be automatically cleaned by Oracle.

Prerequisites when using the *global* emulation

- You must create a database user (schema) dedicated to this emulation, with the name "**TEMPTABS**":

```
CREATE USER temptabs IDENTIFIED BY pswd;
```

- All database users must have sufficient privileges to use Global Temporary Tables in the **TEMPTABS** schema: If you want programs to create Global Temporary Table on the fly, you must grant a CREATE ANY TABLE + CREATE ANY INDEX system privilege to all users. But this means that all users will be able to create/drop tables in any schema (Here Oracle (10g) is missing some fine-grained system privilege to create/drop tables in a particular schema). You better "prepare" the database by creating the Global Temporary Table with the TEMPTABS user (do not forget to specify ON COMMIT PRESERVE ROWS option), and give INSERT, UPDATE, DELETE and SELECT object privileges to PUBLIC, for example:

```
CREATE GLOBAL TEMPORARY TABLE temptabs.mytable
  ( k INT PRIMARY KEY, c CHAR(10) ) ON COMMIT PRESERVE ROWS;
CREATE UNIQUE INDEX temptabs.ix1 ON temptabs.mytable ( c );
GRANT SELECT, UPDATE, INSERT, DELETE ON temptabs.mytable TO PUBLIC;
```

  For testing purpose, consider using a user with DBA privileges, to simplify the configuration.

Limitations of the *global* emulation

- Global Temporary Tables are shared by multiple users/sessions. In order to have the *global* emulation working properly with your application, each temporary table name must be unique for a given table structure, for all programs. You must for example as **tmp1**. It is recommended to use table names as follows:

```
CREATE TEMP TABLE custinfo_1 (
    cust_id INTEGER,
    cust_name VARCHAR(50)
 );
CREATE TEMP TABLE custinfo_2 (
    cust_id INTEGER,
    cust_name VARCHAR(50),
    cust_addr VARCHAR(200)
 );
```

```
CREATE TEMP TABLE custinfo_2 (
    cust_id INTEGER,
    cust_name VARCHAR(50),
```

```
    cust_addr VARCHAR(200)
);
```

- Tokens matching the original table names are converted to unique names in all SQL statements. Make sure you are not using the temp table name for other database objects, like columns. The following example illustrates this limitation:

```
CREATE TABLE tab1 ( key INTEGER, tmp1 CHAR(20) );
CREATE TEMP TABLE tmp1 ( col1 INTEGER, col2 CHAR(20) );
SELECT tmp1 FROM tab1 WHERE ...
```

Creating indexes on temporary tables with *global* emulation

- Indexes created on temporary tables get also the **TEMPTABS** schema prefix.
- When executing a DROP INDEX statement on a temporary table in a program, the database driver just ignores the statement.

SERIALs in temporary table creation with *global* emulation

- You can use the SERIAL data type when creating a temporary table.

  Sequences and triggers will be created in the **TEMPTABS** schema too.

  See issue about SERIALs for more details.

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

ORACLE provides the SUBSTR() function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTR(col1,2,2) = 'RO'
SELECT SUBSTR('Some text',6,3)FROM DUAL -- Gives 'tex'
```

Solution

You must replace all Informix® col[x,y] expressions by SUBSTR(col,x,y-x+1).

In UPDATE instructions, setting column values through subscripts will produce an error with ORACLE:

```
UPDATE tab1 SET col1[2,3]= 'RO' WHERE ...
```

is converted to:

```
UPDATE tab1 SET SUBSTR(col1,2,3-2+1)= 'RO' WHERE ...
```

The LENGTH() function

Informix® provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

Oracle has a equivalent function with the same name, but there is some difference:

Informix® does not count the trailing blanks for CHAR or VARCHAR expressions, while Oracle counts the trailing blanks.

With the Oracle LENGTH function, when using a CHAR column, values are always blank padded, so the function returns the size of the CHAR column. When using a VAR CHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

The Informix® LENGTH() function returns 0 when the given string is empty. That means, LENGTH('') is 0.

Since ORACLE handles empty strings ('') as NULL values, writing "LENGTH('')" is equivalent to "LENGTH(NULL)". In this case, the function returns NULL.

Solution

The ORACLE database interface cannot simulate the behavior of the Informix® LENGTH() function.

You must check if the trailing blanks are significant when using the LENGTH() function.

If you want to count the number of character by ignoring the trailing blanks, you must use the RTRIM() function:

```
SELECT LENGTH(RTRIM(col1)) FROM table
```

SQL conditions which verify that the result of LENGTH() is greater that a given number do not have to be changed, because the expression evaluates to false if the given string is empty (NULL>n):

```
SELECT * FROM x WHERE LENGTH(col)>0
```

Only SQL conditions that compare the result of LENGTH() to zero will not work if the column is NULL. You must check your BDL code for such conditions:

```
SELECT * FROM x WHERE LENGTH(col)=0
```

In this case, you must add a test to verify if the column is null:

```
SELECT * FROM x WHERE ( LENGTH(col)=0 OR col IS NULL )
```

In addition, when retrieving the result of a LENGTH() expression into a BDL variable, you must check that the variable is not NULL.

In ORACLE, you can use the NVL() function in order to get a non-null value:

```
SELECT * FROM x WHERE NVL(LENGTH(c),0)=0
```

Informix® Dynamic Server 7.30 supports the NVL() function, as in ORACLE. You can write the same SQL for both Informix® 7.30 and ORACLE, as shown in this example.

If the Informix® version supports stored procedures, you can create the following stored procedure in the Informix® database in order to use NVL() expressions:

```
create procedure nvl( val char(512), def char(512) )
 returning char(512);
 if val is null then return def;
 else return val;
 end if;
end procedure;
```

With this stored procedure, you can write NVL() expressions like **NVL(LENGTH(c),0)**. This should work in almost all cases and provides upward compatibility with Informix® Dynamic Server 7.30.

Empty character strings

Informix® SQL and ORACLE SQL handle empty quoted strings differently. ORACLE SQL does not distinguish between '' and NULL, while Informix® SQL treats'' (or "") as a string with a length of zero.

Using literal string values that are empty ('') for INSERT or UPDATE statements will result in the storage of NULLs with ORACLE, while Informix® would store the value as a string with a length of zero:

```
insert into tab1 ( col1, col2 ) values ( NULL, '' )
```

Using the comparison expression (col='') with ORACLE has no meaning because an empty string is equivalent to NULL; (col=NULL) expressions will always evaluate to FALSE because this is not a correct expression: The expression should be (col IS NULL).

```
select * from tab1 where col2 IS NULL
```

With Informix® 4GL and Genero BDL, when setting a variable with an empty string constant, it is automatically set to a NULL value. When using one or more space characters, the value is set to one space character:

```
define x char(10)
let x = ""
if x is null then -- evaluates to TRUE
let x = " "
if x = " " then -- evaluates to TRUE
```

Solution

The ORACLE database interface cannot automatically convert comparison expressions like (col="") to (col IS NULL) because this would require an SQL grammar parser. The interface could convert expressions like (col=""), but it would do this for the whole SQL statement:

```
UPDATE tab1 SET col1 = "" WHERE col2 = ""
```

would be converted to an incorrect SQL statement:

```
UPDATE tab1 SET col1 IS NULL WHERE col2 IS NULL
```

To increase portability, you should avoid the usage of literal string values with a length of zero in SQL statements; this would resolve storage and boolean expressions evaluation differences between Informix® and ORACLE.

NULL or program variables can be used instead. Program variables set with empty strings (let x="")are automatically converted to NULL by BDL and therefore are stored as NULL when using both Informix® or ORACLE databases.

String delimiters and object names

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'a string value'`

Informix® allows double quotes as string delimiters, but ORACLE doesn't. This is important, since many BDL programs use that character to delimit the strings in SQL commands.

This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

Solution

The ORACLE database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like the following:

```
'This is a single quote: '''
'This is a single quote: \''
"This is a double quote: """
"This is a double quote: \""
```

Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string ! For example, if the program executes the SQL statement:

`WHERE "`*`tabname`*`"."`*`colname`*`"= "a string value"`

replacing all double quotes by single quotes would produce:

`WHERE '`*`tabname`*`'.'`*`colname`*`' = 'a string value'`

This would produce an error since 'tabname'.'colname' is not allowed by ORACLE.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

Getting one row with SELECT

With Informix®, you must use the system table with a condition on the table id:

```
SELECT user FROM systables WHERE tabid=1
```

Oracle provides the **DUAL** table to generate one row only.

```
SELECT user FROM DUAL
```

Solution

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements, while ORACLE supports the LIKE statement only.

MATCHES requires * and ? wildcard characters, and LIKE uses the % and _ wildcards was equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES allows you to use brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

The LIKE operator has no operator for `[ ]` brackets character ranges.

With ORACLE, columns defined as CHAR(N) are blank padded, and trailing blanks as significant in the LIKE expressions. As result, with a CHAR(5) value such as `'abc '` (with 2 trailing blanks), the expression `(colname LIKE 'ab_')` will not match. To workaround this behavior, you can do `(RTRIM(colname) LIKE '`*`pattern`*`')`. However, consider adding the condition `AND (colname LIKE '`*`patten`*`%')` to force the DB server to optimize the query of the column is indexed. The `CONSTRUCT` instruction uses this technique when the entered criteria does not end with a * star wildcard.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

SQL functions

Almost all Informix® functions and SQL constants have a different name or behavior in ORACLE.

Here is a comparison list of functions and constants:

Table 204: SQL functions and constants (Informix® vs. Oracle)

Informix®	ORACLE
today	trunc(sysdate)
current year to second	sysdate
day(value)	to_number(to_char(value, 'dd'))
extend(dtvalue, first to last)	to_date(nvl(to_char(dtvalue, 'fmt-mask'), '19000101000000'), 'fmt-mask')
mdy(m,d,y)	to_date(to_char(m,'09') || to_char(d,'09') || to_char(y,'0009'), 'MMDDYYYY')
month(date)	to_number(to_char(date, 'mm'))
weekday(date)	to_number(to_char(date, 'd')) -1
year(date)	to_number(to_char(date, 'yyyy'))
date("string" | integer)	No equivalent - Depends from DBDATE in IFX
user	user *! Uppercase/lowercase: See* The User Constant
trim([leading | trailing | both "char" FROM] "string")	ltrim() and rtrim()
length(c)	length(c) *! Different behavior: See* The Length Function
pow(x,y)	power(x,y)

Solution

You must review the SQL statements using TODAY / CURRENT / EXTEND expressions.

You can define stored functions in the ORACLE database, to simulate Informix® functions. This works only for functions that are not already implemented by ORACLE:

```
create or replace function month( adate in date )
  return number
is
  v_month number;
begin
  v_month:= to_number( to_char( adate, 'mm' ) );
```

```
  return (v_month);
end month;
```

Querying system catalog tables

As in Informix®, ORACLE provides system catalog tables (actually, system views). But the table names and their structure are quite different.

Solution

No automatic conversion of Informix® system tables is provided by the database interface.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( <col-list> ) = ( <val-list> )
```

or

```
UPDATE table SET table.* = myrecord.*
```

```
UPDATE table SET * = myrecord.*
```

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

The USER constant

Both Informix® and ORACLE provide the **USER** constant, which identifies the current user connected to the database server.

Informix®:

```
SELECT USER FROM systables WHERE tabid=1
```

Oracle:

```
SELECT USER FROM DUAL
```

However, there is a difference:

- Informix® returns the user identifier as defined in the operating system, where it can be case-sensitive (UNIX™) or not (NT).
- ORACLE returns the user identifier which is stored in the database. By default ORACLE converts the user name to uppercase letters, if you do not put the user name in double quotes when creating it.

This is important if your application stores user names in database records (for example, to audit data modifications). You can, for example, connect to ORACLE with the name 'scott', and perform the following SQL operations:

```
(1) INSERT INTO mytab ( creator, comment )
    VALUES( USER, 'example' );
(2) SELECT * FROM mytab
    WHERE creator = 'scott';
```

The first command inserts 'SCOTT' (in uppercase letters) in the creator column. The second statement will not find the row.

Solution

When creating a user in ORACLE, you can put double quotes around the user name in order to force ORACLE to store the given user identifier as is:

```
CREATE USER "username" IDENTIFIED BY pswd
```

To verify the user names defined in the ORACLE database, connect as SYSTEM and list the records of the ALL_USERS table as follows:

```
SELECT * FROM ALL_USERS;

USERNAME     USER_ID      CREATED
------------------------------------------------------------
SYS                0      02-OCT-98
SYSTEM             5      02-OCT-98
DBSNMP            17      02-OCT-98
FBDL              20      03-OCT-98
Paul              21      03-OCT-98
```

The GROUP BY clause

Informix® allows you to use column numbers in the GROUP BY clause

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY 1
```

Oracle does not support column numbers in the GROUP BY clause.

Solution

Use column names instead:

```
SELECT ord_date, sum(ord_amount) FROM order GROUP BY ord_date
```

The star (asterisk) in SELECT statements

Informix® allows you to use the star character in the select list along with other expressions:

```
SELECT col1, * FROM tab1 ...
```

Oracle does not support this. You must use the table name as a prefix to the star:

```
SELECT col1, tab1.* FROM tab1 ...
```

Solution

Always use the table name before the star.

BDL programming

Oracle Database related programming topics.

Handling SQL errors when preparing statements

The ORACLE interface is implemented with the ORACLE Call Interface (OCI). This library does not provide a way to send SQL statements to the database server during the BDL PREPARE instruction, as in the Informix® interface. The statement is sent to the server only when opening the cursors or when executing the statement.

Therefore, when preparing an SQL statement with the BDL PREPARE instruction, no SQL errors can be returned if the statement has syntax errors, or if a column or a table name does not exist in the database. However, an SQL error will occur after the OPEN or EXECUTE instructions.

Solution

Make sure your BDL programs do not test the STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

Informix® specific-SQL statements in BDL

The BDL compiler supports several Informix-specific SQL statements that have no meaning when using ORACLE:

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution

Review your BDL source and remove all static SQL statements which are Informix® specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

ORACLE does not support insert cursors.

Solution

Insert cursors are emulated by the ORACLE database interface.

Cursors WITH HOLD

Informix® closes opened cursors automatically when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction. In ORACLE, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a transaction ends. Actually, all ORACLE cursors are 'WITH HOLD' cursors unless the FOR UPDATE clause is used in the SELECT statement.

Solution

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed.

Since ORACLE automatically closes FOR UPDATE cursors when the transaction ends, opening cursors declared FOR UPDATE and WITH HOLD results in an SQL error that does not normally appear with Informix® under the same conditions. Review the program logic in order to find another way to set locks.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to prevent several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
```

```
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

- The row must be fetched in order to set the lock.
- If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared "WITH HOLD", the lock is released when the cursor is closed.

ORACLE allows individual and exclusive row locking with:

`SELECT ... FOR UPDATE [OF col-list]`

- A lock is acquired for each selected row when the cursor is opened, before the first fetch.
- Cursors using SELECT ... FOR UPDATE are automatically closed when the transaction ends; Note that locks are **not** released **when a cursor is closed**.

ORACLE's locking granularity is at the row level.

To control the behavior of the program when locking rows, Informix® provides a specific instruction to set the wait mode:

`SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }`

The default mode is NOT WAIT. This is an Informix® specific-SQL statement.

In order to simulate the same behavior in ORACLE, your can use the NOWAIT keyword in the SELECT ... FOR UPDATE statement, as follows:

`SELECT ... FOR UPDATE [OF col-list] NOWAIT`

With this option, ORACLE immediately returns an SQL error if the row is locked by another user.

Solution

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

Cursors declared with SELECT ... FOR UPDATE using the "WITH HOLD" clause cannot be supported with ORACLE. See Cursors with Hold and UPDATE/DELETE WHERE CURRENT OF for more details.

If your BDL application uses pessimistic locking with SELECT ... FOR UPDATE, you must review the program logic for OPEN cursor and CLOSE cursor statements inside transactions (BEGIN WORK + COMMIT WORK / ROLLBACK WORK).

UPDATE/DELETE WHERE CURRENT OF

Informix® allows positioned UPDATEs and DELETEs with the "`WHERE CURRENT OF cursor`" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

UPDATE/DELETE ... WHERE CURRENT OF<cursor> is not support by the Oracle database API. However, ROWIDs can be used for positioned updates/deletes.

Solution

UPDATE/DELETE ... WHERE CURRENT OF instructions are managed by the ORACLE database interface. The ORACLE database interface replaces "WHERE CURRENT OF cursor" by "WHERE ROWID=:rid" and sets the value of the ROWID returned by the last FETCH done with the given cursor.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

ORACLE does not provide LOAD and UNLOAD instructions, but provides external tools like SQL*Plus and SQL*Loader.

Solution

In BDL programs, the LOAD and UNLOAD instructions are supported with ORACLE, with some limitations:

- There is a difference when using ORACLE DATE columns. DATE columns created in the ORACLE database are equivalent to Informix® DATETIME YEAR TO SECOND columns. In LOAD and UNLOAD, all ORACLE DATE columns are treated as Informix® DATETIME YEAR TO SECOND columns and thus will be unloaded with the "YYYY-MM-DD hh:mm:ss" format.

  The same problem appears for Informix® INTEGER and SMALLINT values, which are stored in an ORACLE database as NUMBER(?) columns. Those values will be unloaded as Informix® DECIMAL(10) and DECIMAL(5) values, that is, with a trailing dot-zero ".0".
- When using an Informix® database, simple dates are unloaded using the DBDATE format (ex:"23/12/1998"). Therefore, unloading from an Informix® database for loading into an ORACLE database is not supported.

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

Oracle supports SQL Interruption in a similar way. The db client must issue an OCIBreak() OCI call to interrupt a query.

Solution

The ORACLE database driver supports SQL interruption and converts the native SQL error code -1013 to the Informix® error code -213.

Scrollable cursors

The Genero programming language supports scrollable cursors.

Oracle 9.0 and higher support native scrollable cursors.

Solution

By default, the Oracle database driver uses native scrollable cursors by setting the OCI_STMT_SCROLLABLE_READONLY statement attribute.

See Scrollable cursors on page 422 for more details about scroll cursor emulation.

SQL adaptation guide for PostgreSQL 9.x

Installation (Runtime Configuration)
PostgreSQL related installation topics.
Install PostgreSQL and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Compile and install the PostgreSQL Server on your computer. PostgreSQL is a free database, you can download the sources from www.postgresql.org.
2. Read PostgreSQL installation notes for details about the "`data`" directory creation with the `initdb` utility.
3. Set configuration parameters in `postgresql.conf`:
   a) PostgreSQL 9.1 and higher have by default the `standard_conforming_strings` parameter set to `on`.

      The ODI drivers for PostgreSQL 9.1 and + do no longer escape the backslash characters in string literals with a second backslash.

Start a `postmaster` process to listen to database client connections.

> **Important:** If you want to connect through TCP (for example from a Windows™ PostgreSQL client), you must start postmaster with the `-i` option and setup the "`pg_hba.conf`" file for security (trusted hosts and users).

4. Create a PostgreSQL database with the `createdb` utility, by specifying the character set of the database.

```
$ createdb -h hostname dbname --encoding encoding --locale locale
```

5. If you plan to use SERIAL emulation, you need the plpgsql procedure language, because the database interface uses this language to create serial triggers.

   Starting with PostgreSQL version 9.0, the plpgsql language is available by default. Prior to version 9.0, you must create the language in your database with the following command:

```
$ createlang -h hostname plpgsql dbname
```

6. Connect to the database as the administrator user and create a database user dedicated to your application, the application administrator:

```
dbname=# CREATE USER appadmin PASSWORD 'password';
CREATE USER
dbname=# GRANT ALL PRIVILEGES ON DATABASE dbname TO appadmin;
GRANT
dbname=# \q
```

7. Create the application tables.

   Convert Informix® data types to PostgreSQL data types. See Data type conversion table: Informix to PostgreSQL on page 698 for more details.

8. If you plan to use the SERIAL emulation, you must prepare the database.

   See SERIAL data types on page 693 for more details.

Prepare the runtime environment - connecting to the database

1. In order to connect to PostgreSQL, you must have a PostgreSQL database driver "dbmpgs" in `FGLDIR/dbdrivers`.

   On HP/UX LP64, the PostgreSQL database driver must be linked with the libxnet library if you want to use networking.

2. The PostgreSQL client software is required to connect to a database server.

   Check whether the PostgreSQL client library (`libpq.*`) is installed on the machine where the BDL programs run.

3. Make sure that the PostgreSQL client environment variables are properly set.

   Check, for example, PGDIR (the path to the installation directory), PGDATA (the path to the data files directory), etc. See the PostgreSQL documentation for more details.

4. Check the database client locale settings (for example, set the PGCLIENTENCODING environment variable).

   The database client locale must match the locale used by the runtime system (LC_ALL, LANG).

5. Verify the environment variable defining the search path for the PostgreSQL database client shared libraries (libpq.so on UNIX™, LIBPQ.DLL on Windows™).

Table 205: Shared library environment setting for PostgreSQL

PostgreSQL version	Shared library environment setting
PostgreSQL 9.0 and higher	*UNIX*™: Add $PGDIR/lib to LD_LIBRARY_PATH (or its equivalent). *Windows*™: Add %PGDIR%\bin to PATH.

6. To verify if the PostgreSQL client environment is correct, you can start the PostgreSQL command interpreter:

```
$ psql dbname -U appadmin -W
```

7. Set up the fglprofile entries for database connections.
   a) Define the PostgreSQL database driver:

```
dbi.database.dbname.driver = "dbmpgs"
```

   b) The '**source**' parameter defines the name of the PostgreSQL database, as well as additional connection parameters if needed, such as the server host name, the TCP port and specific PostgresSQL connection options.

```
dbi.database.dbname.source = "test1"
```

   The `source` parameter must have the following form:

```
dbname[@host[:port]][?options]
```

   where:

   - *dbname* defines the name of the PostgreSQL database
   - *host* defines the server host name, or IP address (IPv6 host address needs to be enclosed it in square brackets)
   - *port* defines the TCP port
   - *options* is a URI-style query string defining PostgreSQL connection parameters

   For example:

```
mydb@orion:5433?connect_timeout=10&application_name=myapp
```

Database concepts
PostgreSQL related database concepts topics.
Database concepts

Like Informix® servers, PostgreSQL can handle multiple database entities. Tables created by a user can be accessed without the owner prefix by other users as long as they have access privileges to these tables.

Solution

Create a PostgreSQL database for each Informix® database.

Data storage concepts

An attempt should be made to preserve as much of the storage information as possible when converting from Informix® to PostgreSQL. Most important storage decisions made for Informix® database objects (like initial sizes and physical placement) can be reused for the PostgreSQL database.

Storage concepts are quite similar in Informix® and in PostgreSQL, but the names are different.

Data consistency and concurrency

Data consistency involves readers that want to access data currently modified by writers, and *concurrency data access* involves several writers accessing the same data for modification. **Locking granularity** defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®

Informix® uses a locking mechanism to handle data consistency and concurrency. When a process changes database information with UPDATE, INSERT or DELETE, an **exclusive lock** is set on the touched rows. The lock remains active until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Lock wait mode: SET LOCK MODE TO ...
- Isolation level: SET ISOLATION TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is per page.

PostgreSQL

When data is modified, **exclusive locks** are set and held until the end of the transaction. For data consistency, PostgreSQL uses a **multi-version consistency model**: A copy of the original row is kept for readers before performing writer modifications. Readers do not have to wait for writers as in Informix®. The simplest way to think of the PostgreSQL implementation of read consistency is to imagine each user operating a private copy of the database, hence the multi-version consistency model. The **lock wait** mode cannot be changed as in Informix®. Locks are set at the **row level** in PostgreSQL and this cannot be changed.

Control:

- No lock wait mode control is provided.
- Isolation level: SET TRANSACTION ISOLATION LEVEL ...
- Explicit exclusive lock: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed.

The main difference between Informix® and PostgreSQL is that readers do not have to wait for writers in PostgreSQL.

Solution

The SET ISOLATION TO ... Informix® syntax is replaced by SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL ... in PostgreSQL. The next table shows the isolation level mappings done by the PostgreSQL database driver:

Table 206: Isolation level mappings done by the PostgreSQL database driver

SET ISOLATION instruction in program	Native SQL command
SET ISOLATION TO DIRTY READ	SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED
SET ISOLATION TO CURSOR STABILITY	SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED
SET ISOLATION TO REPEATABLE READ	SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE

For portability, it is recommended that you work with Informix® in the read committed isolation level, make processes wait for each other (lock mode wait), and create tables with the "lock mode row" option.

See the Informix® and PostgreSQL documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Informix® and PostgreSQL handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

PostgreSQL supports transaction with savepoints:

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be placed with SAVEPOINT *name* .
- Transactions can be rolled back to a savepoint with ROLLBACK TO SAVEPOINT *name* .
- Savepoints can be released with RELEASE SAVEPOINT *name* .
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, according to the transaction model.

The main difference between Informix® and PostgreSQL resides in the fact that PostgreSQL cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
INSERT INTO tab1 ( 1, 'abc' )
    -- PK constraint violation = SQL Error, whole TX is aborted
COMMIT WORK
```

With Informix®, this code will leave the table with one row inside, since the first INSERT statement succeeded. With PostgreSQL, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction. To workaround this problem in PostgreSQL you can use SAVEPOINT as described in Solution on page 688.

Solution

Informix® transaction handling commands are automatically converted to PostgreSQL instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with PostgreSQL.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. The SQL statements that can potentially raise an SQL error must be protected with a SAVEPOINT. If an error occurs, just rollback to the savepoint:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
CALL sql_protect()
INSERT INTO tab1 ( 1, 'abc' )
 -- PK constraint violation = SQL Error
CALL sql_unprotect()
COMMIT WORK
...
FUNCTION sql_protect()
  IF NOT dbtype == "PGS" THEN
     RETURN
  END IF
  SAVEPOINT _sql_protect_
END FUNCTION

FUNCTION sql_unprotect()
  IF NOT dbtype == "PGS" THEN
     RETURN
  END IF
  IF SQLCA.SQLCODE < 0 THEN
     ROLLBACK TO SAVEPOINT _sql_protect_
  ELSE
     RELEASE SAVEPOINT _sql_protect_
  END IF
END FUNCTION
```

> **Note:** If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Database users

Until version 11.70.xC2, Informix® database users had to be created at the operating system level and be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as in most other db servers. Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

PostgreSQL users must be registered in the database. They are created by the **createuser** utility:

```
$ createuser --username=username --password
```

Solution

Based on the application logic (is it a multiuser application ?), you have to create one or several PostgreSQL users.

Setting privileges

Informix® and PostgreSQL user privileges management are quite similar.

PostgreSQL provides **user groups** to grant or revoke permissions to more than one user at the same time.

Data dictionary

PostgreSQL related data dictionary topics.

BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way; as in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

PostgreSQL supports the BOOLEAN data type and stores 't' or 'f' values for TRUE and FALSE representation. It is not possible to insert the integer values 1 or 0; values must be true, false, '1' or '0'.

Solution

The PostgreSQL database interface supports the BOOLEAN data type, and converts the BDL BOOLEAN integer values to a CHAR(1) of 't' or 'f'.

CHARACTER data types

Informix® supports the following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N)is expressed in bytes, not characters as in some other databases)

PostgreSQL provides the following character types:

- CHAR(N) with N<= 10485760 characters
- VARCHAR(N) with N<= 10485760 characters; The length specification is optional.
- TEXT with a limit of 1GB

In PostgreSQL, CHAR, VARCHAR and TEXT types store data in single byte or multibyte character sets. For CHAR and VARCHAR, the size is specified in a number of characters, not bytes. The character set used to store data for these types is defined by the database character set, which can be specified when you create the database with the *createdb* tool or the CREATE DATABASE SQL command.

> **Note:** The VARCHAR type of PostgreSQL can be used without a length specification. If no size is specified, the column accepts strings of any size. However, as Genero BDL needs to know the size of CHAR and VARCHAR columns to define fields and program variables from a schema file, you should not create tables in PostgreSQL having VARCHAR columns without size specification. If you try to extract a schema with fgldbsch, this tool will report that the VARCHAR column cannot be converted to a BDL type for the .sch file.

Automatic character set conversion between the PostgreSQL client and server is supported. You must properly specify the client character set for PostgreSQL. This can be done in different ways, with the SET CLIENT_ENCODING TO SQL command for example, or with configuration parameters. See the PostgreSQL documentation for more details.

Solution

Informix® CHAR(N) types must be mapped to PostgreSQL CHAR(N) types, and Informix® VARCHAR(N) or LVARCHAR(N) columns must be mapped to PostgreSQL VARCHAR(N).

> **Note:** When creating a table from the BDL program with NCHAR or NVARCHAR types, the type names will be left as is and produce an SQL error because these types are not supported by PostgreSQL.

You can store single-byte or multibyte character strings in PostgreSQL CHAR, VARCHAR and TEXT columns.

PostgreSQL uses character length semantics: When you define a CHAR(20) and the database character set is multibyte, the column can hold more bytes/characters than the Informix® CHAR(20) type, when using byte length semantics.

When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a PostgreSQL database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in PostgreSQL database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

See also the section about Localization.

NUMERIC data types

Informix® supports several data types to store numbers:

Table 207: Informix® numeric data types

Informix® data type	Description
SMALLINT	16 bit signed integer
INT / INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
INT8	64 bit signed integer (replaced by BIGINT)
DEC / DECIMAL	Equivalent to DECIMAL(16)

Informix® data type	Description
DEC / DECIMAL(p)	Floating-point decimal number
DEC / DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL / SMALLFLOAT	32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)]	64-bit floating point decimal (C double)

Solution

PostgreSQL supports the following data types to store numbers:

Table 208: PostgreSQL numeric data types

PostgreSQL data type	Description
NUMERIC(p,s) / DECIMAL(p,s)	Decimals with precision and scale (fractional part)
NUMERIC(p) / DECIMAL(p)	Integers with p digits (no fractional part)
NUMERIC / DECIMAL	Floating point numbers (no limit)
FLOAT4	16 bit variable precision
FLOAT8	32 bit variable precision
INT2	16 bit signed integer
INT4	32 bit signed integer
INT8/BIGINT	64 bit signed integer

ANSI types like SMALLINT, INTEGER, FLOAT are supported by PostgreSQL as aliases to INT2, INT4 and FLOAT8 native types.

Informix® DECIMAL(p) floating point types are converted to DECIMAL without precision/scale, to store any floating point number in PostgreSQL.

DATE and DATETIME data types

Informix® provides two data types to store dates and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

PostgreSQL provides the following data type to store date and time information:

- DATE = for year, month, day storage.
- TIME [(p)] [{with|without} time zone] = for hour, minute, second and fraction of second storage.
- TIMESTAMP [(p)] [{with|without} time zone] = for year, month, day, hour, minute, second and fraction of second storage.

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in Informix®, PostgreSQL can convert quoted

strings to date time data according to the DateStyle session parameter. PostgreSQL always accepts ISO date time strings.

Date arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used, and an INTERVAL value if a DATETIME is used in the expression.
- In PostgreSQL, the result of an arithmetic expression involving DATE values is an INTEGER representing a number of days.
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. PostgreSQL does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix-specific and have no equivalent in PostgreSQL.

Solution

The DATE type of PostgreSQL is equivalent to the DATE type in Informix® (stores year, month, day). Use PostgreSQL DATE data type for Informix® DATE columns.

PostgreSQL TIME(N) WITHOUT TIME ZONE data type can be used to store DATETIME HOUR TO ??? values.

PostgreSQL TIMESTAMP(N) WITHOUT TIME ZONE data type can be used to store DATETIME YEAR TO ??? values.

The SQL Translator of the PostgreSQL driver makes the following conversions automatically for the DATETIME types:

- DATETIME HOUR TO MINUTE is converted to PostgreSQL TIME(0) WITHOUT TIME ZONE (seconds set to 00).
- DATETIME HOUR TO SECOND is converted to PostgreSQL TIME(0) WITHOUT TIME ZONE.
- DATETIME HOUR TO FRACTION(N) is converted to PostgreSQL TIME(N) WITHOUT TIME ZONE.
- DATETIME YEAR TO MINUTE is converted to PostgreSQL TIMSTAMP(0) WITHOUT TIME ZONE (seconds set to 00).
- DATETIME YEAR TO SECOND is converted to PostgreSQL TIMESTAMP(0) WITHOUT TIME ZONE.
- DATETIME YEAR TO FRACTION(N) is converted to PostgreSQL TIMESTAMP(N) WITHOUT TIME ZONE.

Other DATETIME types will be mapped to PostgreSQL TIMESTAMP(N) types. Missing date or time parts default to 1900-01-01 00:00:00.

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

The Informix® INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

Starting with version 8.4, PostgreSQL provides an INTERVAL data type which is equivalent to the Informix® INTERVAL type. The following are some features of the PostgreSQL 8.4 interval type:

- It is possible to specify the interval class / precision with YEAR, MONTH, DAY, HOUR, MINUTE and SECOND[(p)] fields.
- Fractional part of seconds can be defined with up to 6 digits.
- The INTERVALs value range is from -178000000 to +178000000 years.
- Input and output format can be controlled with the SET interval style command.

Solution

Starting with Genero 2.21, database drivers dbmpgs84x and higher convert the Informix-style INTERVAL type to the native PostgreSQL INTERVAL type. See the data type conversion table for the exact conversion rules.

> **Important:** The PostgreSQL database driver forces the interval style session parameter to 'iso_8601', this is required to insert and fetch interval database with the libpq CAPI functions. You must not change this setting during program execution.

While PostgreSQL INTERVALs support up to 9 digits for the higher unit like Informix®, YEAR values range from -178000000 to +178000000 only. This limitation exists in PostgreSQL 8.4 and maybe solved in future versions.

With PostgreSQL and driver versions prior to 8.4, the INTERVAL data type is converted to CHAR(50).

SERIAL data types

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in `SQLCA.SQLERRD[2]`, while the new SERIAL8 and BIGSERIAL value must be fetched with a `SELECT dbinfo('bigserial')` query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

PostgreSQL SERIAL data type:

- PostgreSQL's SERIAL data type has the same name as in Informix®, but it behaves differently.
- You cannot define a start value (SERIAL(100)).
- You cannot specify zero as serial value to get a new serial, the PostgreSQL serial is based on default values, thus you must omit the serial column in the INSERT statement.
- When you INSERT a row with a specific value for the serial column, the underlying sequence will not be incremented. As result, the next INSERT that does not specify the serial column may get a new sequence that was already inserted explicitly.
- With some old versions of PostgreSQL, when you drop the table you must drop the sequence too.

PostgreSQL sequences:

- Sequences are totally detached from tables.
- The purpose of sequences is to provide unique integer numbers.
- Sequences are identified by a sequence name.
- To create a sequence, you must use the CREATE SEQUENCE statement.

  Once a sequence is created, it is permanent (like a table).
- To get a new sequence value, you must use the nextval() function:

```
INSERT INTO tab1 VALUES ( nextval('tab1_seq'), ... )
```

- To get the last generated number, PostgreSQL provides the currval() function:

```
SELECT currval('tab1_seq')
```

Solution

The Informix® SERIAL data type can be emulated with three different methods.

The method used to emulate SERIAL types is defined by the `ifxemul.datatype.serial.emulation` FGLPROFILE parameter:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial.emulation = {"native"|"regtable"|"trigseq"}`

- native: uses the native PostgreSQL serial data type.
- regtable: uses insert triggers with the SERIALREG table.
- trigseq: uses insert triggers with sequences.

The default emulation technique is "`native`".

This entry must be used in conjonction with:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial = {true|false}`

If the `datatype.serial` entry is set to false, the emulation method is ignored.

Using the `native` serial emulation

The "`native`" mode is the default serial emulation mode, using the native PostgreSQL SERIAL data type. In this mode, the original type name will be left untouched by the SQL Translator and you will get the behavior of the PostgreSQL SERIAL column type, based on sequences.

> **Note:** INSERT statements cannot use the serial column, even with a value zero. When using a NULL value, PostgreSQL will report an non-null constraint error. Therefore, the serial column must be omitted from the INSERT statement.

The `sqlca.sqlerrd[2]` register is not set after an INSERT when using a PostgreSQL version prior to version 8.3.

See also the PostgreSQL documentation for more details about the native SERIAL type.

Using the `regtable` serial emulation

With the "`regtable`" mode, the SERIAL data type is emulated with a PostgreSQL INTEGER data type and INSERT triggers using the table SERIALREG which is dedicated to sequence production. After an insert, `sqlca.sqlerrd[2]` register holds the last generated serial value. BIGSERIAL and SERIAL8 types can be converted to BIGINT in PostgreSQL, but the `sqlca.sqlerrd[2]` register cannot be used since it is defined as an INTEGER type.

The triggers can be created manually during the application database installation procedure, or automatically from a BDL program: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL data type to INTEGER and dynamically creates the triggers.

You must create the SERIALREG table as follows:

```
CREATE TABLE SERIALREG (
   TABLENAME VARCHAR(50) NOT NULL,
   LASTSERIAL DECIMAL(20,0) NOT NULL,
   PRIMARY KEY ( TABLENAME )
)
```

> **Important:** The SERIALREG table must be created before the triggers. The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

With this emulation mode, INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements that do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value. Consider removing the serial column from the INSERT statements.

Using the `trigseq` serial emulation

With "`trigseq`", the SERIAL data type is emulated with a PostgreSQL INTEGER data type and INSERT triggers using a sequence *`tablename`*`_seq`. After an insert, `sqlca.sqlerrd[2]` register holds the last generated serial value.

The triggers can be created manually during the application database installation procedure, or automatically from a BDL program: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the SERIAL data type to INTEGER and dynamically creates the triggers.

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

With this emulation mode, INSERT statements using NULL for the SERIAL column will produce a new serial value:

```
INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col2) VALUES ('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value. Consider removing the serial column from the INSERT statements.

Notes common to all serial emulation modes

Since `sqlca.sqlerrd[2]` is defined as an INTEGER, it cannot hold values from BIGSERIAL (BIGINT) auto incremented columns. If you are using BIGSERIAL columns, you must query the sequence pseudo-column CURRVAL() or fetch the LASTSERIAL column from the SERIALREG table if used.

For SQL portability, INSERT statements should be reviewed to remove the SERIAL column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES ( 0 , p_value )
```

can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.*) -- will use the serial column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

> **Important:** When using the Static SQL INSERT and UPDATE syntax using record.* without braces, make sure that you database schema files contain information about serials: This information can be lost when extracting the schema from a PostgreSQL database which does not use native serial emulation. See Database Schema for more details about the serial flag in column type encoding (data type code must be 6)

ROWIDs

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

When the feature is enabled, PostgreSQL tables are automatically created with a OID column (Object Identifier) of type INTEGER. The behavior is equivalent to Informix® ROWID columns (see Solution).

Solution

The database automatically converts ROWID keywords to OID for PostgreSQL. You can execute "SELECT ROWID FROM" and "UPDATE .. WHERE ROWID = ?" statements as with Informix®.

> **Note:**
>
> - Starting with PostgreSQL version 8.1, OIDs are no longer supported by default. You need to define the **default_with_oid** parameter in postgresql.conf to get OID columns created for tables. See Database configuration and design tasks.
> - SQLCA.SQLERRD[6]is not supported. All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the PostgreSQL interface.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

PostgreSQL provides the TEXT and BYTEA data types for large objects storage. With these data types, large objects are handled as a whole. In fact PostgreSQL does also provide another way to store blobs, through the large objects facility based on stream-style access. The large object facility is provided as a set of C and SQL API functions to create / delete / modify large objects identified by a unique object id (OID). For example, the `lo_create(-1)` SQL function will create a new large object and return a new object id that will be used to handle the LOB. See PostgreSQL documentation for more details.

Solution

TEXT and BYTE data can be stored in PostgreSQL TEXT and BYTEA columns.

Genero BDL does not interface automatically with the PostgreSQL Large Object facility. However, the OID values can be stored in BIGINT variables, and you can use server-side LOB functions to convert large objects to BYTEA data, that can be fetched into BYTE variables. The next code example creates a table with an OID column, imports a LOB from an image file, and then fetches the LOB back into a BYTE:

```
MAIN
    DEFINE img BYTE, obj_id BIGINT
```

```
    CONNECT TO "test1+driver='dbmpgs'" USER "postgres" USING "fourjs"

    # Need superuser privileges to create the LOB....
    WHENEVER ERROR CONTINUE
    DROP TABLE t1
    WHENEVER ERROR STOP
    EXECUTE IMMEDIATE "create table t1 ( k int, image oid )"
    GRANT SELECT ON t1 TO PUBLIC
    INSERT INTO t1 VALUES ( 1, lo_import("/var/images/landscape.png") )
    SELECT image INTO obj_id FROM t1 WHERE k=1
    DISPLAY "obj_id = ", obj_id
    EXECUTE IMMEDIATE "grant select on large object "||obj_id||" to public"

    # Next block can be executed by any user:
    LOCATE img IN FILE -- a temp file will be used
    SELECT loread(lo_open(image, 262144), 1000000)
      INTO img FROM t1 WHERE k=1
    DISPLAY length(img)

    # Delete the object
    SELECT lo_unlink(obj_id) FROM t1 WHERE k=1

    DROP TABLE t1

END MAIN
```

Constraints

Constraint naming syntax

Both Informix® and PostgreSQL support primary key, unique, foreign key, default and check constraints, but the constraint naming syntax is different. PostgreSQL expects the "CONSTRAINT" keyword **before** the constraint specification and Informix® expects it **after**.

UNIQUE constraint example

Table 209: UNIQUE constraint example (Informix® vs. PostgreSQL)

Informix®	PostgreSQL

`CREATE TABLE emp (`
`  ...`
`  emp_code CHAR(10)`
`  UNIQUE CONSTRAINT pk_emp,`
`  ...`
	`CREATE TABLE emp (`
	`  ...`
	`  emp_code CHAR(10)`
	`  CONSTRAINT pk_emp UNIQUE,`
	`  ...`

Unique constraints

> **Note:** When using a unique constraint, Informix® allows only one row with a NULL value, while PostgreSQL allows several rows with NULL!

Solution

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for PostgreSQL.

Triggers

Informix® and PostgreSQL provide triggers with similar features, but the trigger creation syntax and the programming languages are totally different.

Solution

Informix® triggers must be converted to PostgreSQL triggers "by hand".

Stored procedures

Both Informix® and PostgreSQL support stored procedures, but the programming languages are totally different. With PostgreSQL you must create the stored procedure language before writing triggers or stored procedures.

Solution

Informix® stored procedures must be converted to PostgreSQL manually.

See SQL Programming for more details about executing stored procedures with PostgreSQL.

Name resolution of SQL objects

Informix® uses the following form to identify an SQL object:

`[`*`database`*`[@`*`dbservername`*`]:][{`*`owner`*`|"`*`owner`*`"}.]`*`identifier`*

With PostgreSQL, an object name takes the following form:

`[`*`owner`*`.]`*`identifier`*

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

Data type conversion table: Informix to PostgreSQL

Table 210: Data type conversion table (Informix to PostgreSQL)

Informix® data types	PostgreSQL data types (before 8.4)	PostgreSQL data types (since 8.4)
CHAR(n)	CHAR(n)	CHAR(n)
VARCHAR(n[,m])	VARCHAR(n)	VARCHAR(n)
LVARCHAR(n[,m])	VARCHAR(n)	VARCHAR(n)
NCHAR(n)	N/A	N/A
NVARCHAR(n[,m])	N/A	N/A
BOOLEAN	BOOLEAN	BOOLEAN
SMALLINT	INT2	INT2
INT / INTEGER	INT4	INT4
BIGINT	BIGINT	BIGINT
INT8	BIGINT	BIGINT
SERIAL[(start)]	INTEGER (see note 1)	INTEGER (see note 1)
BIGSERIAL[(start)]	BIGINT (see note 1)	BIGINT (see note 1)

Informix® data types	**PostgreSQL data types (before 8.4)**	**PostgreSQL data types (since 8.4)**
SERIAL8[(start)]	BIGINT (see note 1)	BIGINT (see note 1)
DOUBLE PRECISION / FLOAT[(n)]	FLOAT4	FLOAT4
REAL / SMALLFLOAT	FLOAT8	FLOAT8
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL(p)	DECIMAL (no precision = floating point)	DECIMAL (no precision = floating point)
NUMERIC / DEC / DECIMAL	DECIMAL	DECIMAL
MONEY(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
MONEY(p)	DECIMAL(p,2)	DECIMAL(p,2)
MONEY	DECIMAL(16,2)	DECIMAL(16,2)
DATE	DATE	DATE
DATETIME HOUR TO MINUTE	TIME(0) WITHOUT TIME ZONE	TIME(0) WITHOUT TIME ZONE
DATETIME HOUR TO SECOND	TIME(0) WITHOUT TIME ZONE	TIME(0) WITHOUT TIME ZONE
DATETIME HOUR TO FRACTION(p)	TIME(p) WITHOUT TIME ZONE	TIME(p) WITHOUT TIME ZONE
DATETIME YEAR TO MINUTE	TIMESTAMP(0) WITHOUT TIME ZONE	TIMESTAMP(0) WITHOUT TIME ZONE
DATETIME YEAR TO SECOND	TIMESTAMP(0) WITHOUT TIME ZONE	TIMESTAMP(0) WITHOUT TIME ZONE
DATETIME YEAR TO FRACTION(p)	TIMESTAMP(p) WITHOUT TIME ZONE	TIMESTAMP(p) WITHOUT TIME ZONE
DATETIME q1 TO q2 (other than above)	TIMESTAMP(p) WITHOUT TIME ZONE	TIMESTAMP(p) WITHOUT TIME ZONE
INTERVAL YEAR[(p)] TO MONTH	CHAR(50)	INTERVAL YEAR TO MONTH
INTERVAL YEAR[(p)] TO YEAR	CHAR(50)	INTERVAL YEAR
INTERVAL MONTH[(p)] TO MONTH	CHAR(50)	INTERVAL MONTH
INTERVAL DAY[(p)] TO FRACTION(n)	CHAR(50)	INTERVAL DAY TO SECOND(n)
INTERVAL DAY[(p)] TO SECOND	CHAR(50)	INTERVAL DAY TO SECOND(0)
INTERVAL DAY[(p)] TO MINUTE	CHAR(50)	INTERVAL DAY TO MINUTE
INTERVAL DAY[(p)] TO HOUR	CHAR(50)	INTERVAL DAY TO HOUR
INTERVAL DAY[(p)] TO DAY	CHAR(50)	INTERVAL DAY
INTERVAL HOUR[(p)] TO FRACTION(n)	CHAR(50)	INTERVAL HOUR TO SECOND(n)

Informix® data types	**PostgreSQL data types (before 8.4)**	**PostgreSQL data types (since 8.4)**
INTERVAL HOUR[(p)] TO SECOND	CHAR(50)	INTERVAL HOUR TO SECOND(0)
INTERVAL HOUR[(p)] TO MINUTE	CHAR(50)	INTERVAL HOUR TO MINUTE
INTERVAL HOUR[(p)] TO HOUR	CHAR(50)	INTERVAL HOUR
INTERVAL MINUTE[(p)] TO FRACTION(n)	CHAR(50)	INTERVAL MINUTE TO SECOND(n)
INTERVAL MINUTE[(p)] TO SECOND	CHAR(50)	INTERVAL MINUTE TO SECOND(0)
INTERVAL MINUTE[(p)] TO MINUTE	CHAR(50)	INTERVAL MINUTE
INTERVAL SECOND[(p)] TO FRACTION(n)	CHAR(50)	INTERVAL SECOND(n)
INTERVAL SECOND[(p)] TO SECOND	CHAR(50)	INTERVAL SECOND(0)
INTERVAL FRACTION[(p)] TO FRACTION(n)	CHAR(50)	INTERVAL SECOND(n)
TEXT	TEXT	TEXT
BYTE	BYTEA	BYTEA

Notes:

1. For more details about serial emulation, see SERIAL data types on page 693.

Data manipulation
PostgreSQL related data manipulation topics.
Reserved words

SQL object names like table and column names cannot be SQL reserved words in PostgreSQL.

Solution

Table or column names which are PostgreSQL reserved words must be renamed.

Outer joins

In Informix® SQL, outer tables can be defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER(b)
 WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
 WHERE a.key = b.akey
  AND b.key1 = c.bkey1
   AND b.key2 = c.bkey2
```

PostgreSQL supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
```

```
 ON cust.key = order.custno
```

```
SELECT ...
 FROM cust LEFT OUTER JOIN order
  LEFT OUTER JOIN item
  ON order.key = item.ordno
 ON cust.key = order.custno
WHERE order.cdate > current date
```

See the PostgreSQL reference for a complete description of the syntax.

Solution

For better SQL portability, use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The PostgreSQL interface can convert most Informix® OUTER specifications to ANSI outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work: " `FROM OUTER(tab2), tab1`".
2. The outer join in the WHERE part must use the table name as prefix.

   Example: " `WHERE tab1.col1 = tab2.col2`".

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   Example: "... `FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10`".
2. Statements composed of 2 or more SELECT instructions using OUTERs are not supported.

   Example: " `SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"`

Remarks:

1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias: " `OUTER( tab1 alias1)`".
2. In the outer join, *`outertab.col`* can be placed on both right or left sides of the equal sign.

   OUTER join example with table on the left: " `WHERE outertab.col1 = maintab.col2`".
3. Table names detection is not case-sensitive.

   Example: " `SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2`".
4. Temporary tables are supported in OUTER specifications.

Transactions handling

Informix® and PostgreSQL handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be set with SAVEPOINT *name* [UNIQUE].
- Transactions can be rolled back to a savepoint with ROLLBACK [WORK] TO SAVEPOINT [*name*].
- Savepoints can be released with RELEASE SAVEPOINT *name*.
- Statements executed outside of a transaction are automatically committed.

- DDL statements can be executed (and canceled) in transactions.

PostgreSQL supports transaction with savepoints:

- Transactions are started with BEGIN WORK.
- Transactions are validated with COMMIT WORK.
- Transactions are canceled with ROLLBACK WORK.
- Savepoints can be placed with SAVEPOINT *name* .
- Transactions can be rolled back to a savepoint with ROLLBACK TO SAVEPOINT *name* .
- Savepoints can be released with RELEASE SAVEPOINT *name* .
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.
- If an SQL error occurs in a transaction, the whole transaction is aborted.

Transactions in stored procedures: avoid using transactions in stored procedures to allow the client applications to handle transactions, according to the transaction model.

The main difference between Informix® and PostgreSQL resides in the fact that PostgreSQL cancels the entire transaction if an SQL error occurs in one of the statements executed inside the transaction. The following code example illustrates this difference:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
INSERT INTO tab1 ( 1, 'abc' )
    -- PK constraint violation = SQL Error, whole TX is aborted
COMMIT WORK
```

With Informix®, this code will leave the table with one row inside, since the first INSERT statement succeeded. With PostgreSQL, the table will remain empty after executing this piece of code, because the server will rollback the whole transaction. To workaround this problem in PostgreSQL you can use SAVEPOINT as described in Solution on page 702.

Solution

Informix® transaction handling commands are automatically converted to PostgreSQL instructions to start, validate or cancel transactions.

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with PostgreSQL.

You must review the SQL statements inside BEGIN WORK / COMMIT WORK instruction and check if these can raise an SQL error. The SQL statements that can potentially raise an SQL error must be protected with a SAVEPOINT. If an error occurs, just rollback to the savepoint:

```
CREATE TABLE tab1 ( k INT PRIMARY KEY, c CHAR(10) )
WHENEVER ERROR CONTINUE
BEGIN WORK
INSERT INTO tab1 ( 1, 'abc' )
CALL sql_protect()
INSERT INTO tab1 ( 1, 'abc' )
 -- PK constraint violation = SQL Error
CALL sql_unprotect()
COMMIT WORK
...
FUNCTION sql_protect()
  IF NOT dbtype == "PGS" THEN
     RETURN
  END IF
  SAVEPOINT _sql_protect_
```

```
END FUNCTION

FUNCTION sql_unprotect()
  IF NOT dbtype == "PGS" THEN
     RETURN
  END IF
  IF SQLCA.SQLCODE < 0 THEN
     ROLLBACK TO SAVEPOINT _sql_protect_
  ELSE
     RELEASE SAVEPOINT _sql_protect_
  END IF
END FUNCTION
```

Note: If you want to use savepoints, do not use the UNIQUE keyword in the savepoint declaration, always specify the savepoint name in ROLLBACK TO SAVEPOINT, and do not drop savepoints with RELEASE SAVEPOINT.

Temporary tables

Informix® temporary tables are created through the CREATE TEMP TABLE DDL instruction or through a SELECT ... INTO TEMP statement. Temporary tables are automatically dropped when the SQL session ends, but they can be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

PostgreSQL support temporary tables as Informix® does, with a little syntax difference in the SELECT INTO TEMP instruction.

Solution

Temporary tables are well supported with native PostgreSQL temp tables.

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY col1[1,3]
```

PostgreSQL provides the SUBSTRING() function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE SUBSTRING(col1 from 2 for 2) = 'RO'
SELECT SUBSTRING('Some text' from 6 for 3) ... -- Gives 'tex'
```

Solution

You must replace all Informix® col[x,y] expressions by SUBSTRING(col from x for (y-x+1)).

Note:

- In UPDATE instructions, setting column values through subscripts will produce an error with PostgreSQL:

  `UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...`

  is converted to:

  `UPDATE tab1 SET SUBSTRING(col1 from 2 for (3-2+1)) = 'RO' WHERE ...`

- Column subscripts in ORDER BY expressions are also converted and produce an error with PostgreSQL:

  ```
  SELECT ... FROM tab1 ORDER BY col1[1,3]
  ```

  is converted to:

  ```
  SELECT ... FROM tab1 ORDER BY SUBSTRING(col1 from 1 for(3-1+1))
  ```

String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE "tabname"."colname" = 'string'`

Informix® allows double quotes as string delimiters, but PostgreSQL doesn't. This is important since many BDL programs use that character to delimit the strings in SQL commands.

> **Note:** This problem concerns only double quotes within SQL statements. Double quotes used in pure BDL string expressions are not subject to SQL compatibility problems.

Solution

The PostgreSQL database interface can automatically replace all double quotes by single quotes.

Escaped string delimiters can be used inside strings like following:

```
'This is a single quote: '''
'This is a single quote: \''
"This is a double quote: """
"This is a double quote: \""
```

Database object names cannot be delimited by double quotes because the database interface cannot determine the difference between a database object name and a quoted string. For example, if the program executes the SQL statement:

`WHERE "tabname"."colname" = "string"`

replacing all double quotes by single quotes would produce:

`WHERE 'tabname'.'colname' = 'string'`

This would produce an error since `'tabname'.'colname'` is not allowed by PostgreSQL.

Although double quotes are replaced automatically in SQL statements, you should use only single quotes to enforce portability.

Using column aliases in SELECT

PostgreSQL expects the ANSI notation for column aliases:

```
SELECT col1 AS col1_alias FROM ...
```

Informix® supports the ANSI notation.

Solution

The database interface cannot convert Informix® alias specification to the ANSI notation.

Review your programs and replace the Informix® notation with the ANSI form.

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements. PostgreSQL supports the LIKE statement as in Informix®, plus the ~ operators that are similar but different from the Informix® MATCHES operator.

MATCHES requires * and ? wildcard characters, and LIKE uses the % and _ wildcards was equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES allows brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

The PostgreSQL LIKE operator has no operator for `[ ]` brackets character ranges.

The PostgreSQL ~ operator expects regular expressions as follows: `( col ~ 'a.*' )`

With PostgreSQL, columns defined as CHAR(N) are blank padded, and trailing blanks are significant in the LIKE expressions. As result, with a CHAR(5) value such as `'abc '` (with 2 trailing blanks), the expression `(colname LIKE 'ab_')` will not match. To workaround this behavior, you can do `(RTRIM(colname) LIKE '`*`pattern`*`')`. However, consider adding the condition `AND (colname LIKE '`*`patten`*`%')` to force the DB server to optimize the query of the column is indexed. The `CONSTRUCT` instruction uses this technique when the entered criteria does not end with a * star wildcard.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no `[ ]` bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

Querying system catalog tables

As in Informix®, PostgreSQL provides system catalog tables (actually, system views). But the table names and their structure are quite different.

Solution

No automatic conversion of Informix® system tables is provided by the database interface.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

`UPDATE table SET ( `*`col-list`*` ) = ( `*`val-list`*` )`

Or

```
UPDATE table SET table.* = codeph.*
```

```
UPDATE table SET * = myrecord.*
```

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

The LENGTH() function

Informix® provides the LENGTH() function:

```
SELECT LENGTH("aaa"), LENGTH(col1) FROM table
```

PostgreSQL has a equivalent function with the same name, but there is some difference:

Informix® does not count the trailing blanks for CHAR or VARCHAR expressions, while PostgreSQL counts the trailing blanks.

With the PostgreSQL LENGTH function, when using a CHAR column values are always blank padded, so the function returns the size of the CHAR column. When using a VARCHAR column, trailing blanks are significant, and the function returns the number of characters, including trailing blanks.

PostgreSQL raises an error if the LENGTH() parameter is NULL. Informix® returns zero instead.

Solution
The PostgreSQL database interface cannot simulate the behavior of the Informix® LENGTH() SQL function.

Review the program logic and make sure you do not pass NULL values to the LENGTH() SQL function.

You must check if the trailing blanks are significant when using the LENGTH() function.

If you want to count the number of character by ignoring the trailing blanks, you must use the RTRIM() function.

BDL programming
PostgreSQL related programming topics.
Handling SQL errors when preparing statements

The PostgreSQL connector is implemented with the PostgreSQL **libpq** API. This library does not provide a way to send SQL statements to the database server during the BDL PREPARE instruction, like the Informix® interface does. The statement is sent to the server only when opening the cursors or when executing the statement, because the database driver needs to provide the data types of the SQL parameters (only known at OPEN / EXECUTE time).

Therefore, when preparing an SQL statement with the BDL PREPARE instruction, no SQL errors can be returned if the statement has syntax errors or if a column or a table name does not exist in the database. However, an SQL error will occur after the OPEN or EXECUTE instructions.

Solution

Check that your BDL programs do not test STATUS or SQLCA.SQLCODE variable just after PREPARE instructions.

Change the program logic in order to handle the SQL errors when opening the cursors (OPEN) or when executing SQL statements (EXECUTE).

Informix® specific SQL statements in BDL

The BDL compiler supports several Informix® specific SQL statements that have no meaning when using PostgreSQL.

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution

Review your BDL source and remove all static SQL statements which are Informix® specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

PostgreSQL does not support insert cursors.

Solution

Insert cursors are emulated by the PostgreSQL database interface.

Cursors WITH HOLD

Informix® closes opened cursors automatically when a transaction ends unless the WITHHOLD option is used in the DECLARE instruction. In PostgreSQL, opened cursors using SELECT statements without a FOR UPDATE clause are not closed when a transaction ends. Actually, all PostgreSQL cursors are 'WITH HOLD' cursors unless the FOR UPDATE clause issued in the SELECT statement.

Cursors declared FOR UPDATE and using the WITH HOLD option cannot be supported with PostgreSQL because FOR UPDATE cursors are automatically closed by PostgreSQL when the transaction ends.

Solution

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed.

Since PostgreSQL automatically closes FOR UPDATE cursors when the transaction ends, opening cursors declared FOR UPDATE and the WITH HOLD option results in an SQL error that does not normally appear with Informix® under the same conditions. Review the program logic in order to find another way to set locks.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

In both Informix® and PostgreSQL, locks are released when closing the cursor or when the transaction ends.

PostgreSQL locking granularity is at the row level.

To control the behavior of the program when locking rows, Informix® provides a specific instruction to set the wait mode:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds }
```

The default mode is NOT WAIT. This as an Informix-specific SQL statement; PostgreSQL has no equivalent for "SET LOCK MODE TO NOT WAIT".

Solution

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

You must review the program logic if you use pessimistic locking because it is based on the NOT WAIT mode which is not supported by PostgreSQL.

UPDATE/DELETE WHERE CURRENT OF

Informix® allows positioned UPDATEs and DELETEs with the "`WHERE CURRENT OF` *`cursor`*" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

UPDATE/DELETE ... WHERE CURRENT OF is supported by PostgreSQL with server-side cursors created with a DECLARE statement.

Solution

UPDATE/DELETE ... WHERE CURRENT OF instructions are executed as is. Since SELECT FOR UPDATE statements are now executed with a server cursor by using a DECLARE PostgreSQL statement, native positioned update/delete takes place.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

PostgreSQL does not provide LOAD and UNLOAD instructions.

Solution

LOAD and UNLOAD instructions are supported.

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

PostgreSQL supports SQL Interruption in a similar way. The db client must issue an PQcancel() libPQ call to interrupt a query.

Solution

The PostgreSQL database driver supports SQL interruption and converts the SQLSTATE code 57014 to the Informix® error code -213.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

PostgreSQL supports native scrollable cursors.

Solution

The PostgreSQL database driver uses native scrollable cursors by declaring server cursors with the SCROLL clause.

SQL adaptation guide for SQLite 3.x

Installation (Runtime Configuration)
SQLite related installation topics.
Install SQLite and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. If the `dbmsqt` database driver is linked dynamically with the SQLite library, you must install the SQLite software on your computer. However, on most platforms, the driver has an embedded version of the SQLite library, and on platforms such as Linux™ as Mac OS X™, the SQLite library is usually present.
   The minimum required version is SQLite 3.6.
2. Create a new SQLite database.
   To create a new database with tables, start the sqlite3 command line tool and execute SQL statements:

```
$ sqlite3 /var/data/stores.db
sqlite> CREATE TABLE customer ( cust_id INT PRIMARY KEY, ... );
$ .exit
```

   To create an empty database, you can also issue the following command:

```
$ sqlite3 /var/data/stores.db ""
```

   or create an empty file with operating system command:

```
$ touch /var/data/stores.db
```

   And empty file can also be created from a program by using a `base.Channel` object:

```
DEFINE ch base.Channel
LET ch = base.Channel.create()
CALL ch.openFile("/var/data/stores.db","w")
CALL ch.close
```

Prepare the runtime environment - connecting to the database

1. In order to connect to SQLite, you must have a database driver "dbmsqt" in `FGLDIR/dbdrivers`. On most platforms, the SQLite driver is linked statically with the SQLite library, in other word SQLite is embedded in the ODI driver. However, on some platforms such as Linux™ and Max OS X, where the SQLite library is usually present.
2. Make sure that the SQLite environment variables are properly set. You may want to define an environment variable such as SQLITEDIR the hold the installation directory of SQLite, which can then be used to set PATH and LD_LIBRARY_PATH. See SQLite documentation for more details.
3. If the SQLite library is not embedded in the the dbmsqt* driver, the environment must be set to find the SQLite library. Verify the environment variable defining the search path for the SQLite shared library.

   Table 211: Shared library environment setting for SQLite

SQLite version	Shared library environment setting (if SQLite lib not built-in driver)
SQLite 3.6 and higher	*UNIX*™: Add $SQLITEDIR/lib to LD_LIBRARY_PATH (or its equivalent). *Windows*™: Add %SQLITEDIR%\bin to PATH.

4. Make sure that all operating system users running the application have read/write access to the database file.

5. SQLite uses UTF-8 encoding. If the locale used by the runtime system (LANG/LC_ALL) is not compatible to UTF-8 (for example, fr_FR.iso88591), Genero will do the appropriate character set conversions.
6. Set up the fglprofile entries for database connections.
   a) Define the SQLite database driver:

   ```
   dbi.database.dbname.driver = "dbmsqt"
   ```

   b) The "source" parameter defines the path to the SQLite database file. Note that the database file must reside on the local disk (SQLite does not support network file systems). SQLite also supports in-memory database creation with the :memory: db specification. See SQLite documentation (sqlite3_open) for more details.

   ```
   dbi.database.dbname.source = "/opt/myapp/stock.dbs"
   ```

   c) If the "source" parameter defines a relative path or a simple file name and the SQLite database file does not reside in that location according to the current directory of the fglrun process, define the DBPATH environment variable to find the database file. See DBPATH documentation for more details about this environment variable.

   ```
   DBPATH="/opt/myapp"
   ```

Database concepts
SQLite related database concepts topics.
Database concepts

Informix® servers can handle multiple database entities, while SQLite can manage several database files.

Solution

Map each Informix® database to a SQLite database file.

Consider creating the SQLite database file before using the connection instruction. The database file can be created as an empty file, with a OS shell command (touch) or by program by using the file utility classes.

It is possible to specify an SQLite database file name in the database specification in CONNECT TO or DATABASE instructions:

```
DATABASE "/opt/myapp/database/stock1.dbs"
```

However, it is recommended to use an indirection by using an abstract name identifier in the program, and by defining the real database file with the "source" connection parameter. The file defined by "source" is then found directly (can be a relative or absolute path), or according to DBPATH settings if not found from the current directory of fglrun (when it's not an absolute path).

In the program:

```
DATABASE stock
```

In the FGLPROFILE configuration file, define the SQLite driver and the database file:

```
dbi.database.stock.driver = "dbmsqt"
dbi.database.stock.source = "/opt/myapp/database/stock1.dbf"
```

FGLPROFILE could also define the file name only:

```
dbi.database.stock.source = "stock1.dbf"
```

And the file would be found by using DBPATH:

```
DBPATH="/opt/myapp/database"
```

When specifying `:memory:` as database file name, an empty SQLite database is created in memory. This can be useful if the persistence of the data is not required after the program has terminated:

```
DATABASE ":memory:"
```

Concurrency management

Informix® is a multiuser database engine, while SQLite is typically used for a single-user application. SQLite 3 supports multiuser access to the same database file, but it is not designed for large multiuser applications.

SQLite 3 supports two isolation levels: SERIALIZABLE (the default), and READ UNCOMMITTED. The isolation level can be changed with the PRAGMA command.

By default in the SERIALIZABLE isolation level, SQLite will raise an SQL error if a program tries to access a database resource in use by another program. To avoid the SQL error and force programs to wait for each other, programs define the behavior when the SQLite database is busy (SQLITE_BUSY), with a specific API call. No SQL command exists for this.

Solution

We recommend that you use SQLite for single-user DB applications. If several programs must access the same SQLite database, each program must perform a SET LOCK MODE TO WAIT instruction after the connection: SET LOCK MODE will be mapped to a call to the sqlite3_busy_timeout() SQLite API function to get the same behavior as Informix®, while SET ISOLATION instructions will be ignored.

Transactions handling

Informix® and SQLite have similar commands to begin, commit or rollback transaction. There are however some important differences you must be aware of.

With SQLite, DDL statements can be executed (and canceled) in transaction blocks, as with Informix®.

Informix® version 11.50 introduces savepoints with the following instructions:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] ]
RELEASE SAVEPOINT name
```

SQLite supports savepoints too. However, there are differences:

1. SAVEPOINT can be used instead of BEGIN TRANSACTION. In this case, RELEASE is like a COMMIT.
2. The syntax of a rollback to the savepoint is ROLLBACK [TRANSACTION] TO [SAVEPOINT] *name* .
3. The syntax of a release of the savepoint is RELEASE [SAVEPOINT] *name* .
4. Rollback must always specify the savepoint name.
5. You cannot rollback to a savepoint if cursors are opened.
6. In SQLite versions prior to 3.7, you cannot rollback are transaction if a cursor is open.

Solution

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with SQLite. The BEGIN WORK, COMMIT WORK and ROLLBACK WORK commands are translated the native commands of SQLite.

> **Note:** If you want to use savepoints, always specify the savepoint name in ROLLBACK TO SAVEPOINT and do not open cursors during transactions using savepoints. If you are using an

SQLite versions prior to 3.7, it is not possible to perform a ROLLBACK WORK if a cursor (with hold) is currently open.

See also SELECT FOR UPDATE

Database users

Informix® supports database users that must be explicitly declared to the database by granting privileges.

SQLite does not have the database users concept. However, the operating system user must have read/write access to the database file.

Solution

SQLite is mainly designed for single-user applications.

Data dictionary
SQLite related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way. As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

SQLite does not implement a native BOOLEAN type, but accepts BOOLEAN in the SQL syntax.

Solution

The SQLite database interface supports the BOOLEAN data type, and converts the BDL BOOLEAN integer values to a CHAR(1) of '1' or '0'.

CHARACTER data types

Informix® supports the following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- NVARCHAR(N[,M]) with N<=255 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/ NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/ NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N) is expressed in bytes, not characters as in some other databases)

SQLite 3 provides the TEXT native data type with no strict size limitation. SQLite allows the CHAR(n), VARCHAR(n), NCHAR(n) and NVARCHAR(n) type names to be used, but actually stores the data in a TEXT native type.

SQLite treats empty strings as NOT NULL values like Informix®.

> **Note:** With the default BINARY collation, SQLite compares VARCHAR and CHAR values by taking trailing blanks into account. Informix® always ignores trailing blanks when comparing CHAR/VARCHAR values.

SQLite supports only the UTF-8 character encoding. Thus, client applications must provide UTF-8 encoded strings.

Solution

The database interface supports character string variables in SQL statements for input (BDLUSING) and output (BDL INTO).

> **Important:** With the default BINARY collation, CHAR and VARCHAR comparison in SQLite takes trailing blanks into account. As result, some queries returning rows with Informix® may not return the same result set with SQLite. When creating a table in SQLite, you can change the default collation rule to force the database engine to trim trailing blanks before comparing CHAR/VARCHAR values, by specifying COLLATION RTRIM in the column definitions. When creating a table from a Genero program, if Informix® emulation is enabled for the CHAR/VARCHAR types, the SQLite database driver adds automatically COLLATE RTRIM after the CHAR(N) or VARCHAR(N) type, to get the same comparison semantics as Informix®.

Regarding character sets, the SQLite database driver automatically converts character strings used in the programs to/from UTF-8 for SQLite.

SQLite uses character length semantics: When you define a CHAR(20) and the database character set is multibyte, the column can hold more bytes/characters than the Informix® CHAR(20) type, when using byte length semantics.

When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a SQLite database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in SQLite database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

See also the section about Localization.

NUMERIC data types

Informix® supports several data types to store numbers:

Table 212: Informix® numeric data types

Informix® data type	Description
SMALLINT	16 bit signed integer
INT / INTEGER	32 bit signed integer
BIGINT	64 bit signed integer
INT8	64 bit signed integer (replaced by BIGINT)
DEC / DECIMAL	Equivalent to DECIMAL(16)
DEC / DECIMAL(p)	Floating-point decimal number
DEC / DECIMAL(p,s)	Fixed-point decimal number
MONEY	Equivalent to DECIMAL(16,2)
MONEY(p)	Equivalent to DECIMAL(p,2)
MONEY(p,s)	Equivalent to DECIMAL(p,s)
REAL / SMALLFLOAT	32-bit floating point decimal (C float)
DOUBLE PRECISION / FLOAT[(n)]	64-bit floating point decimal (C double)

SQLite 3 supports INTEGER (8 byte integer) and REAL (8 byte floating point) as native types to store numbers, but allows synonyms:

Table 213: SQLite numeric data types and supported synonyms

Supported synonyms	SQLite type affinity
INT, INTEGER, TINYINT, SMALLINT, MEDIUMINT, BIGINT, UNSIGNED BIG INT, INT2, INT8	INTEGER (8 bytes!)
REAL, DOUBLE, DOUBLE PRECISION, FLOAT	REAL (8 bytes!)
DECIMAL(p,s), NUMERIC	NUMERIC (based on REAL)

Important: Exact decimal types like DECIMAL(p,s) may be stored as floating point numbers (REAL), INTEGERs or TEXT, according to the type affinity selected by SQLite. When converted to floating point type, data loss and rounding rule differences are possible with SQLite.

Solution

Informix® numeric types are not translated by the database driver.

Since SQLite 3 does not have exact decimal types like DECIMAL(p,s), you must pay attention to the rounding rules and data loss when using numbers with many significant digits. Arithmetic operations like division have different results than with Informix®. It is better to fetch the original column value into a DECIMAL variable, and do arithmetic operations in the application program.

DATE and DATETIME data types

Informix® provides two data types to store date and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

SQLite 3 does not have a native type for date/time storage, but you can use data/time type names and functions based on the string representation of dates and times. The date/time values are stored in the TEXT native type. The date/time functions of SQLite are based on standard DATE (YYYY-MM-DD), TIME (hh:mm:ss) and TIMESTAMP (YYYY-MM-DD hh:mm:ss) concepts.

For maximum flexibility with other RDBMS SQL languages, SQLite allows to define table columns with your own type names. You can for example use the SMALLDATETIME, SMALLTIME, TIME(N), DATETIME(N) type names.

Solution

The following conversions are done by the ODI SQLite driver for date/time types:

- DATE type is not translated, it will be used as is by SQLite.
- DATETIME HOUR TO MINUTE is translated to SMALLTIME.
- DATETIME HOUR TO SECOND is translated to TIME.
- DATETIME HOUR TO FRACTION(n) is transalated to TIME(n).
- DATETIME YEAR TO DAY is translated to TINYDATETIME.
- DATETIME YEAR TO MINUTE is translated to SMALLDATETIME.
- DATETIME YEAR TO SECOND is translated to DATETIME.
- DATETIME YEAR TO FRACTION(n) is translated to DATETIME(n).
- DATETIME with another precision as above are translated to TIMESTAMP.

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

Informix's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

SQLite 3 does not provide a data type similar to Informix® INTERVAL.

Solution

It is not recommended that you use the INTERVAL data type because SQLite 3 has no equivalent native data type. This would cause problems when doing INTERVAL arithmetic on the database server side. However, INTERVAL values can be stored in CHAR(50) columns.

SERIAL data types

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERT statements that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

SQLite supports the **AUTOINCREMENT** attribute for columns:

- Only one column must be declared as INTEGER PRIMARY KEY AUTOINCREMENT.
- To get the last generated number, SQLite provides the sqlite_sequence table:

  ```
  SELECT seq FROM sqlite_sequence WHERE name='table_name';
  ```
- When inserting a zero in the auto-increment column, SQLite will not generate a new sequence like Informix® does.
- When inserting a NULL in the auto-increment column, SQLite generates a new sequence; Informix® denies NULLs in SERIALs.

Solution

When using SQLite, the SERIAL data type is converted to INTEGER PRIMARY KEY AUTOINCREMENT.

The SQLCA.SQLERRD[2] register is filled automatically after each INSERT with the last generated number, by fetching the value from the `sqlite_sequence` table.

SQLite (V 3.6) does not support auto-incremented BIGINTs. Therefore, BIGSERIAL or SERIAL8 cannot be converted.

Because SQLite does not behave like Informix® regarding zero and NULL value specification for auto-incremented columns, all INSERT statements must be reviewed to remove the SERIAL column from the list.

For example, the following statement:

```
INSERT INTO tab (col1,col2) VALUES ( 0 , p_value)
```

Can be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.*) -- will use the serial column
```

Can be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

ROWIDs

When creating a table, Informix® automatically adds a ROWID integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

SQLite supports ROWID columns as 64-bit integers. Informix® ROWIDs are 16-bit integers.

With Informix®, SQLCA.SQLERRD[6] contains the ROWID of the last INSERTed or UPDATEd row. This is not supported with SQLite because SQLite ROWIDs are not INTEGERs.

Solution

If the BDL application uses Informix® ROWIDs as primary keys, the program logic should be reviewed in order to use the real primary keys.

If you cannot avoid the use of rowids, you must change the type of the variables which hold ROWID values. Instead of using INTEGER, you must use DECIMAL(20).

> **Note:** All references to SQLCA.SQLERRD[6] must be removed because this variable will not contain the ROWID of the last INSERTed or UPDATEd row when using the SQLite interface.

Foreign key support

Foreign keys are an important feature in modern database design, to enforce database integrity:

```
CREATE TABLE orders (
   ... ,
   FOREIGN KEY(ord_customer) REFERENCES customer(cust_num) )
  )
```

SQLite (3.6.19 and +) implements foreign key support, but this feature is not enabled by default. In fact, it is possible to define foreign keys on tables, but when doing database operations, the constraints are not enforced until you enable it explicitely with a PRAGMA command.

Solution

In order to turn on foreign key constraint checking, you must issue a PRAGMA command, which can for example be executed with a EXECUTE IMMEDIATE instruction:

```
EXECUTE IMMEDIATE "PRAGMA foreign_keys = ON"
```

Future releases of SQLite might change this, so that foreign key constraints enabled by default.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

SQLite 3 provides TEXT and BLOB native data types for large objects storage.

Solution

The SQLite database interface can convert BDL TEXT data to TEXT and BYTE data to BLOB.

Data type conversion table: Informix to SQLite

Table 214: Data type conversion table between Informix® and SQLite

Informix® data types	SQLite data types
CHAR(n)	CHAR(n) COLLATE RTRIM
VARCHAR(n[,m])	VARCHAR(n) COLLATE RTRIM
LVARCHAR(n)	VARCHAR(n) COLLATE RTRIM
NCHAR(n)	NCHAR(n)
NVARCHAR(n)	NVARCHAR(n)
BOOLEAN	BOOLEAN
SMALLINT	SMALLINT
INT / INTEGER	INTEGER
BIGINT	BIGINT
INT8	BIGINT
SERIAL[(start)]	INTEGER (see note 1)
BIGSERIAL[(start)]	N/A (see note 1)
INT8[(start)]	N/A (see note 1)
DOUBLE PRECISION / FLOAT[(n)]	FLOAT
REAL / SMALLFLOAT	SMALLFLOAT
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL(p)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL	DECIMAL
MONEY(p,s)	DECIMAL(p,s)
MONEY(p)	DECIMAL(p,2)
MONEY	DECIMAL(16,2)
TEXT	TEXT
BYTE	BLOB
DATE	DATE
DATETIME HOUR TO MINUTE	SMALLTIME
DATETIME HOUR TO SECOND	TIME
DATETIME HOUR TO FRACTION(n)	TIME(n)
DATETIME YEAR TO DAY	TINYDATETIME
DATETIME YEAR TO MINUTE	SMALLDATETIME
DATETIME YEAR TO SECOND	DATETIME

Informix® data types	SQLite data types
DATETIME YEAR TO FRACTION(n)	DATETIME(n)
DATETIME q1 TO q2 (different from above)	TIMESTAMP
INTERVAL q1 TO q2	CHAR(50)

Notes:

1. For more details about serial emulation, see SERIAL data types on page 715.

Data manipulation
SQLite related data manipulation topics.
Outer joins

The original syntax of OUTER joins of Informix® is different from the SQLite outer join syntax:

In Informix® SQL, outer tables are defined in the FROM clause with the **OUTER** keyword:

```
SELECT ... FROM a, OUTER(b)
  WHERE a.key = b.akey

SELECT ... FROM a, OUTER(b,OUTER(c))
  WHERE a.key = b.akey
  AND b.key1 = c.bkey1
  AND b.key2 = c.bkey2
```

SQLite 3 supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
 ON cust.key = order.custno

SELECT ...
FROM cust LEFT OUTER JOIN order
    LEFT OUTER JOIN item
    ON order.key = item.ordno
   ON cust.key = order.custno
WHERE order.accepted = 1
```

See the SQLite 3 SQL reference for a complete description of the syntax.

Solution

For better SQL portability, you should use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The SQLite 3 interface can convert most Informix® OUTER specifications to SQLite 3 outer joins.

Prerequisites:

1. In the FROM clause, the main table must be the first item and the outer tables must be listed from left to right in the order of outer levels.

   Example which does not work: "FROM OUTER(tab2), tab1".
2. The outer join in the WHERE clause must use the table name as prefix.

   Example: "WHERE tab1.col1 = tab2.col2".

Restrictions:

1. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   Example: "... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10".

2. Statements composed by 2 or more SELECT instructions using OUTERs are not supported.

   Example: "SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"

Remarks:

1. Table aliases are detected in OUTER expressions.

   OUTER example with table alias: "OUTER(tab1 alias1)".
2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.

   OUTER join example with table on the left: "WHERE outertab.col1 = maintab.col2 ".
3. Table names detection is not case-sensitive.

   Example: "SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2".
4. Temporary tables are supported in OUTER specifications.

Transactions handling

Informix® and SQLite have similar commands to begin, commit or rollback transaction. There are however some important differences you must be aware of.

With SQLite, DDL statements can be executed (and canceled) in transaction blocks, as with Informix®.

Informix® version 11.50 introduces savepoints with the following instructions:

```
SAVEPOINT name [UNIQUE]
ROLLBACK [WORK] TO SAVEPOINT [name] ]
RELEASE SAVEPOINT name
```

SQLite supports savepoints too. However, there are differences:

1. SAVEPOINT can be used instead of BEGIN TRANSACTION. In this case, RELEASE is like a COMMIT.
2. The syntax of a rollback to the savepoint is ROLLBACK [TRANSACTION] TO [SAVEPOINT] *name* .
3. The syntax of a release of the savepoint is RELEASE [SAVEPOINT] *name* .
4. Rollback must always specify the savepoint name.
5. You cannot rollback to a savepoint if cursors are opened.
6. In SQLite versions prior to 3.7, you cannot rollback are transaction if a cursor is open.

Solution

Regarding transaction control instructions, BDL applications do not have to be modified in order to work with SQLite. The BEGIN WORK, COMMIT WORK and ROLLBACK WORK commands are translated the native commands of SQLite.

> **Note:** If you want to use savepoints, always specify the savepoint name in ROLLBACK TO SAVEPOINT and do not open cursors during transactions using savepoints. If you are using an SQLite versions prior to 3.7, it is not possible to perform a ROLLBACK WORK if a cursor (with hold) is currently open.

See also SELECT FOR UPDATE

Temporary tables

Informix® temporary tables are created through the CREATE TEMP TABLE DDL instruction or through a SELECT ... INTO TEMP statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

> **Note:** BDL reports create a temporary table when the rows are not sorted externally (by the source SQL statement).

Informix® allows you to create indexes on temporary tables. No name conflict occurs when several users create an index on a temporary table by using the same index identifier.

SQLite supports temporary tables with the CREATE TEMP TABLE statement.

Solution

Informix® CREATE TEMP TABLE statements are kept as is and SELECT INTO TEMP statements are converted to SQLite native SQL CREATE TEMP TABLE AS SELECT ...

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements, while SQLite supports the LIKE statement only.

MATCHES requires * and ? wildcard characters, and LIKE uses the % and _ wildcards was equivalents.

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x' )
```

MATCHES allows you to use brackets to specify a set of matching characters at a given position:

```
( col MATCHES '[Pp]aris' )
( col MATCHES '[0-9][a-z]*' )
```

The SQLite LIKE operator has no operator for [ ] brackets character ranges.

Solution

The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( col-list ) = ( val-list )
```

Or

```
UPDATE table SET table.* = codeph.*
```

```
UPDATE table SET * = myrecord.*
```

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

BDL programming
SQLite related programming topics.
Informix-specific SQL statements in BDL

The BDL compiler supports several Informix® specific SQL statements that have no meaning when using SQLite:

- CREATE DATABASE
- DROP DATABASE
- START DATABASE (SE only)

- ROLLFORWARD DATABASE
- SET [BUFFERED] LOG
- CREATE TABLE with special options (storage, lock mode, etc.)

Solution

Review your BDL source and remove all static SQL statements which are Informix® specific.

INSERT cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

SQLite does not support insert cursors.

Solution

Insert cursors are emulated by the SQLite database interface.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to prevent several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

- The row must be fetched in order to set the lock.
- If the cursor is local to a transaction, the lock is released when the transaction ends. If the cursor is declared "WITH HOLD", the lock is released when the cursor is closed.

SQLite does not support the FOR UPDATE close in SELECT syntax.

Solution

Review the program logic and remove SELECT ... FOR UPDATE statements, as SQLite doesn't support them.

UPDATE/DELETE WHERE CURRENT OF

Informix® allows positioned UPDATEs and DELETEs with the "`WHERE CURRENT OF cursor`" clause, if the cursor has been DECLARED with a SELECT ... FOR UPDATE statement.

SELECT ... FOR UPDATE is not supported by SQLite.

Solution

Review the program logic and use primary keys to update the rows.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into a text file and the LOAD instructions insert rows from a text file into a database table.

SQLite 3.0 does not natively provide LOAD / UNLOAD instructions.

Solution

LOAD and UNLOAD instructions are supported.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

SQLite 3.x does not support native scrollable cursors.

Solution

The SQLite database driver emulates scrollable cursors with temporary files.

See Scrollable cursors on page 422 for more details about scroll cursor emulation.

Modifying many rows in a table

SQLite is very slow when doing commits, because of the technique used to ensure data integrity (see SQLite documentation for details).

When a program executes a DML statement like INSERT, it will be automatically committed by SQLite. As result, there will be as many transactions/commits as data manipulation statements.

It takes for example about 10 seconds to insert 1000 rows on an Intel core i7 2.60GHz CPU / 5400.0 RPM HDD computer.

Solution

If a program must modify many rows in a table, execute the SQL statement within a transaction block delimited by BEGIN WORK / COMMIT WORK instructions. This will dramatically speed up the process.

See Performance with transactions on page 453.

Optimizing database file usage

By default, when deleting a large amount of data in an SQLite database it leaves behind empty space, causing the database file to be larger than stricly necessary.

This might be an issue with some mobile applications, when the disk space of the mobile device is limited.

Solution

Execute the VACUUM SQL command, to truncate the database file and reduce the disk usage.

According to the application, the VACUUM command can be executed:

- when starting the application,
- after doing a large db operation (like a synchronization with a central db),
- as a manual option that the user can trigger.

Note that SQLite also supports "PRAGMA auto_vacuum", but it appears that it's not as efficient as the VACUUM command, regarding page fragmentation.

Pay attention to the fact that VACUUM needs twice the disk space of the actual database file, because it rebuilds totally the db file.

VACUUM is not Informix SQL syntax, thus you can't write it directly in the BDL code: You must use EXECUTE IMMEDIATE:

```
EXECUTE IMMEDIATE "VACUUM"
```

SQL adaptation guide for SAP Sybase ASE 16.x

Installation (Runtime Configuration)
Sybase ASE related installation topics.
Install Sybase ASE and create a database - database configuration/design tasks

If you are tasked with installing and configuring the database, here is a list of steps to be taken:

1. Install Sybase ASE software on your computer, with the Sybase client software.

   Make sure that the server is started and environment variables are properly set (On UNIX™, you will find SYBASE.* shell scripts to source in the installation directory).

2. Try to connect to the server with the isql command line tool.

   If needed, change the password of the "sa" database administrator:

```
$ isql -S server_name -U sa
1> sp_password old_password, new_password
2> go
Password correctly set.
(return status = 0)
```

   Starting with Sybase ASE 15.7, the password of the sa user is defined at installation time.

3. Define server's default character set: You must identify what server character set you want to use (typically, utf8) and re-configure the server if needed.

   With Sybase ASE, the db character set cannot be specified at the database level, it is defined at the server level, typically during the installation. It is also possible to change the srever character set with the `charset` utility and with the `sp_configure` stored procedure. You have to shutdown the server, start a first time to have the server take the new character set into account and then restart a second time for use. See Sybase documentation for more details or more recent versions of Sybase ASE.

   Make sure that you select a case-sensitive character set / sort order.

   > **Note:** Check the `$SYBROOT/locales/locales.dat` file, to make sure that your current locale (LANG/LC_ALL) is listed in the file. You may want to add the following lines for UTF-8 support, under the section of your operating system:
   >
   > ```
   >   locale = POSIX, us_english, utf8
   >   locale = en_US.utf8, us_english, utf8
   > ; Windows only:
   >   locale = .fglutf8, us_english, utf8
   > ```

   Example:

```
$ export DSQUERY=servername
$ charset -Usa -P binary.srt utf8
Please enter sa's Password:
Loading file 'binary.srt'.

Found a [sortorder] section.

This is Class-1 sort order.

Finished loading the Character Set Definition.

Finished loading file 'binary.srt'.

1 sort order loaded successfully

$ isql -Usa -P
1> sp_configure 'default sortorder id', 50, 'utf8'
2> go
```

```
3> shutdown
4> go
Server SHUTDOWN by request.
ASE is terminating this process.
.......

$ $SYBROOT/ASE_*/install/RUN_servername
... (server makes some initialization / setup and stops) ...

$ $SYBROOT/ASE_*/install/RUN_servername
...
```

4. Create a new Sybase database entity, with sufficient storage devices for data and transaction log.

   Use either the Sybase Central, the Sybase Control Center GUI tool or use isql with SQL commands.

   Connect to the server with the sa user.

   First create database devices for data and transaction log. Define a transaction that can hold the biggest transaction your application can do to avoid administrative tasks to dump the log when the server hangs. When creating the database, use the new created database devices as database segments:

   ```
   use master
   go
   disk init
        name = "devname",
        physname = "filename",
        size = devsize ...
   go
   create database dbname
      on devname
      with ...
   go
   ```

5. Leave the default transaction mode ("unchained" mode), to force explicit transaction start and end commands.

   See the `set chained` command for more details.

6. The database allows NULLs by default when creating columns.

   This is controlled by the `'allow nulls by default'` option. If this option is set to OFF, columns created without NULL or NOT NULL keywords are NOT NULL by default:

   ```
   master..sp_dboption dbname, 'allow nulls by default', true
   go
   ```

7. The database must allow Data Definition Language (DDL) statements in transaction blocks.

   To turn this on, use following commands:

   ```
   master..sp_dboption dbname, 'ddl in tran', true
   go
   checkpoint
   go
   ```

8. For development purpose, consider to set the database option to truncate the transaction log when a checkpoint occurs, otherwise you will have to dump the transaction log when it is full.

   Command to automatically truncate the transaction log on checkpoint:

   ```
   master..sp_dboption dbname, 'trunc log on chkpt', true
   go
   ```

9. Create a new login dedicated to your application: the application administrator.

Assign the new created database as default database for this user:

```
use dbname
go
sp_addlogin 'username', 'password', dbname, ... options ...
go
```

10. Create a new database user linked to the new application administrator login:

    In Sybase Central, open to the "Databases" node, select "Users" and right-click "New" ...

```
use dbname
go
sp_adduser 'username', 'group', ... options ...
go
```

    See documentation for more details about database users and privileges. You must create groups to make tables visible to all users.

11. If you plan to use SERIAL emulation based on triggers using a registration table, create the SERIALREG table.

    Create the triggers for each table using a SERIAL. See issue SERIAL data types for more details.

12. Create the application tables.

    Do not forget to convert Informix® data types to Sybase ASE data types. See topic data type Conversion Table for more details. In order to make application tables visible to all users, make sure that all users are members of the group of the owner of the application tables. For more details, see ASE documentation ("Database object names and prefixes").

Prepare the runtime environment - connecting to the database

1. In order to connect to Sybase ASE, you must have a Sybase ASE database driver "dbmase" in FGLDIR/dbdrivers.
2. If you want to connect to a remote database server, you must have the Sybase ASE Client Software installed on the computer running BDL applications.

   The Sybase Open Client Library is required.
3. Make sure that the Sybase ASE client environment variables are properly set.

   Check for example SYBASE (the path to the installation directory), SYBASE_ASE (the name of the server sub-directory), SYBASE_OCS (the name of the client sub-directory), etc. See Sybase ASE documentation for more details.
4. Verify the environment variable defining the search path for Sybase OCS database client shared libraries (libsybct[64].so, libsybcs[64].so UNIX™, LIBSYBCT[64].DLL and LIBSYBCS[64].DLL on Windows™).

   Table 215: Shared library environment setting for Sybase ASE

Sybase ASE version	Shared library environment setting
Sybase ASE 16.0 and higher	*UNIX*: Add $SYBASE_OCS/lib to LD_LIBRARY_PATH (or its equivalent). *Windows*: Add %SYBASE_OCS%\dll to PATH. Where SYBASE_OCS is the directory of the Sybase Open Client Software.

5. The name of the Sybase server must be registered in a configuration file.

   On UNIX™, the server name must be defined in the "`interfaces`" file located in $SYBASE. On Windows™, the server name must be defined in the "`sql.ini`" file located in `%SYBASE%\ini`. You

may want to define the DSQUERY environment variable to the name of the server. See Sybase documentation for more details.

When connecting from a Genero program, both database and server names can be specified with:

database@server

For more details see the description for the connection data source parameter in DATABASE and CONNECT instructions.

6. Check the database client locale settings of Sybase.

   The Sybase client locale must match the locale used by the runtime system (LC_ALL, LANG on UNIX™, ANSI code page on Windows™).

   By default, Sybase OCS uses the character set defined by the operating system. On Windows™, this is the ANSI code page, on UNIX™ it is defined by LC_CTYPE, LC_ALL or LANG environment variables. Note that Genero BDL allows to define the LANG environment variable also on Windows™. The value of the LANG environment variable must be listed in the "`locales.dat`" file under the `$SYBASE/locales` directory, otherwise you will get an error when connecting to the database.

   > **Note:** Check the `$SYBROOT/locales/locales.dat` file, to make sure that your current locale (LANG/LC_ALL) is listed in the file. You may want to add the following lines for UTF-8 support, under the section of your operating system:

   ```
     locale = POSIX, us_english, utf8
     locale = en_US.utf8, us_english, utf8
   ; Windows only:
     locale = .fglutf8, us_english, utf8
   ```

   See also Sybase OCS documentation regarding localization and character set definition.

7. Test the Sybase ASE Client Software: Make sure the server is started and try to connect to a database by using the Sybase ASE command interpreter:

   ```
   $ isql -S server -U appadmin -P password
   ```

8. Set up the fglprofile entries for database connections:

   a) Define the Sybase ASE database driver:

      ```
      dbi.database.dbname.driver = "dbmase"
      ```

   b) Define the connection timeout with the following fglprofile entry:

      ```
      dbi.database.dbname.ase.logintime = integer
      ```

      This entry defines the number of seconds to wait for a connection.

      Default is 5 seconds.

   c) Define the number of rows to be pre-fetched for result sets:

      ```
      dbi.database.dbname.ase.prefetch.rows = integer
      ```

      Default is 10 rows.

Database concepts
Sybase ASE related database concepts topics.

Database concepts

As in Informix®, a Sybase ASE engine can manage multiple database entities. When creating a database object such as a table, Sybase ASE allows you to use the same object name in different databases.

Data consistency and concurrency

Data consistency involves readers which want to access data currently modified by writers and *concurrency data access* involves several writers accessing the same data for modification. *Locking granularity* defines the amount of data concerned when a lock is set (row, page, table, ...).

Informix®:

Informix® uses a locking mechanism to manage data consistency and concurrency. When a process modifies data with UPDATE, INSERT or DELETE, an exclusive lock is set on the affected rows. The lock is held until the end of the transaction. Statements performed outside a transaction are treated as a transaction containing a single operation and therefore release the locks immediately after execution. SELECT statements can set **shared locks** according to the **isolation level**. In case of locking conflicts (for example, when two processes want to acquire an exclusive lock on the same row for modification or when a writer is trying to modify data protected by a shared lock), the behavior of a process can be changed by setting the **lock wait mode**.

Control:

- Isolation level: SET ISOLATION TO ...
- Lock wait mode: SET LOCK MODE TO ...
- Locking granularity: CREATE TABLE ... LOCK MODE {PAGE|ROW}
- Explicit locking: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is read committed.
- The default lock wait mode is "not wait".
- The default locking granularity is per page.

Sybase ASE:

As in Informix®, Sybase ASE uses locks to manage data consistency and concurrency. The database manager sets **exclusive locks** on the modified rows and **shared locks** when data is read, according to the **isolation level**. The locks are held until the end of the transaction. When multiple processes want to access the same data, the latest processes must wait until the first finishes its transaction or the lock timeout occurred. The **lock granularity** is at the row or table level. For more details, see Sybase ASE's Documentation.

Control:

- The lock wait mode can be controlled with: SET LOCK {WAIT seconds | NOWAIT}
- Isolation level: Can be set with: SET TRANSACTION ISOLATION LEVEL = {0|1|2|3}
- Locking granularity: Row level.
- Explicit locking: SELECT ... FOR UPDATE

Defaults:

- The default isolation level is Read Committed (readers cannot see uncommitted data; no shared lock is set when reading data).

Solution

The SET ISOLATION TO ... Informix® syntax is replaced by SET TRANSACTION ISOLATION LEVEL ... in Sybase ASE. The next table shows the isolation level mappings done by the Sybase ASE database driver:

Table 216: Isolation level mappings done by the Sybase ASE database driver

SET ISOLATION instruction in program	Native SQL command
SET ISOLATION TO DIRTY READ	SET TRANSACTION ISOLATION LEVEL = 0
SET ISOLATION TO COMMITTED READ [READ COMMITTED] [RETAIN UPDATE LOCKS]	SET TRANSACTION ISOLATION LEVEL = 1
SET ISOLATION TO CURSOR STABILITY	SET TRANSACTION ISOLATION LEVEL = 2
SET ISOLATION TO REPEATABLE READ	SET TRANSACTION ISOLATION LEVEL = 3

For portability, it is recommended that you work with Informix® in the read committed isolation level, to make processes wait for each other (lock mode wait) and to create tables with the "lock mode row" option.

The SET LOCK MODE TO ... Informix® syntax is replaced by SET LOCK ... in Sybase ASE. If SET LOCK MODE TO WAIT is used in programs (i.e. wait forever), the driver will simulate this with a SET LOCK WAIT 5000 in Sybase ASE:

Table 217: SET LOCK MODE instruction for Sybase ASE

SET LOCK MODE instruction in program	Native SQL command
SET LOCK MODE TO NOT WAIT	SET LOCK NOWAIT
SET LOCK MODE TO WAIT n	SET LOCK WAIT n
SET LOCK MODE TO WAIT	SET LOCK WAIT 5000

See the Informix® and Sybase ASE documentation for more details about data consistency, concurrency and locking mechanisms.

Transactions handling

Informix® and Sybase ASE handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Sybase ASE:

- Sybases supports two transaction modes:
  1. The SQL standards-compatible mode, called *chained* mode, to get implicit transaction.
  2. The default mode, called *unchained* mode, where transactions have to be started/ended explicitly.
- Transactions are started with "BEGIN TRANSACTION [name]".
- Transactions are validated with "COMMIT TRANSACTION [name]".
- Transactions are canceled with "ROLLBACK TRANSACTION [name]".
- Transactions save points can be placed with "SAVEPOINT [name]".
- Sybase ASE supports named and nested transactions.
- DDL statements can be executed in transactions blocks when the 'ddl in tran' option is set to true with:

```
master..sp_dboption dbname, 'ddl in tran', true
go
```

```
checkpoint
go
```

Solution

Informix® transaction handling commands are automatically converted to Sybase ASE instructions to start, commit or rollback transactions.

Make sure that the database uses the default *unchained* mode (set chained off) and allows DDLs in transactions ('ddl in tran' option is true).

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Sybase ASE.

Database users

Until version 11.70.xC2, Informix® database users must be created at the operating system level and be members of the 'informix' group. Starting with 11.70.xC2, Informix® supports database-only users with the CREATE USER instruction, as most other db servers. Any database user must have sufficient privileges to connect and use resources of the database; user rights are defined with the GRANT command.

Before a user can access an Sybase ASE database, the system administrator (DBA) must declare the application users in the database with the GRANT statement. You may also need to define groups in order to make tables visible to other users.

Solution

See Sybase ASE documentation for more details on database logins and users.

Setting privileges

Informix® and Sybase ASE user privileges management are quite similar.

Sybase ASE provides **user groups** to grant or revoke permissions to more than one user at the same time.

Data dictionary
Sybase ASE related data dictionary topics.
BOOLEAN data type

Informix® supports the BOOLEAN data type, which can store 't' or 'f' values. Genero BDL implements the BOOLEAN data type in a different way: As in other programming languages, Genero BOOLEAN stores integer values 1 or 0 (for TRUE or FALSE). The type was designed this way to assign the result of a boolean expression to a BOOLEAN variable.

Sybase ASE provides the BIT data type to store boolean values. However, unlike Informix® types, BIT columns cannot be NULL and thus you must specify the NOT NULL constraint when creating the table.

Solution

The Sybase ASE database interface converts BOOLEAN type to BIT columns and stores 1 or 0 values in the column.

You must explicitly specify the NOT NULL constraint in the CREATE TABLE statement.

CHARACTER data types

Informix® supports following character data types:

- CHAR(N) with N<= 32767 bytes
- VARCHAR(N[,M]) with N<=255 bytes
- NCHAR(N) with N<= 32767 bytes
- LVARCHAR(N), without the 255 bytes limit (max size varies according to IDS version)

In Informix®, both CHAR/VARCHAR and NCHAR/NVARCHAR data types can be used to store single-byte or multibyte encoded character strings. The only difference between CHAR/VARCHAR and NCHAR/NVARCHAR is for sorting: N[VAR]CHAR types use the collation order, while [VAR]CHAR types use the byte order. The character set used to store strings in CHAR/VARCHAR/NCHAR/NVARCHAR columns is defined by the DB_LOCALE environment variable. The character set used by applications is defined by the CLIENT_LOCALE environment variable. Informix® uses Byte Length Semantics (the size N that you specify in [VAR]CHAR(N)is expressed in bytes, not characters as in some other databases)

Sybase ASE implements the following character data types:

- CHAR(N) with N <= 16384 bytes
- VARCHAR(N) with N <= 16384 bytes
- NCHAR(N) with N <= 16384 characters
- NVARCHAR(N) with N <= 16384 characters
- UNICHAR(N) with N <= 16384 characters
- UNIVARCHAR(N) with N <= 16384 characters

Like Informix®, Sybase ASE can store multibyte characters in CHAR / VARCHAR columns, according to the database character set. For example, Sybase can store UTF-8 strings in CHAR/VARCHAR columns. For multibyte character sets, you could also use the NCHAR / NVARCHAR or UNICHAR / UNIVARCHAR Sybase ASE types, the only difference with CHAR / VARCHAR is that the length is specified in characters instead of bytes. The UNICHAR / UNIVARCHAR store characters in 16bit UCS-2 charset only, but this is transparent to the database client.

Sybase supports automatic character set conversion between the client application and the server. By default, the Sybase database client character set is defined by the operating system locale where the database client runs. On Windows™, it is the ANSI code page of the login session (can be overwritten by setting the LANG environment variable), on UNIX™ it is defined by the LC_CTYPE, LC_ALL or LANG environment variable. You may need to edit the `$SYBASE/locales/locales.dat` file to map the OS locale name to a known Sybase character set.

Unlike most other database engines, Sybase ASE trims trailing blanks when inserting character strings in a VARCHAR column.

For example:

```
CREATE TABLE t1 ( k INT, vc VARCHAR(5))
INSERT INTO t1 VALUES ( 1, 'abc ' )
SELECT '['||vc||']' FROM t1 WHERE k = 1
----------------------------------------------
[abc]
```

With other database servers you would get 1 blank after `abc`:

```
[abc ]
```

Solution

If your application must support multibyte character sets like BIG5 or UTF-8, you should use CHAR / VARCHAR Sybase data types, where the length is specified in bytes like with Informix®.

Check that your database schema does not use CHAR, VARCHAR or LVARCHAR types with a length exceeding the Sybase ASE limit.

If your application creates tables with NCHAR/NVARCHAR types, the same type name will be used in Sybase. Keep in mind that the size of NCHAR/NVARCHAR in Sybase is specified in characters, while Informix® uses a number of bytes.

When using a multibyte character set (such as UTF-8), define database columns with the size in character units, and use character length semantics in BDL programs with FGL_LENGTH_SEMANTICS=CHAR.

When extracting a database schema from a Sybase database, the schema extractor uses the size of the column in characters, not the octet length. If you have created a CHAR(10 (characters)) column a in Sybase database using the UTF-8 character set, the .sch file will get a size of 10, that will be interpreted according to FGL_LENGTH_SEMANTICS as a number of bytes or characters.

Do not forget to properly define the database client character set, which must correspond to the runtime system character set.

Since trailing blanks are trimmed for VARCHARs, make sure that your application does not rely on this non-standard behavior.

See also the section about Localization

NUMERIC data types

Sybase ASE offers numeric data types which are quite similar to Informix® numeric data types. This table shows general conversion rules for numeric data types:

Table 218: Numeric data types (Informix® vs. Sybase ASE)

Informix®	**Sybase ASE**
SMALLINT	SMALLINT
INTEGER (synonym: INT)	INTEGER (synonym: INT)
BIGINT	BIGINT
INT8	BIGINT
DECIMAL[(p[,s)] (synonyms: DEC, NUMERIC) DECIMAL(p,s) defines a fixed point decimal where p is the total number of significant digits and s the number of digits that fall on the right of the decimal point. DECIMAL(p) defines a floating point decimal where p is the total number of significant digits. The precision p can be from 1 to 32. DECIMAL is treated as DECIMAL(16).	DECIMAL[(p[,s)] (synonyms: DEC, NUMERIC) DECIMAL[(p[,s])] defines a fixed point decimal where p is the total number of significant digits and s the number of digits that fall on the right of the decimal point. The precision p can be from 1 to 38. The default precision is 18 and the default scale is 0: • DECIMAL in Sybase ASE = DECIMAL(18,0) in Informix® • DECIMAL(p) in Sybase ASE = DECIMAL(p,0) in Informix®
MONEY[(p[,s])	Sybase ASE provides the MONEY and SMALLMONEY data types, but the currency symbol handling is quite different. Therefore, Informix® MONEY columns should be implemented as DECIMAL columns in Sybase ASE.
SMALLFLOAT (synonyms: REAL)	REAL
FLOAT[(n)] (synonyms: DOUBLE PRECISION) The precision (n) is ignored.	DOUBLE PRECISION

Sybase ASE does not support implicit character string to numeric conversions. For example, if you compare an integer column to '123' in a WHERE clause, Sybase will raise a conversion error. The problem exists also when using CHAR or VARCHAR SQL parameters.

Solution

In BDL programs

When creating tables from BDL programs, the database interface automatically converts Informix® data types to corresponding Sybase ASE data types.

There is no Sybase ASE equivalent for the Informix® DECIMAL(p) floating point decimal (i.e. without a scale). If your application is using such data types, you must review the database schema in order to use Sybase ASE compatible types. To workaround the Sybase ASE limitation, the Sybase ASE database drivers convert DECIMAL(p) types to a DECIMAL(2*p, p), to store all possible numbers an Informix® DECIMAL(p) can store. However, the original Informix® precision cannot exceed 19, since Sybase ASE maximum DECIMAL precision is 38(2*19). If the original precision is bigger as 19, a CREATE TABLE statement executed from a Genero program will fail with an Sybase ASE error 2756.

Database creation scripts

- SMALLINT and INTEGER columns do not have to use another data type in Sybase ASE.
- For DECIMALs, check the precision limit. Always use a precision and a scale.
- Convert MONEY columns to DECIMAL(p,s) columns. Always use a precision and a scale.
- Convert SMALLFLOAT columns to REAL columns.
- Since FLOAT precision is ignored in Informix®, convert this data type to FLOAT(15).

Since Sybase ASE does not support implicit character string to numeric conversions, you must check that your programs do not use string literals or CHAR/VARCHAR SQL parameters in integer expressions, as in this example:

```
DEFINE pv CHAR(1)
CREATE TABLE mytable ( v1 INT, v2 INT )
LET pv = '1'
SELECT * FROM mytable WHERE v1 = '1' AND v2 = pv
```

DATE and DATETIME data types

Informix® provides two data types to store dates and time information:

- DATE = for year, month and day storage.
- DATETIME = for year to fraction(1-5) storage.

Sybase ASE provides these data type to store dates:

- DATE = for year, month, day storage.
- TIME = for hour, minutes, seconds, fraction(3) storage.
- SMALLDATETIME = for hour, minutes, seconds, fraction(3) storage.
- DATETIME = for hour, minutes, seconds, fraction(3) storage.
- BIGTIME = for hour, minutes, seconds, fraction(6) storage.
- BIGDATETIME = for year, month, day, hour, minutes, seconds, fraction(6) storage.

String representing date time information

Informix® is able to convert quoted strings to DATE / DATETIME data if the string contents matches environment parameters (i.e. DBDATE, GL_DATETIME). As in Informix®, Sybase ASE can convert quoted strings representing datetime data in the ANSI format. The CONVERT() SQL function allows you to convert strings to dates.

Date time arithmetic

- Informix® supports date arithmetic on DATE and DATETIME values. The result of an arithmetic expression involving dates/times is a number of days when only DATEs are used and an INTERVAL value if a DATETIME is used in the expression.
- Informix® automatically converts an integer to a date when the integer is used to set a value of a date column. Sybase ASE does not support this automatic conversion.
- Complex DATETIME expressions (involving INTERVAL values for example) are Informix® specific and have no equivalent in Sybase ASE.
- With Sybase ASE you must use built-in functions to do date/time computing (for example, see dateadd() function).
- Informix® converts automatically an integer to a date when the integer is used to set a value of a date column. Sybase ASE does not support this automatic conversion.

Solution

Sybase ASE has the same DATE data type as Informix® (year, month, day). So you can use Sybase ASE DATE data type for Informix® DATE columns.

Sybase ASE BIGTIME data type can be used to store Informix® DATETIME HOUR TO SECOND and DATETIME HOUR TO FRACTION(5) values. The database interface makes the conversion automatically.

Informix® DATETIME values with any precision from YEAR to FRACTION(5) can be stored in Sybase ASE BIGDATETIME columns. The database interface makes the conversion automatically. Missing date or time parts default to 1900-01-01 00:00:00.0. For example, when using a DATETIME HOUR TO MINUTE with the value of "11:45", the ASE TIMESTAMP value will be "1900-01-01 11:45:00.0".

See also Date and time in SQL statements on page 432 for good SQL programming practices.

INTERVAL data type

Informix's INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: year-month intervals and day-time intervals.

Sybase ASE does not provide a data type corresponding to the Informix® INTERVAL data type.

Solution

The INTERVAL data type is not well supported because the database server has no equivalent native data type. However, you can store into and retrieve from CHAR columns BDL INTERVAL values.

SERIAL data type

Informix® supports the SERIAL, SERIAL8 and BIGSERIAL data types to produce automatic integer sequences. SERIAL is based on INTEGER (32 bit), while SERIAL8 and BIGSERIAL can store 64 bit integers:

- The table column must be of type SERIAL, SERIAL8 or BIGSERIAL.
- To generate a new serial, no value or a zero value is specified in the INSERT statement:

```
INSERT INTO tab1 ( c ) VALUES ( 'aa' )
INSERT INTO tab1 ( k, c ) VALUES ( 0, 'aa' )
```

- After INSERT, the new SERIAL value is provided in SQLCA.SQLERRD[2], while the new SERIAL8 and BIGSERIAL value must be fetched with a SELECT dbinfo('bigserial') query.

Informix® allows you to insert rows with a value different from zero for a serial column. Using an explicit value will automatically increment the internal serial counter, to avoid conflicts with future INSERTs that are using a zero value:

```
CREATE TABLE tab ( k SERIAL); -- internal counter = 0
INSERT INTO tab VALUES ( 0 ); -- internal counter = 1
```

```
INSERT INTO tab VALUES ( 10 ); -- internal counter = 10
INSERT INTO tab VALUES ( 0 ); -- internal counter = 11
DELETE FROM tab; -- internal counter = 11
INSERT INTO tab VALUES ( 0 ); -- internal counter = 12
```

Sybase ASE IDENTITY columns:

- When creating a table, the IDENTITY keyword must be specified after the column data type:

  ```
  CREATE TABLE tab1 ( k integer identity, c char(10) )
  ```

- You cannot specify a start value
- A new number is automatically created when inserting a new row:

  ```
  INSERT INTO tab1 ( c ) VALUES ( 'aaa' )
  ```

- To get the last generated number, Sybase ASE provides a global variable:

  ```
  SELECT @@IDENTITY
  ```

- When IDENTITY_INSERT is ON, you can set a specific value into a IDENTITY column, but zero does not generate a new serial:

  ```
  SET IDENTITY_INSERT tab1 ON
  INSERT INTO tab1 ( k, c ) VALUES ( 100, 'aaa' )
  ```

Informix® SERIALs and MS Sybase ASE IDENTITY columns are quite similar; the main difference is that MS Sybase ASE does not generate a new serial when you specify a zero value for the identity column.

Solution

With Sybase ASE, the SERIAL emulation can use IDENTITY columns (1) or insert triggers based on the SERIALREG table (2). The first solution is faster, but does not allow explicit serial value specification in insert statements; the second solution is slower but allows explicit serial value specification. You can initially use the second solution to have unmodified BDL programs working on Sybase ASE, but you should update your code to use native IDENTITY columns for performance.

The method used to emulate SERIAL types is defined by the `ifxemul.datatype.serial.emulation` FGLPROFILE parameter:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial.emulation = {"native"|"regtable"}`

- native: uses IDENTITY columns.
- regtable: uses insert triggers with the SERIALREG table.

The default emulation technique is "`native`".

This entry must be used in conjonction with:

`dbi.database.`*`dbname`*`.ifxemul.datatype.serial = {true|false}`

If the `datatype.serial` entry is set to false, the emulation method is ignored.

Using the `native` serial emulation

In database creation scripts, all SERIAL data types must be converted by hand to INTEGER IDENTITY data types, while BIGSERIAL must be converted to BIGINT IDENTITY.

Start values SERIAL(n) / BIGSERIAL(n) cannot be converted, there is no INTEGER IDENTITY(n) in Sybase ASE.

Tables created from the BDL programs can use the SERIAL data type: When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER IDENTITY[(n,1)]".

In BDL, the new generated SERIAL value is available from the SQLCA.SQLERRD[2] variable. This is supported by the database interface which performs a "SELECT @@IDENTITY". However, SQLCA.SQLERRD[2] is defined as an INTEGER, it cannot hold values from BIGINT identity columns. If you are using BIGINT IDENTITY columns, you must use @@IDENTITY.

When you insert a row with zero as serial value, the serial column gets the value zero. You must review all INSERT statements using zero for the serial column. For example, the following statement:

```
INSERT INTO tab (col1, col2)VALUES (0, p_value)
```

must be converted to:

```
INSERT INTO tab (col2) VALUES (p_value)
```

Static SQL INSERT using records defined from the schema file must also be reviewed:

```
DEFINE rec LIKE tab.*
INSERT INTO tab VALUES ( rec.*) -- will use the serial column
```

can be converted to:

```
INSERT INTO tab VALUES rec.* -- without braces, serial column is removed
```

Using the `regtable` serial emulation

First, you must prepare the database and create the SERIALREG table as follows:

```
CREATE TABLE serialreg (
    tablename VARCHAR(50) NOT NULL,
    lastserial BIGINT NOT NULL,
    PRIMARY KEY ( tablename )
)
```

In database creation scripts, all SERIAL[(n)] data types must be converted to INTEGER data types, BIGSERIAL column types must be changed to BIGINT, and you must create one trigger for each table. To know how to write those triggers, you can create a small Genero program that creates a table with a SERIAL column. Set the FGLSQLDEBUG environment variable and run the program. The debug output will show you the native trigger creation command.

Tables created from the BDL programs can use the SERIAL data type. When a BDL program executes a CREATE [TEMP] TABLE with a SERIAL column, the database interface automatically converts the "SERIAL[(n)]" data type to "INTEGER" and creates the insert triggers.

Sybase ASE does not allow you to create triggers on temporary tables. Therefore, you cannot create temp tables with a SERIAL column when using this solution.

Note:

- SELECT ... INTO TEMP statements using a table created with a SERIAL column do not automatically create the SERIAL triggers in the temporary table. The type of the column in the new table is INTEGER.
- Sybase ASE triggers are not automatically dropped when the corresponding table is dropped. Database administrators must be aware of this behavior when managing schemas.
- INSERT statements using NULL for the SERIAL column will produce a new serial value:

  ```
  INSERT INTO tab ( col1, col2 ) VALUES ( NULL, 'data' )
  ```

This behavior is mandatory in order to support INSERT statements which do not use the serial column:

```
INSERT INTO tab (col2) VALUES('data')
```

Check if your application uses tables with a SERIAL column that can contain a NULL value.

- The serial production is based on the SERIALREG table which registers the last generated number for each table. If you delete rows of this table, sequences will restart at 1 and you will get unexpected data.

ROWIDs

When creating a table, Informix® automatically adds a "ROWID" integer column (applies to non-fragmented tables only). The ROWID column is auto-filled with a unique number and can be used like a primary key to access a given row.

Sybase ASE tables have no ROWIDs.

Solution

If the BDL application uses ROWIDs, the program logic should be reviewed in order to use the real primary keys (usually, serials which can be supported).

All references to SQLCA.SQLERRD[6] must be removed because this variable will not hold the ROWID of the last INSERTed or UPDATEd row when using the Sybase ASE interface.

Case sensitivity

In Informix®, database object names like table and column names are not case sensitive:

```
CREATE TABLE Customer ( Custno INTEGER, ... )
SELECT CustNo FROM cuSTomer ...
```

In Sybase ASE, database object names **and character data** are case-insensitive by default:

```
CREATE TABLE Customer ( Custno INTEGER, CustName CHAR(20) )
INSERT INTO CUSTOMER VALUES ( 1, 'TECHNOSOFT' )
SELECT CustNo FROM cuSTomer WHERE custname = 'techNOSoft'
```

Solution

When you create a Sybase ASE database with `dbinit`, you can use the -c option to make the database case-sensitive.

Large OBject (LOB) types

IBM® Informix® and Genero support the TEXT and BYTE types to store large objects: TEXT is used to store large text data, while BYTE is used to store large binary data like images or sound.

Sybase ASE provides the TEXT and IMAGE data types for large objects storage.

Important: Sybase ASE 16.0 does not support TEXT/IMAGE expressions in WHERE clauses.

The ASE driver is implemented with the Sybase Open Client Library C API. In Sybase version 16.0, this API has a limited support for LOBs, especially when it comes to update LOB data in the database: You cannot directly INSERT large LOB data, you must first INSERT nulls and then UPDATE the row with the real data. Additionally, UPDATE can only take one LOB parameter at a time. Fetching LOB data is supported, with the following limitation: LOB columns must appear at the end of the SELECT list.

Solution

TEXT and BYTE character data types are supported by the Sybase ASE database interface, with some limitation.

When INSERTing TEXT/BYTE in a table, you must first insert with nulls, the update the new row, and only with one TEXT/BYTE parameter at a time:

```
DEFINE ptext TEXT, pbyte BYTE
...
LOCATE ptext IN ...
LOCATE pbyte IN ...
CREATE TABLE tab (k INT, t TEXT, b BYTE)
-- First INSERT a new row with NULLs
INSERT INTO tab VALUES (123,null,null)
-- Then UPDATE first TEXT column
UPDATE tab SET t = ptext WHERE k = 123
-- Then UPDATE second BYTE column
UPDATE tab SET b = pbyte WHERE k = 123
```

Fetching TEXT and BYTE columns is possible as long as the columns appear at the end of the SELECT list. For example, if you have a statement such as (where pdata is a TEXT or BYTE column):

```
SELECT pid, pdata, ptimestamp FROM pic WHERE ...
```

Put the BYTE column at the end of the SELECT list:

```
SELECT pid, ptimestamp, pdata FROM pic WHERE ...
```

The ALTER TABLE instruction

Informix® and MS Sybase ASE use different implementations of the ALTER TABLE instruction. For example, Informix® allows you to use multiple ADD clauses separated by comma. This is not supported by Sybase ASE:

Informix®:

```
ALTER TABLE customer ADD(col1 INTEGER), ADD(col2 CHAR(20))
```

Sybase ASE:

```
ALTER TABLE customer ADD col1 INTEGER, col2 CHAR(20)
```

Solution

No automatic conversion is done by the database interface. There is no real standard for this instruction (that is, no common syntax for all database servers). Read the SQL documentation and review the SQL scripts or the BDL programs in order to use the database server specific syntax for ALTER TABLE.

Constraints

Constraint naming syntax

Both Informix® and Sybase ASE support primary key, unique, foreign key, default and check constraints. But Sybase ASE does not support constraint naming syntax:

```
CREATE TABLE emp (
   emp_code CHAR(10) UNIQUE CONSTRAINT pk_emp,
   ... )
```

Solution: Constraint naming syntax

The database interface does not convert constraint naming expressions when creating tables from BDL programs. Review the database creation scripts to adapt the constraint naming clauses for Sybase ASE.

Triggers

Informix® and Sybase ASE provide triggers with similar features, but the programming languages are totally different.

Sybase ASE does not support triggers on temporary tables.

Solution

Informix® triggers must be converted to Sybase ASE triggers "by hand".

Stored procedures

Both Informix® and Sybase ASE support stored procedures, but the programming languages are totally different.

Solution

Informix® stored procedures must be converted to Sybase ASE"by hand".

See SQL Programming for more details about executing stored procedures with Sybase ASE.

Name resolution of SQL objects

Informix® uses the following form to identify an SQL object:

```
[database[@dbservername]:][{owner|"owner"}.]identifier
```

With Sybase ASE, an object name takes the following form:

```
[{database|[database]}.[{owner|[owner]}.]]{identifier|[identifier]}
```

Informix® database object names are **not case sensitive** in non-ANSI databases.

Sybase ASE database objects names are **case sensitive** by default.

Solution

As a general rule, to write portable SQL, you should only use simple database object names without any database, server or owner qualifier and without quoted identifiers.

Always create and use tables and columns names in lower case.

Data type conversion table: Informix to Sybase ASE

Table 219: Data type conversion table (Informix to Sybase ASE)

Informix® data types	Sybase ASE data types
CHAR(n)	CHAR(n) (limit = page size, ex:16384 bytes)
VARCHAR(n[,m])	VARCHAR(n) (limit = page size, ex:16384 bytes)
LVARCHAR(n)	VARCHAR(n) (limit = page size, ex:16384 bytes)
NCHAR(n)	NCHAR(n) (length in characters)
NVARCHAR(n[,m])	NVARCHAR(n) (length in characters)
BOOLEAN	BIT (must be NOT NULL!)
SMALLINT	SMALLINT

Informix® data types	**Sybase ASE data types**
INT / INTEGER	INTEGER
BIGINT	BIGINT
INT8	BIGINT
SERIAL without start value!	INTEGER (see note 1)
BIGSERIAL without start value!	BIGINT (see note 1)
SERIAL8 without start value!	BIGINT (see note 1)
DOUBLE PRECISION / FLOAT[(n)]	DOUBLE PRECISION
REAL / SMALLFLOAT	REAL
NUMERIC / DEC / DECIMAL(p,s)	DECIMAL(p,s)
NUMERIC / DEC / DECIMAL(p) with p<=19	DECIMAL(2*p,p)
NUMERIC / DEC / DECIMAL(p) with p>19	N/A
NUMERIC / DEC / DECIMAL	DECIMAL(32,16)
MONEY(p,s)	DECIMAL(p,s)
MONEY(p)	DECIMAL(p,2)
MONEY	DECIMAL(16,2)
DATE	DATE(yyyy-mm-dd)
DATETIME HOUR TO FRACTION(n)	BIGTIME(hh:mm:ss.ffffff)
DATETIME HOUR TO SECOND	BIGTIME(hh:mm:ss.ffffff)
Other sort of DATETIME type	BIGDATETIME(yyyy-mm-dd hh:mm:ss.ffffff)
INTERVAL q1 TO q2	CHAR(50)
TEXT	TEXT
BYTE	IMAGE

Notes:

1. For more details about serial emulation, see SERIAL data type on page 733.

Data manipulation
Sybase ASE related data manipulation topics.
Reserved words

Even if Sybase ASE allows SQL reserved keywords as SQL object names if enclosed in square braces (create table [table] (col1 int)), you should take care of your existing database schema and check that you do not use Sybase ASE SQL words.

Solution

Database objects having a name which is a Sybase ASE SQL reserved word must be renamed.

All BDL application sources must be verified. To check if a given keyword is used in a source, you can use UNIX™ 'grep' or 'awk' tools. Most modifications can be automatically done with UNIX™ tools like 'sed' or 'awk'.

Outer joins

The original OUTER join syntax of Informix® is different from the Sybase ASE outer join syntax:

In Informix® SQL, outer tables can be defined in the **FROM** clause with the **OUTER** keyword:

```
SELECT ... FROM cust, OUTER(order)
 WHERE cust.key = order.custno

SELECT ... FROM cust, OUTER(order,OUTER(item))
 WHERE cust.key = order.custno
    AND order.key = item.ordno
    AND order.accepted = 1
```

Sybase ASE Version 7 supports the ANSI outer join syntax:

```
SELECT ... FROM cust LEFT OUTER JOIN order
                     ON cust.key = order.custno

SELECT ...
  FROM cust LEFT OUTER JOIN order
                 LEFT OUTER JOIN item
                 ON order.key = item.ordno
             ON cust.key = order.custno
 WHERE order.accepted = 1
```

The old way to define outer joins in Sybase ASE looks like the following:

```
SELECT ... FROM a, b WHERE a.key *= b.key
```

See the Sybase ASE reference manual for a complete description of the syntax.

Solution

For better SQL portability, you should use the ANSI outer join syntax instead of the old Informix® OUTER syntax.

The Sybase ASE interface can convert simple Informix® OUTER specifications to Sybase ASE ANSI outer joins.

Prerequisites:

1. The outer join in the WHERE part must use the table name as prefix.

   Example: `"WHERE tab1.col1 = tab2.col2 "`
2. Additional conditions on outer table columns cannot be detected and therefore are not supported:

   Example: `"... FROM tab1, OUTER(tab2) WHERE tab1.col1 = tab2.col2 AND tab2.colx > 10"`
3. Statements composed of 2 or more SELECT instructions using OUTERs are not supported.

   Example: `"SELECT ... UNION SELECT" or "SELECT ... WHERE col IN (SELECT...)"`

   Note:

   1. Table aliases are detected in OUTER expressions.

      OUTER example with table alias: `"OUTER( tab1 alias1)"`
   2. In the outer join, <outer table>.<col> can be placed on both right or left sides of the equal sign.

      OUTER join example with table on the left: `"WHERE outertab.col1 = maintab.col2 "`
   3. Table names detection is not case-sensitive.

      Example: `"SELECT ... FROM tab1, TAB2 WHERE tab1.col1 = tab2.col2"`

4. Temporary tables are supported in OUTER specifications.

Transactions handling

Informix® and Sybase ASE handle transactions in a similar manner.

Informix® native mode (non ANSI):

- Transactions are started with "BEGIN WORK".
- Transactions are validated with "COMMIT WORK".
- Transactions are canceled with "ROLLBACK WORK".
- Statements executed outside of a transaction are automatically committed.
- DDL statements can be executed (and canceled) in transactions.

Sybase ASE:

- Sybases supports two transaction modes:
  1. The SQL standards-compatible mode, called *chained* mode, to get implicit transaction.
  2. The default mode, called *unchained* mode, where transactions have to be started/ended explicitly.
- Transactions are started with "BEGIN TRANSACTION [name]".
- Transactions are validated with "COMMIT TRANSACTION [name]".
- Transactions are canceled with "ROLLBACK TRANSACTION [name]".
- Transactions save points can be placed with "SAVEPOINT [name]".
- Sybase ASE supports named and nested transactions.
- DDL statements can be executed in transactions blocks when the 'ddl in tran' option is set to true with:

```
master..sp_dboption dbname, 'ddl in tran', true
go
checkpoint
go
```

Solution

Informix® transaction handling commands are automatically converted to Sybase ASE instructions to start, commit or rollback transactions.

Make sure that the database uses the default *unchained* mode (set chained off) and allows DDLs in transactions ('ddl in tran' option is true).

Regarding the transaction control instructions, the BDL applications do not have to be modified in order to work with Sybase ASE.

Temporary tables

Informix® temporary tables are created through the CREATE TEMP TABLE DDL instruction or through a SELECT ... INTO TEMP statement. Temporary tables are automatically dropped when the SQL session ends, but they can also be dropped with the DROP TABLE command. There is no name conflict when several users create temporary tables with the same name.

The CREATE TEMP TABLE and SELECT INTO TEMP statements are not supported in Sybase ASE.

Sybase ASE supports temporary tables by using the # pound sign before the table name:

```
CREATE TABL #temp1 ( kcol INTEGER, .... )
SELECT * INTO #temp2 FROM customers WHERE ...
```

Solution

In BDL, Informix® temporary tables instructions are converted to generate native Sybase ASE temporary tables.

SELECT INTO TEMP statements cannot be converted, because Sybase ASE does not provide a way to create a temporary table from a result set, such as CREATE TABLE xx AS (SELECT ...).

Substrings in SQL

Informix® SQL statements can use subscripts on columns defined with the character data type:

```
SELECT ... FROM tab1 WHERE  col1[2,3] = 'RO'
SELECT ... FROM tab1 WHERE  col1[10] = 'R' -- Same as col1[10,10]
UPDATE tab1 SET  col1[2,3] = 'RO' WHERE ...
SELECT ... FROM tab1 ORDER BY  col1[1,3]
```

Sybase ASE provides the SUBSTRING() function, to extract a substring from a string expression:

```
SELECT .... FROM tab1 WHERE  SUBSTRING(col1,2,2)  = 'RO'
SELECT SUBSTRING('Some text',6,3 ) FROM DUAL -- Gives 'tex'
```

Solution

You must replace all Informix® col[x,y] expressions by SUBSTRING(col from x for (y-x+1)).

Note:

- In UPDATE instructions, setting column values through subscripts will produce an error with PostgreSQL:

  `UPDATE tab1 SET col1[2,3] = 'RO' WHERE ...`

  is converted to:

  `UPDATE tab1 SET SUBSTRING(col1 from 2 for (3-2+1)) = 'RO' WHERE ...`
- Column subscripts in ORDER BY expressions are also converted and produce an error with PostgreSQL:

  `SELECT ... FROM tab1 ORDER BY col1[1,3]`

  is converted to:

  `SELECT ... FROM tab1 ORDER BY SUBSTRING(col1 from 1 for(3-1+1))`

String delimiters

The ANSI string delimiter character is the single quote ('string'). Double quotes are used to delimit database object names ("object-name").

Example: `WHERE"tabname"."colname" = 'a string value'`

As Informix®, Sql Server Anywhere allows to use double quotes as string delimiters, if the QUOTED_IDENTIFIER session option is OFF (the default):

`SET QUOTED_IDENTIFIER OFF`

Remark: This problem concerns only double quotes within SQL statements. Double quotes used in BDL string expressions are not subject of SQL compatibility problems.

Solution

When the `dbi.database.`*`dbname`*`.ifxemul.dblquotes` FGLPROFILE option is set, the Sybase ASE database interface converts all double quotes to single quotes in SQL statements. The Sybase ASE database driver does not set the `QUOTED_IDENTIFIER` option implicitly.

Getting one row with SELECT

With Informix®, you must use the system table with a condition on the table id:

`SELECT user FROM systables WHERE tabid=1`

With Sybase ASE, you can omit the FROM clause to generate one row only:

```
SELECT user
```

Solution

Check the BDL sources for "FROM systables WHERE tabid=1" and use dynamic SQL to resolve this problem.

MATCHES and LIKE in SQL conditions

Informix® supports MATCHES and LIKE in SQL statements, while Sybase ASE supports the LIKE statement only.

The MATCHES operator of Informix® uses the star (*), question mark (?) and square braces ([ ]) wildcard characters. The LIKE operator of SQL SERVER offers the percent (%), underscore (_) and square braces ([ ]) wildcard characters:

```
( col MATCHES 'Smi*' AND col NOT MATCHES 'R?x[a-z]' )
( col LIKE 'Smi%' AND col NOT LIKE 'R_x[a-z]' )
```

Solution
The database driver is able to translate Informix® MATCHES expressions to LIKE expressions, when no [ ] bracket character ranges are used in the MATCHES operand.

However, for maximum portability, consider replacing the MATCHES expressions to LIKE expressions in all SQL statements of your programs.

Avoid using CHAR(N) types for variable length character data (such as name, address).

See also: MATCHES and LIKE operators on page 438.

Querying system catalog tables

As in Informix®, Sybase ASE provides system catalog tables (sysobjects, syscolumns, etc.) in each database, but the table names and their structure are quite different.

Solution

No automatic conversion of Informix® system tables is provided by the database interface.

Syntax of UPDATE statements

Informix® allows a specific syntax for UPDATE statements:

```
UPDATE table SET ( col-list ) = ( val-list )
```

Or

```
UPDATE table SET table.* = codeph.*
```

```
UPDATE table SET * = myrecord.*
```

Solution

Static UPDATE statements using this syntax are converted **by the compiler** to the standard form:

```
UPDATE table SET column=value [,...]
```

BDL programming
Sybase ASE related programming topics.
Informix-specific SQL statements in BDL

The BDL compiler supports several Informix® specific SQL statements that have no meaning when using Sybase ASE.

Examples:

- CREATE DATABASE dbname IN dbspace WITH BUFFERED LOG
- START DATABASE (SE only)
- ROLLFORWARD DATABASE
- CREATE TABLE ... IN dbspace WITH LOCK MODE ROW

Solution

Review your BDL source and remove all static SQL statements that are Informix-specific.

Insert cursors

Informix® supports insert cursors. An "insert cursor" is a special BDL cursor declared with an INSERT statement instead of a SELECT statement. When this kind of cursor is open, you can use the PUT instruction to add rows and the FLUSH instruction to insert the records into the database.

For Informix® database with transactions, OPEN, PUT and FLUSH instructions must be executed within a transaction.

Sybase ASE does not support insert cursors.

Solution

Insert cursors are emulated by the Sybase ASE database interface.

Cursors WITH HOLD

Informix® automatically closes opened cursors when a transaction ends unless the WITH HOLD option is used in the DECLARE instruction.

Sybase ASE does not close cursors when a transaction ends, as long as the global parameter close_on_endtrans is off.

Solution

BDL cursors that are not declared "WITH HOLD" are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK is performed by the BDL program.

SELECT FOR UPDATE

A lot of BDL programs use pessimistic locking in order to avoid several users editing the same rows at the same time.

```
DECLARE cc CURSOR FOR
SELECT ... FROM tab WHERE ... FOR UPDATE
OPEN cc
FETCH cc <-- lock is acquired
...
CLOSE cc <-- lock is released
```

- A transaction must be started before opening cursors declared for update.
- The row must be fetched in order to set the lock.
- The lock is released when the transaction ends (if the cursor is not declared "WITH HOLD") or when the cursor is closed.

Sybase ASE ignores the FOR UPDATE clause when not used in a native Sybase SQL DECLARE command. In order to lock rows when doing a SELECT, with Sybase you must add the **holdlock** hint or the **at isolation repeatable read** clause. Sybase supports SELECT locking outside transactions (i.e. WITH HOLD cursors).

- Locks are acquired when opening the cursor.
- When the cursor (WITH HOLD) is opened outside a transaction, locks are released when the cursor is closed.
- When the cursor is opened inside a transaction, locks are released when the transaction ends.

Sybase ASE's locking granularity is at the row level, page level or table level (the level is automatically selected by the engine for optimization).

To control the behavior of the program when locking rows, Informix® provides a specific instruction to set the wait mode:

```
SET LOCK MODE TO { WAIT | NOT WAIT | WAIT seconds}
```

The default mode is WAIT. SET LOCK MODE is as an Informix® specific SQL statement which is translated by the driver.

Solution

SELECT FOR UPDATE statements are supported: The Sybase ASE driver adds the "**at isolation repeatable read**" keywords to the end of any SELECT FOR UPDATE statement.

Sybase ASE requires a PRIMARY KEY or UNIQUE INDEX on the table using in the SELECT .. FOR UPDATE statement.

Sybase ASE locks the rows when you open the cursor. You will have to test SQLCA.SQLCODE after doing an OPEN.

The database interface is based on an emulation of an Informix® engine using transaction logging. Therefore, opening a SELECT ... FOR UPDATE cursor declared outside a transaction will raise an SQL error -255 (not in transaction).

The SELECT FOR UPDATE statement cannot contain an ORDER BY clause if you want to perform positioned updates/deletes with WHERE CURRENT OF.

The LOAD and UNLOAD instructions

Informix® provides two SQL instructions to export / import data from / into a database table: The UNLOAD instruction copies rows from a database table into an text file and the LOAD instruction inserts rows from an text file into a database table.

Sybase ASE has LOAD and UNLOAD instructions, but those commands are related to database backup and recovery. Do not confuse with Informix® commands.

Solution

LOAD and UNLOAD instructions are supported.

The LOAD instruction does not work with tables using emulated SERIAL columns because the generated INSERT statement holds the "SERIAL" column which is actually a IDENTITY column in Sybase ASE. See the limitations of INSERT statements when using SERIALs.

In Sybase ASE, Informix® DATETIME data is stored in BIGDATETIME columns, but DATETIME columns are similar to Informix® DATETIME YEAR TO FRACTION(5) columns. Therefore, when using LOAD and UNLOAD, those columns are converted to text data with the format "YYYY-MM-DD hh:mm:ss.fffff".

SQL Interruption

With Informix®, it is possible to interrupt a long running query if the SQL INTERRUPT ON option.

Solution

The Sybase ASE database driver supports SQL interruption and raises error code -213 if the statement is interrupted.

Scrollable Cursors

The Genero programming language supports scrollable cursors.

Sybase ASE supports native scrollable cursors.

Solution

The Sybase ASE database driver uses the native Sybase ASE Open Client Library scrollable cursors.

User interface

These topics cover programming the user interface (UI) with the Genero Business Development Language.

- User interface basics on page 747
- Form definitions on page 769
- Dialog instructions on page 1034
- User interface programming on page 1249

User interface basics

This section introduces to the foundation of the Genero user interface.

- The user interface on page 27
- Genero user interface modes on page 752
- The dynamic user interface on page 747
- Establish a GUI front-end connection on page 755
- The abstract user interface tree on page 749
- Special user interface features on page 759
- Configuring a text terminal on page 762

The dynamic user interface

The dynamic user interface is the base concept of the Genero user interaction components.

The *dynamic user interface* (DUI) concept implements a flexible graphical user interface programming toolkit, based on the usage of XML standards to define an abstract representation of the application forms, that can be displayed by different sort of display devices called front ends, which execute on the user workstation or on the same platform as the runtime system.

By using the same program source code, the abstract definition of the user interface that can be manipulated at runtime as a tree of interface objects. This tree is called the abstract user interface tree.

The runtime system is in charge of the abstract user interface tree and the front end is in charge of rendering this abstract tree visible on the screen. The front end gets a copy of that tree which is automatically synchronized by the runtime system by using the front end protocol.

In development, application screens are defined by form specification files. These files are compiled by the fglform form compiler to produce the runtime form files that can be deployed in production environments.

The following schema describes the dynamic user interface concept, showing how the abstract user Interface tree is shared by the runtime system and the front end.

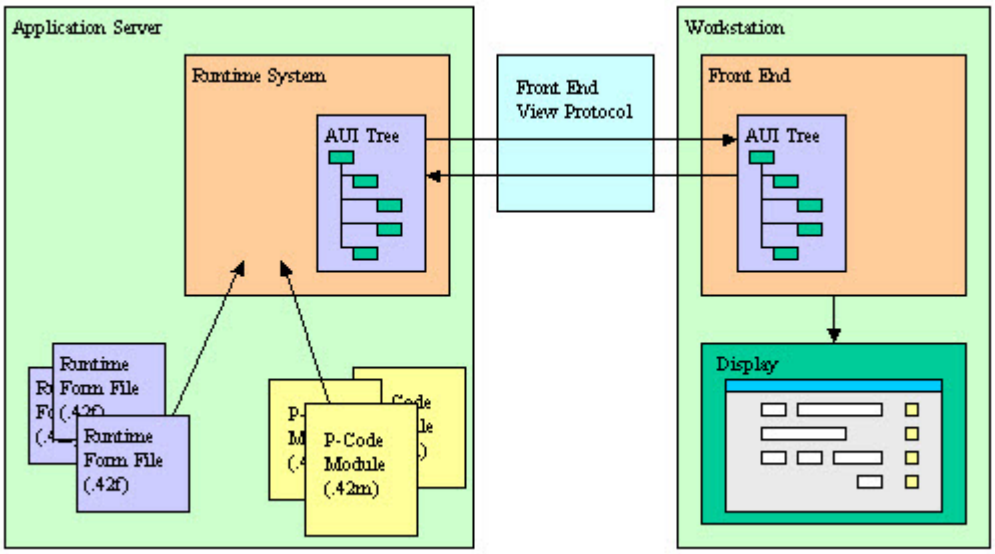


Figure 35: AUI tree shared between the runtime system and front end

The abstract user interface tree (AUI tree) on the front-end is synchronized with the runtime system AUI tree when a user interaction instruction takes the control. This means that the user will not see any display as long as the program is doing batch processing, until an interactive statement is reached.

For example, the following program shows nothing:

```
MAIN
  DEFINE cnt INTEGER
  OPEN WINDOW w WITH FORM "myform"
  FOR cnt=1 TO 10
    DISPLAY BY NAME cnt
    SLEEP 1
  END FOR
END MAIN
```

If you want to show something on the screen while the program is running in a batch procedure, you must force synchronization with the front-end, by calling the `refresh()` method of the `ui.Interface` built-in class:

```
MAIN
  DEFINE cnt INTEGER
  OPEN WINDOW w WITH FORM "myform"
  FOR cnt=1 TO 10
    DISPLAY BY NAME cnt
    CALL ui.Interface.refresh()   -- Sync the front-end!
    SLEEP 1
  END FOR
END MAIN
```

Note that the `refresh()` method must only be called when really needed: By default, the AUI tree will be automatically synchronized when the control goes back to the runtime system dialog instruction.

Keep in mind that when the AUI trees are synchronized, only the changes are sent to the front-end. If a modification has been made that does not result in a change in the values of the attributes of a node of the tree (for example, you change the contents of an image file but keep the same name), that modification will not be sent to the front-end.

Note that when running on a mobile device, both front-end and runtime system execute on the same platform. Still the AUI tree protocol takes place, and both component perform the tasks they are dedicated to.

The abstract user interface tree

The abstract user interface tree is the XML representation of the application forms displayed to the end user.

The *abstract user interface tree* (AUI tree) is a DOM tree describing the objects of the user interface elements of a program at a given time.

A copy of the AUI tree is held by both the front end and the runtime system.

AUI tree synchronization is automatically done by the runtime system using the front end protocol.

The programs can manipulate the AUI tree element by using XML utility classes or high-level built-in classes such as `ui.Dialog` and `ui.Form`.

What does the abstract user interface tree contain?

The abstract user interface defines a tree of objects organized by parent/child relationship. The different kinds of user interface objects are defined by attributes. The AUI tree can be serialized as text according to the XML standard notation.

The following example shows a part of an AUI tree defining a toolbar serialized with the XML notation:

```
<ToolBar>
  <ToolBarItem name="f5" text="List" image="list" />
  <ToolBarSeparator/>
  <ToolBarItem name="Query" text="Query" image="search" />
  <ToolBarItem name="Add" text="Append" image="add" />
  ...
</ToolBar>
```

Manipulating the abstract user interface tree

Modifying the AUI tree with user interface specific built-in classes

The objects of the abstract user interface tree can be queried and modified at runtime with specific built-in classes like `ui.Form`, provided to manipulate form elements.

The next code example gets the current window object, then gets the current form in that window, and hides a group-box form element identified by the name "`gb1`":

```
DEFINE w ui.Window
DEFINE f ui.Form
LET w = ui.Window.getCurrent()
LET f = w.getForm()
CALL f.setElementHidden("gb1",1)
```

Using the user interface specific built-in classes is the recommended way to modify the AUI tree in your programs.

Using low-level APIs to modify the AUI tree

In very special cases, you can also directly access the nodes of the AUI tree by using DOM built-in classes like `om.DomDocument` and `om.DomNode`.

> **Important:** As we continue to add new features to the product we encounter situations that may force us to modify the AUI Tree in order to add new elements types and attributes. If you are using the low level API's to directly modify the tree, your code may be slightly impacted when we release

a change in the AUI Tree structure. In order to minimize the impact of any such AUI tree definition changes, we would like to suggest the following course of action with regards to use of the DOM/SAX API's:

1. Place all custom calls to the DOM/SAX API within centralized Library functions that are accessible to all modules, as opposed to scattering function calls throughout your code base.
2. Do not create nodes or change attributes that are not explicitly documented as modifiable. For example, `TopMenu` or `ToolBar` nodes can be created and configured dynamically, but you should not add `FormField` nodes to existing forms, or modify yourself the `active` attribute of fields or actions.

To get the user interface nodes at runtime, the language provides different kinds of API functions or methods, according to the context. For example, to get the root of the AUI tree, call the `ui.Interface.getRootNode()` method. You can also get the current form node with `ui.Form.getNode()` or search for an element by name with the `ui.Form.findNode()` method.

XML node types and attribute names

By tradition the language uses uppercase keywords, such as `LABEL` in form files, and the examples in this documentation reflect that convention. The language itself is not case-sensitive. However, XML is case-sensitive, and by convention node types use uppercase/lowercase combinations to indicate word boundaries. Therefore, the nodes and attributes of an abstract user interface tree are handled as follows:

- Node types - the first letter of the node type is always capitalized. Subsequent letters are lowercase, unless the type consists of multiple words joined together. In that case, the first letter of each of the multiple words is capitalized (the camel-case convention). Examples: `Label`, `FormField`, `DateEdit`, `Edit`.
- Attribute names - the first letter of the name is always lowercase; subsequent letters are also lowercase, unless the name consists of multiple words joined together. In that case, the first letter of each subsequent word is capitalized (the Lower camel-case convention). Examples: `text`, `colName`, `width`, `tabIndex`
- Attribute values - the values are enclosed in quotes, and the runtime system does not convert them.

If you reference AUI tree XML nodes or attributes in your code, you must always respect the naming conventions.

Actions in the abstract user interface tree

The abstract user interface identifies all possible actions that can be received by the current interactive instruction with a list of `Action` nodes. The list of possible actions are held by a `Dialog` node. An `Action` node is identified by the '`name`' attribute and defines common properties such as the accelerator key, default image, and default text.

Interactive elements are bound to `Action` nodes by the '`name`' attribute. For example, a toolbar button (a.k.a toolbar item) with the name 'cancel' is bound to the `Action` node having the name 'cancel', which in turn defines the accelerator key, the default text, and the default image for the button.

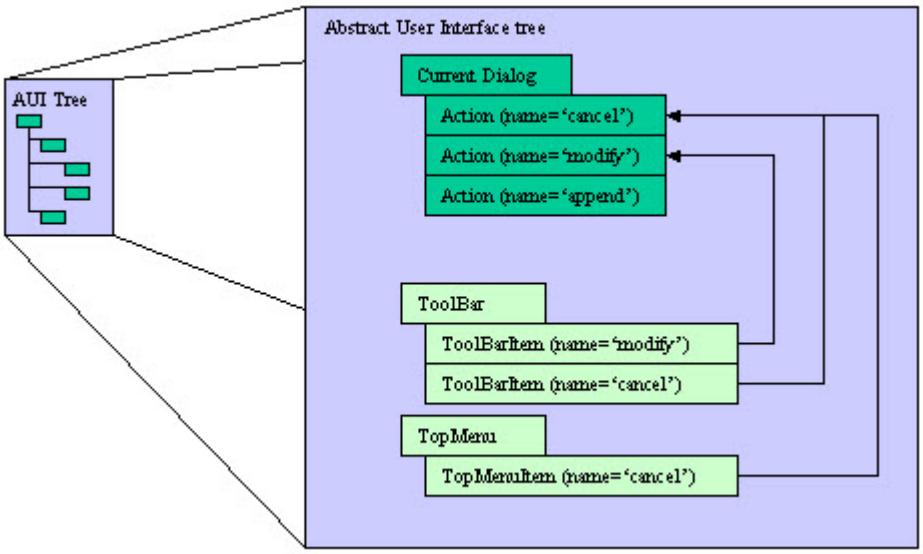


Figure 36: AUI Tree binding

When an interactive element is used (such as a form field input, toolbar button click, or menu option selection), an `ActionEvent` node is sent to the runtime system. The name of the `ActionEvent` node identifies what `Action` occurred and the '`idRef`' attribute indicates the source element of the action.

Inspecting the AUI tree of a front end

The abstract user interface tree build on a front end side can be inspected

When executing a program displaying on a front end, it is possible to inspect the content of the abstract user interface built on the front end side. The way to show the AUI tree depends on the type of front end.

Genero Desktop Client

The GDC must have been started in debug mode (`-aD` option).

In the current window of the running program, do a control-right-click with the mouse: This will open the AUI tree debug window.

You can then browse the AUI tree created on the GDC side.

Genero Web Client - JavaScript

The GAS / GWC-JS must have been started with debug option. In the `as.xcf` configuration file, add the following line:

```
<CONFIGURATION ...>
  <APPLICATION_SERVER>
     ...
     <RESOURCE Id="res.uaproxy.param" Source="INTERNAL">--development</
RESOURCE>
     ...
```

Start your application in a web browser: a debug icon should appear on the right of the window. Click the icon to display the AUI debug tree.

You can then browse the AUI tree created on the GMA side.

Genero Mobile for Android

The GMA must execute with debug mode enabled in the settings panel.

Open a web browser an enter the following URL:

```
http://device-ip-address:6480
```

You can then browse the AUI tree created on the GMA side.

Genero Mobile for iOS

The GMI must have been started in debug mode: the debug option needs to be enabled in GMI app settings on the device.

Open a web browser an enter the following URL:

```
http://device-ip-address:6480 (or 6400)
```

You can then browse the AUI tree created on the GMI side.

Genero user interface modes

User interface modes allow to adapt the application form rendering to different sort of displays.

There supported user interface modes are:

- Text mode rendering on page 752
- Graphical mode rendering on page 753
- Traditional GUI mode on page 753

Text mode rendering

The text user interface (TUI) has been designed for character-based terminals. This mode can be used to run your application on a text terminal hardware or in a terminal emulator.

In order to run a Genero program on text mode, set the FGLGUI environment variable to 0 (zero).

In TUI mode, the application windows/forms will display within the current console/terminal window as shown.

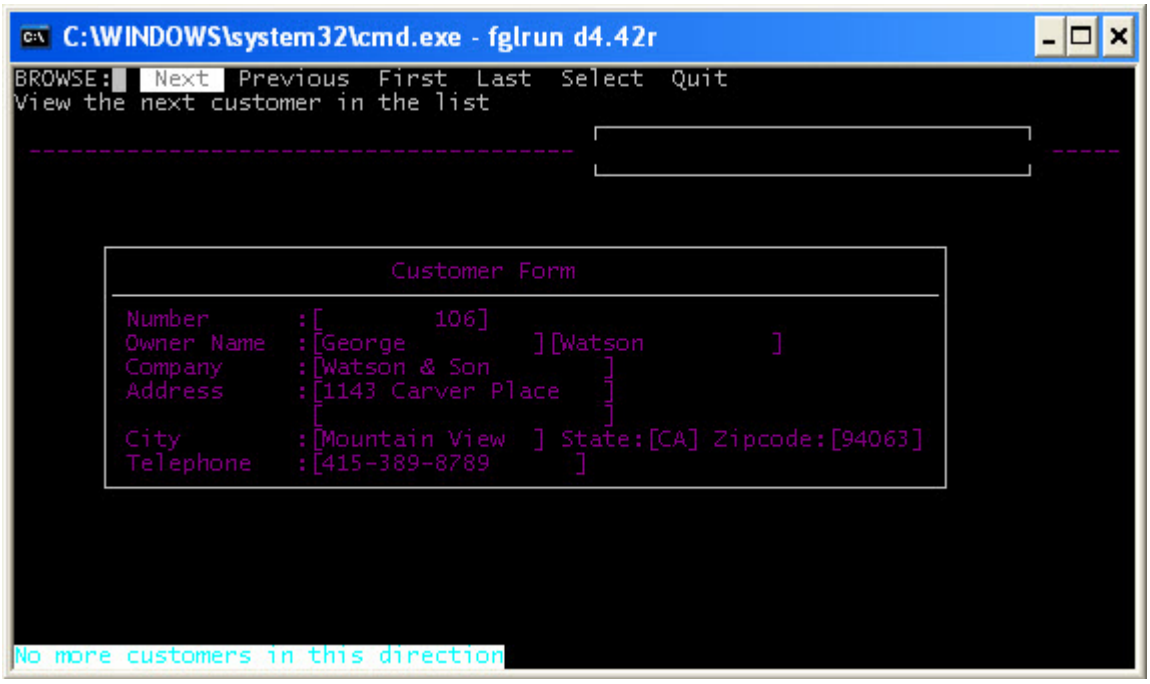


Figure 37: Text mode rendering

On UNIX™ platforms, you need to configure you terminal capabilities with environment variables with TERM, TERMINFO or TERMCAP environment variables.

Graphical mode rendering

Genero supports the graphical user interface (GUI) to provide a real graphical look and feel, for desktop workstation, web browsers and mobile front-end platforms.

When set to 1, the FGLGUI environment variable defines the graphical mode usage. This is the default. In graphical mode, the application forms are displayed on the front-end workstation identified with the FGLSERVER environment variable. Application forms will be rendered with real graphical widgets providing a nice look-and-feel as shown.

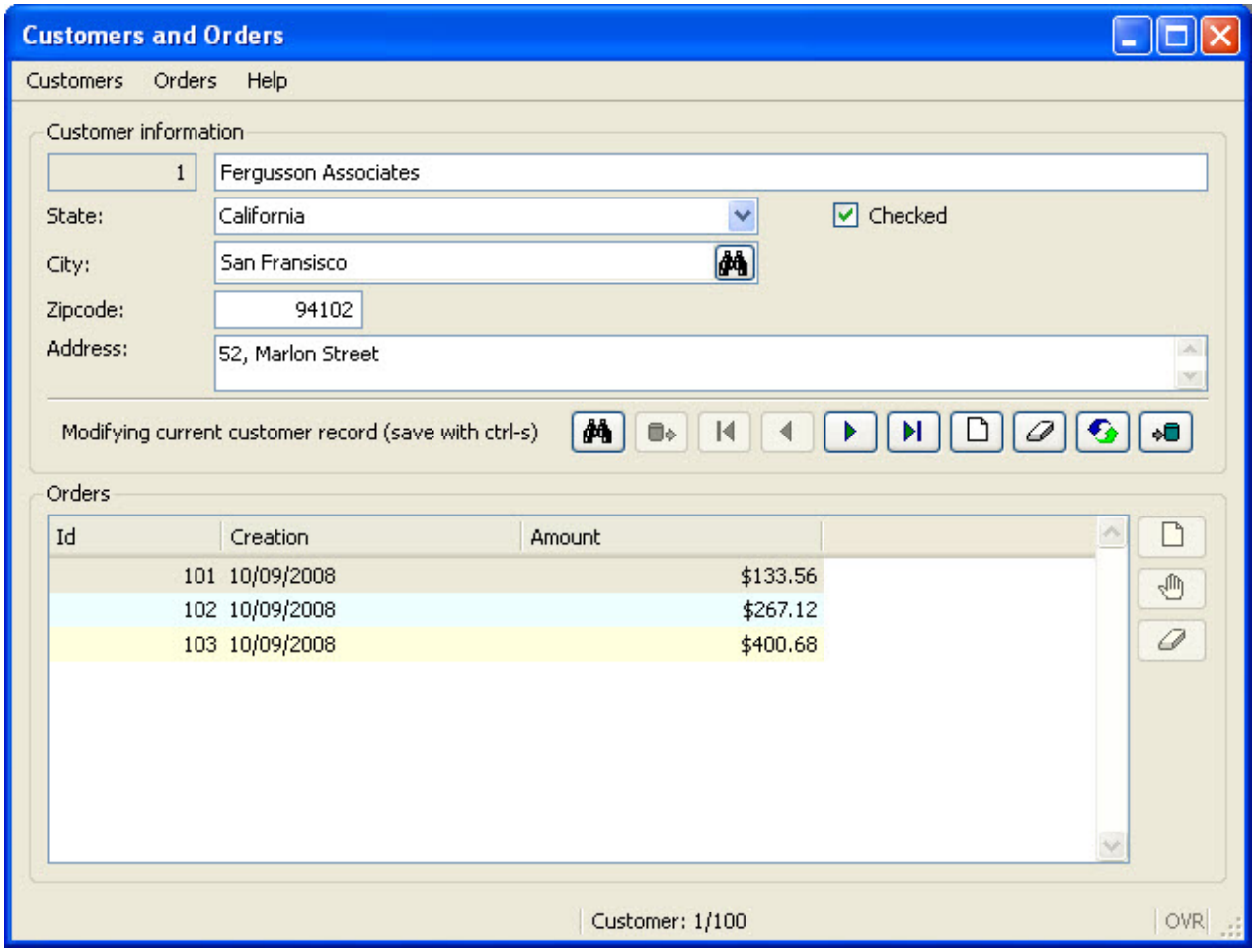


Figure 38: Graphical mode rendering

To simplify migration from text mode to graphical mode with legacy applications, you can use the traditional GUI mode option to render application windows in a single front end GUI window.

Traditional GUI mode

What is the Traditional GUI mode designed for?

With the graphical mode, you immediately get the benefit of standard GUI widgets and windows. Forms are rendered as real movable and re-sizeable windows, form labels and fields become widgets using variable fonts, toolbars and pull-down menus are displayed, and error messages are displayed in the status bar. However, that can be annoying if you have to migrate from a project that was developed for dumb terminals (i.e. TUI mode).

You can use the *traditional GUI mode* to ease migration from TUI based applications to GUI mode.

With the traditional mode, application windows bound to forms using a SCREEN section will be displayed as simple boxes in a main front end window. Other windows bound to forms defined with the LAYOUT section will be displayed as new GUI windows.

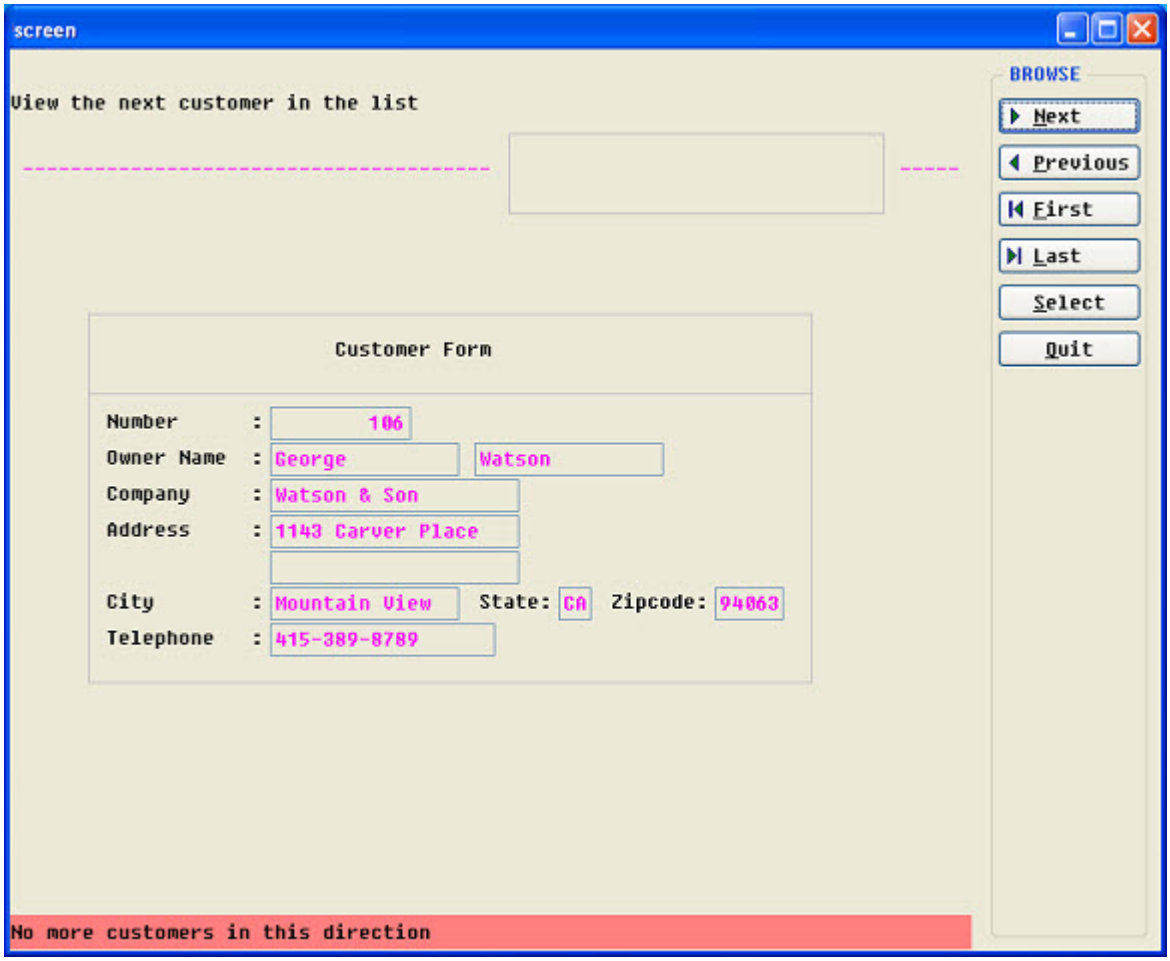


Figure 39: Traditional GUI mode rendering

Enabling the traditional GUI mode

The traditional GUI mode can be enabled with the following FGLPROFILE entry:

```
gui.uiMode = "traditional"
```

By default, the traditional GUI mode is off.

Window rendering rules

If the traditional GUI mode is enabled, the `OPEN WINDOW` statement works differently depending on the layout type of bound forms.

On the front end side, there is one unique main graphical window (a top-level widget called "*compatibility window container*") created to host all the windows created by a program. Traditional forms are form files which have a SCREEN section instead of the LAYOUT section. When migrating from an TUI mode project, all forms initially contain a SCREEN section; hence all windows opened in traditional mode will appear in the compatibility window container.

To rebuild a form file with graphical items such as group boxes, buttons and tables, use a `LAYOUT` section. If the rebuilt form file is loaded via `OPEN WINDOW ... WITH FORM form-file` then, even in traditional mode, the newly created window will appear as a new top-level widget on the front end side. This opens a smooth migration path using the traditional mode; as a first step, it is possible to migrate and enhance some application forms like typical search lists, while keeping the rest of the application forms running in the traditional rendering.

Note, however, that following instructions do not work in traditional GUI mode:

1. `OPEN WINDOW` *`window_id`* `AT` *`line, column`* `WITH` *`height`* `ROWS,` *`width`* `COLUMNS`
2. `OPEN FORM` *`form_id`* `FROM "`*`form_file`*`"`

   (where *form_file* is defined with a `LAYOUT` section)
3. `DISPLAY FORM` *`form_id`*

A runtime error results, because you cannot display a form with dynamic geometry in a fixed geometry container. Only forms with a `SCREEN` section can be displayed at a later stage in a window that was initially opened inside the compatibility window container.

Function key shifting

When the traditional mode is enabled, you can map Shift-Fx and Ctrl-Fx key strokes to F(x+offset) actions. The offset is defined with the `gui.key.add_function` entry:

```
gui.key.add_function = 12
```

This entry defines the number of function keys of the keyboard (default is 12). When defined as 12, a Shift-F1 will be received as an F13 (12+1) action event by the program, and a Control-F1 will be F25 (12*2+1).

Establish a GUI front-end connection

This section explains runtime to front-end connection in it's simplest form.

Connecting with a front-end

In graphical mode, according to the front-end technology that is used (i.e. desktop client, mobile client, web server client), there are different solutions to establish the connection between the runtime system and the front-end.

This topic describes the development context case, where programs are executed directly with `fglrun`. In a production environment, programs will typically be started with another technology, since the execution of programs will be triggered by the end user interacting with the front-end. Read front-end specific documentation for more details.

From the point of view of the runtime system, the front-end acts as a graphical server and thus the programs must connect to that GUI server in order to display forms and get user input.

The runtime system will try to connect to the front-end only when the first interactive instruction like `MENU` or `INPUT` is reached.

For the runtime system, the front end is identified by the FGLSERVER environment variable. This variable defines the host name of the machine where the front end resides, and the number of the front end instance to be used.

The syntax for FGLSERVER is:

```
{hostname|ip-address}[:servernum]
```

For example:

```
$ FGLSERVER=fox:1
$ fglrun myprog
```

The *servernum* parameter is a whole number that defines the instance of the front-end. It is actually defining a TCP port number the front-end is listening to, starting from 6400. For example, if *servernum* equals 2, the TCP port number used is 6402 (6400+2).

This is the standard/basic connection technique, but you can set up different types of configurations. For example, you can have the front end connect to an application server via ssh, to pass through firewalls over the internet. Refer to the front end documentation for more details.

There can an exception to the standard FGLSERVER specification, if the front-end is denied to listen to a port. If you need to revert the connection principle in this particular case, use the `--gui-listen` option of fglrun. With this option, the runtime system will listen to the specified port, so the front-end can bind to the program and start to use the GUI protocol. The procedure to work in such configuration is the following:

1. Start the program with:

```
fglrun --gui-listen tcp-port prog-name
```

2. Connection from the front-end, for example, with an URL with the following format:

```
fgl://host-name:tcp-port
```

The front end protocol

The *front end protocol* (FEP) is an internal protocol used by the runtime system to synchronize the abstract user interface (AUI) representation on the front end side. This protocol defines a simple set of operations to modify the AUI tree. This protocol is based on a command processing principle (send command, receive answer) and can be serialized to be transported over any network protocol, like HTTP for example.

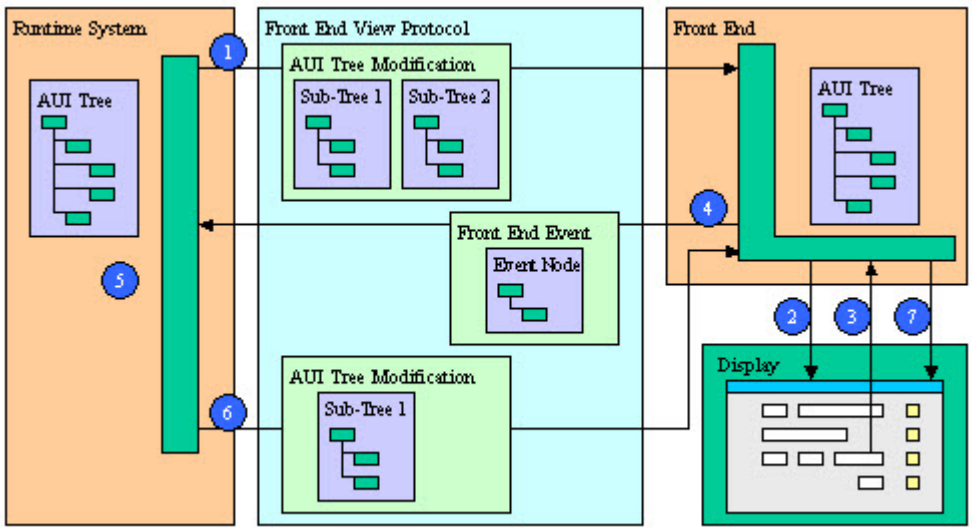


Figure 40: Typical communication between the Runtime System and the Front End

1. Initialization phase: The runtime system sends the initial AUI tree.
2. The front end builds the graphical user interface according to the AUI tree.
3. The front end waits for a user interaction (mouse click, keyboard typing).
4. When the user performs some interaction, the front end sends front end events corresponding to the modifications made by the user.
5. Front end events are analyzed and validated by the runtime system.
6. The runtime system sends back the result of the front end requests, by the way of AUI tree modifications commands.

7. When receiving these commands, the front end modifies its version of the AUI tree and updates the graphical user interface. It then waits for new user interactions (step 3).

Front-end identification

To start a program from the front-end platform, the front-end can open a terminal session on the application server. This is done for example by using a *ssh*, *rlogin*, or *telnet* terminal session. When the terminal session is open, the front end sends a couple of shell commands to set environment variables like FGLSERVER before starting the program to display the application forms on the front end where the terminal session was initiated.

In this configuration, front end identification takes place. The front end identification prevents the display of application forms on a front end that did not start the program on the server. If the front-end was not identified, it would result in an important security problem, as anyone could run a fake program that would display on any front-end and ask for a password.

> **Important: Front end identification is achieved automatically by an initial protocol handshake. However, there can be a security hole if regular operating system users on the application server can overwrite the program or the shell script started by the front end terminal session. Malicious programs can try to display the application on another workstation to read confidential data. As long as basic application users do not have read and write privileges on the program files, there is no risk. To make sure that program files on the server side are protected from basic users, create a special user on the server to manage the application program files, and give other users only read access to those files. As long as basic users cannot modify programs on the server side, there is no security issue.**

Configure the GUI connection timeout

When initiating the connection to the front end, if the front end software is stopped, the host machine is down, or a firewall drops connections for the TCP port used for the GUI connection, the program will stop with an error after a given timeout.

This timeout can be specified with the following FGLPROFILE entry:

```
gui.connection.timeout = seconds
```

The default timeout is 30 seconds.

Wait for front end ping timeout

You can configure the wait-for-ping timeout with the following FGLPROFILE entry:

```
gui.protocol.pingTimeout = 800
```

> **Important:** This feature is not supported when running or mobile devices, or when displaying applications on mobile devices.

It can happen that the user leaves the program for a while without using it. The network policy (firewall) might force a close of the TCP connection after a given period of inactivity. To avoid such connection shutdown when there is no GUI exchange, the front end sends a 'ping' event every N minutes (this is usually configurable in the front-ends) to keep the TCP connection alive. The front end ping is a normal situation and part of the GUI client/server protocol.

> **Important:** With this "keep alive" technique, a front-end connection remains always open, even if the user leaves the workstation for several hours. If your network connection has a cost, you should consider to configure the front-end to turn off the ping event or stop it after a given number of pings. Check the front-end configuration documentation for more details.

If the front end program is not stopped properly (when killed by a system reboot, for example), the TCP connection is lost and the runtime system does not receive any more 'ping' events. In this case, the runtime system waits for a specified time before it stops with fatal error **-8063**.

By default, the runtime system waits for 600 seconds (10 minutes).

> **Important:** If you set the wait-for-ping timeout to a value lower than the ping delay of the front-end, the program will stop with a fatal error after that timeout, even if the TCP connection is still alive. For example, with a front end having a ping delay of 5 minutes, the minimum value for this parameter should be about 330 seconds (5 minutes + 30 seconds to make sure the client ping arrives).

GUI protocol compression

GUI protocol compression might be used to reduce the amount of data exchanged between the front-end and the runtime system. Compression is typically useful on slow networks. The compression algorithm is provided by the standard ZLIB library of the system.

When using the Genero Web Client (GWC/GAS), compression is not useful and is automatically disabled.

Compression makes sense on slow networks (for example, with a phone-line dialup modem, or broadband modem based networks); On fast networks, compression is not required and will in fact use unnecessary processor time.

Compression is disabled by default, and can be enabled with this FGLPROFILE entry:

```
gui.protocol.format = "zlib"
```

If this parameter is defined, but the ZLIB library is not installed on your system or if the ZLIB version is not compatible with the version needed by the runtime system, compression cannot be supported, and the program will stop with error -6317. The ZLIB version must be 1.2.5 (or compatible with version 1.2.5). On Microsoft™ Windows™ platforms, the name of the library must be ZLIB1.DLL; Precompiled binary packages can easily be found on the internet. On UNIX™ platforms, the name of the shared library must be libz.so (normally located in /usr/lib). Note that on Linux™ distributions, you typically have to install the zlib (or zlib1g) package and create a symbolic link for libz.so. The libz.so file is part of zlib-devel package, though.

Front-end errors

When the front end receives an invalid order, it stops the application. The runtime system then stops and displays error -6313 with an additional message, for example:

```
Program stopped at 'myprog.4gl', line number 675.
 FORMS statement error number -6313.
 The User Interface has been destroyed: Unexpected interface version sent
 by the runtime system.
```

Debugging the front-end protocol

When setting the FGLGUIDEBUG environment variable to 1, information about GUI communication will be printed to stderr by the runtime system, and the GUI protocol exchange will be indented for a better readability in the front-end log window.

UNIX™ (shell) example:

```
$ FGLGUIDEBUG=1
$ export FGLGUIDEBUG
$ fglrun myprog 2>guidbg.txt
```

Note that in TUI mode, displayed screens can be dumped by setting the DBSCREENDUMP or DBSCREENOUT environment variables. This can be used to take a snapshot of the current TUI screen, for debugging or testing purpose.

Front-end protocol logging

GUI protocol exchanges can be logged to a file with the `--start-guilog=`*`filename`* option of fglrun, and replayed with the `--run-guilog=`*`filename`* option.

This feature can be used to log .

The options take the log file as parameter:

UNIX™ (shell) example:

```
$ fglrun --start-guilog=mylog.txt myprogram
```

When the program is started, all user interaction and AUI tree updates will be logged to the file specified by the `--start-guilog` option.

The log file can then be replayed with the `--run-guilog` option, to mimic the user interaction, and reproduce potential issues:

```
$ fglrun --run-guilog=mylog.txt
```

Special user interface features

This section describes special features regarding the user interface domain.

The special GUI supported features are:

- Setting key labels on page 759
- Automatic front end startup on page 761
- Text mode screen dump on page 762

Setting key labels

Labels can be defined to decorate buttons controlled by `ON KEY`/`COMMAND KEY` action handlers.

Syntax

Key label configuration can take place at different levels.

- FGLPROFILE definitions

```
key.key-name.text = "label"
```

- Program-level key labels

```
CALL fgl_setkeylabel( "key-name", "label" )
```

- Form level key labels (in KEYS section)

```
KEYS key-name = [%]"label"
[...]
[END]
```

- Dialog level key labels

```
CALL fgl_dialog_setkeylabel( "key-name", "label" )
```

- Form field level key labels (in field definition)

```
KEY key-name = [%]"label"
```

1. *key-name* is the name of the key.
2. *label* is the text to be displayed in the default action view (button).

Usage

When using the graphical mode, `ON KEY` and `COMMAND KEY` action handlers in dialogs can be shown as form buttons when a label text is defined for the key. By defining a label for a key, the runtime system will automatically show a default button for the key action.

> **Important:** Key label configuration is provided for backward compatibility. Consider using action configuration in new programs.

In the next example, the function key `F10` is used to show a detail window in this interactive dialog:

```
INPUT BY NAME myrecord.*
  ON KEY (F10)
    CALL ShowDetail()
END INPUT
```

By default, if you do not specify a label, no default action button is displayed for a function key or control key.

If the text provided for the key label is empty or null, the default action button will not be displayed.

Table 220: Key names recognized by the runtime system

Key Name	Description
`f1` to `f255`	Function keys.
`control-a` to `control-z`	Control keys.
`accept`	Predefined dialog validation action.
`interrupt`	Predefined dialog cancellation action. The action name is *cancel*, not *interrupt*.
`insert`	Predefined `INPUT ARRAY` dialog row insertion action.
`append`	Predefined `INPUT ARRAY` dialog row addition action.
`delete`	Predefined `INPUT ARRAY` dialog row deletion action.
`help`	Predefined help action.

Key labels can be defined at different levels. The order of precedence for key label definition is the following:

1. The label defined with the KEY attribute of the form field.
2. The label defined for the current dialog, using the FGL_DIALOG_SETKEYLABEL function.
3. The label defined in the KEYS section of the form specification file.
4. The label defined as default for a program, using the FGL_SETKEYLABEL function.
5. The label defined in the FGLPROFILE configuration file (`key.`*`key-name`*`.text` entries).

In Genero, you typically define action labels with action attributes. However, if key labels are defined, they will overwrite the text defined in action attributes for the corresponding key action. In BDS 3.xx versions, default key labels are defined in FGLDIR/etc/fglprofile. These defaults have been commented out in Genero to have action attribute text applied (In Genero, by default, fgl_getkeylabel() returns NULL for all keys). If you want to get the same default key labels as in BDS 3.xx, uncomment the key.* lines in FGLDIR/etc/fglprofile.

You can query the label defined at the program level with the FGL_GETKEYLABEL function and, for the current interactive instruction, with the FGL_DIALOG_GETKEYLABEL function.

Automatic front end startup

This section describes how to start a graphical front-end automatically when the runtime system and the front-end reside on the same computer.

When a program starts in graphical mode, the runtime system tries to open a connection to the graphical front end according to the FGLSERVER environment variable. This requires having the front end already started and listening to the TCP port defined according to FGLSERVER.

In some configurations, such as *X11 workstations* or *METAFRAME/Citrix Winframe* or *Microsoft™ Windows™ Terminal Server*, each user may want to start his own front end to have a dedicated process. This can be done by starting the front end automatically when the program executes, according to the DISPLAY (X11) or SESSIONNAME/CLIENTNAME (WTSE) environment variables.

Automatic front end startup settings are defined with `gui.server.autostart.*` entries in FGLPROFILE. In these FGLPROFILE entries, the term "GUI server" refers to the graphical front end.

In a first time, the runtime system tries to establish the connection without starting the front end (in a normal usage, it is already started). The front end is identified by the FGLSERVER environment variable. If FGLSERVER is not defined, it defaults to `localhost:0`, except if `gui.server.autostart.wsmap` entries are defined in FGLPROFILE. When `wsmap` entries are defined, workstation id to GUI server id mapping takes place and FGLSERVER defaults to `localhost:n`, where n is the GUI server number found from `wsmap` entries.

If this first connection fails and the `gui.server.autostart.cmd` entry is defined, the runtime system executes the command to start the GUI server, then waits for n seconds as defined by `gui.server.autostart.wait` entry, and after this delay tries to connect to the front end. If the connection fails, it tries again for a number of attempts defined by the `gui.server.autostart.repeat` entry. Finally, it the last try failed, the runtime system stops with a GUI connection error -6300.

If the `gui.server.autostart.cmd` entry is not defined, neither workstation id to GUI id mapping, nor automatic front-end startup is done.

Here is a detailed description of each `gui.server.autostart` FGLPROFILE entry:

The `cmd` entry is used to define the command to be executed to start the front-end:

```
gui.server.autostart.cmd = "/opt/app/gdc-2.30/bin/gdc -p %d -q -M"
```

Here, `%d` will be replaced by the TCP port the front-end must listen to.

By default the runtime system waits for two seconds before it tries to connect to the front-end. You can change this delay with the **wait** entry:

```
gui.server.autostart.wait = 5   -- wait five seconds
```

The runtime system tries to connect to the front-end ten times. You can change this with the `repeat` entry:

```
gui.server.autostart.repeat = 3 -- repeat three times
```

The following FGLPROFILE entries can be used to define workstation id to front-end id mapping:

```
gui.server.autostart.wsmap.max = 3
gui.server.autostart.wsmap.0.names = "fox:1.0,fox.sxb.4js.com:1.0"
gui.server.autostart.wsmap.1.names = "wolf:1.0,wolf.sxb.4js.com:1.0"
gui.server.autostart.wsmap.2.names = "wolf:2.0,wolf.sxb.4js.com:2.0"
```

The first `wsmap.max` entry defines the maximum number of front-end identifiers to look for. The `wsmap.N.names` entries define a mapping for each GUI server, where **N** is the front-end identifier. The value of those entries defines a comma-separated list of workstation names to match. If no wsmap entries are defined, the GUI server number will default to zero.

For `gui.server.autostart.wsmap` entries, the first GUI server number starts at zero.

On X11 configurations, a workstation is identified by the DISPLAY environment variable. In this example, `fox:1.0` identifies a workstation that will make the runtime start a front end with the number 1.

On Windows™ Terminal Server, the CLIENTNAME environment variable identifies the workstation. If no corresponding front end id can be found in the `wsmap` entries, the front end number defaults to the id of the session defined by the SESSIONNAME environment variable, plus one. The value of this variable has the form *`protocol#id`*; for example, `RDP-Tcp#4` would automatically define a front end id of 5 (4+1).

If the front end processes are started on the same machine as the runtime system, you do not need to set the FGLSERVER environment variable. This will then default to `localhost:`*`id`*, where *id* will be detected according to the `wsmap` workstation mapping entries.

If the front end is executed on a middle-tier machine that is different from the application server, MIDHOST for example, you can set FGLSERVER to MIDHOST without a GUI server id. The workstation mapping will automatically find the id according to `wsmap` settings.

Some front ends such as the Genero Desktop Client (GDC), raise the control panel to the top of the window stack when you try to restart it. In this case the program window might be hidden by the GDC control panel. To avoid this problem, you can use the -M option to start the GDC in minimized mode.

Text mode screen dump

For compatibility with IBM® Informix® 4GL, Genero supports the DBSCREENDUMP and DBSCREENOUT environment variables for debugging purpose, to let you do a screen shot when running in TUI mode and write the result into a file.

To enable TUI screen shot, set either DBSCREENDUMP or DBSCREENOUT to the name of the output file, then run your Genero program with FGLGUI=0 set and press the Ctrl-P key to dump the current screen. Each time you press Ctrl-P the output file will be overwritten.

The DBSCREENDUMP variable writes the screen with escape sequences of TTY attributes, while DBSCREENOUT writes only the characters displayed on the screen, which makes the output more readable.

If both variables are set, the runtime will generate both output files. You should however use different file names, otherwise the output is undefined.

Configuring a text terminal

This section covers topics about text terminal configuration when using the TUI mode (when the FGLGUI environment variable is set to zero).

Terminal type and terminal capabilities definition is not a Genero-specific configuration: TERM, TERMCAP and TERMINFO are also used by other UNIX™ applications and commands.

On UNIX™ platforms, the TERM environment variable must be set to define the terminal type/name. For example, if you execute the application in an xterm X11 window, set TERM=xterm.

On Windows™ platforms, you can run applications in text mode inside a CMD console window. You must not set the TERM environment variable in this case.

Genero supports both *termcap* and *terminfo* implementations of text terminal capabilities. The INFORMIXTERM environment variable defines the type of library used to interact with the terminal. When INFORMIXTERM is set to `termcap` (the default), the runtime system reads terminal capabilities from the file defined by the TERMCAP environment variable. When INFORMIXTERM is set to `terminfo`, the runtime system uses the ncurses library of the operating system to interact with the terminal. We strongly recommend you to use the terminfo solution.

TERMINFO terminal capabilities

When the INFORMIXTERM environment variable is set to `terminfo`, the runtime system will use the ncurses or curses library of the UNIX™ system to display and interact with the terminal device, according to the TERM environment variable.

Make sure that the `libncurses.so` or the `libcurses.so` library is installed on your UNIX™ operating system.

The TERMINFO environment variable can be used to define a different terminal capabilities database as the default. If your UNIX™ system is properly configured, you should not have to set the TERMINFO environment variable.

TERMCAP terminal capabilities

When the INFORMIXTERM environment variable is set to `termcap` or when this variable is undefined, the runtime system will use the termcap terminal capabilities database.

The termcap solution is provided for backward compatibility. You should use terminfo instead, by setting the INFORMIXTERM variable to `terminfo`.

The default termcap database is in the `/etc/termcap` file. If this file is not found, the runtime system will use its default file `$FGLDIR/etc/termcap`. Use the TERMCAP environment variable to specify a different termcap file as the defaults. If you plan to modify the default termcap file, we strongly recommend that you make a copy of the original file and point to the new file with the TERMCAP variable.

In this section we will briefly describe the syntax of the termcap file. For a complete definition please refer to your operating system documentation (see man pages describing the termcap file syntax).

Termcap syntax

All termcap entries contain a list of terminal names, followed by a list of terminal capabilities, in the following format:

- Each capability, including the last one in the entry, is followed by a colon (:).
- ESCAPE is specified as a backslash (\ ) followed by the letter E. CTRL is specified as a caret (^). Do not use the ESCAPE or CTRL keys to indicate escape sequences or control characters in a termcap entry.
- Entries must be defined on a single logical line; a backslash (\ ) appears at the end of each line that wraps to the next line.
- Comment lines begin with a sharp sign (#).

Example: xterm terminal definition:

```
xterm|xterm terminal emulator:\
:km:mi:ms:xn:pt:\
:co#80:li#24:\
:is=\E[r\E[m\E[2J\E[H\E[?7h\E[?1;3;4;6l:\
...
```

Terminal Names

Termcap entries begin with one or more names for the terminal, each separated by a vertical (|) bar. Any one of these names can be used for access to the termcap entry.

Boolean capabilities

Boolean capabilities are two-character codes indicating whether a terminal has a specific feature. If the boolean capability exists in the termcap entry, the terminal has that particular feature.

For example:

```
:bs:am:
# bs backspace with CTRL-H
# am automatic margins
```

Numeric Capabilities

Numeric capabilities are two-character codes followed by a sharp symbol (#) and a value.

For example:

```
:co#80:li#24:
# co number of columns in a line
# li number of lines on the screen
```

The runtime system assumes that the value is zero for any numeric capabilities that are not listed.

String Capabilities

String capabilities specify a sequence that can be used to perform a terminal operation.

A string capability is a two-character code, followed by an equal sign (=) and a string ending at the next delimiter (:).

Most termcap entries include string capabilities for clearing the screen, arrow keys, cursor movement, underscore, function keys, etc.

For example, this shows some string capabilities for a Wyse 50 terminal:

```
:ce=\Et:cl=\E*:\
:nd=^L:up=^K:\
:so=\EG4:se=\EG0:
# ce=\Et clear to end of line
# cl=\E* clear the screen
# nd=^L non-destructive cursor right
# up=^K up one line
# so=\EG4 start stand-out
# se=\EG0 end stand-out
```

Genero-specific termcap definitions

Extending Function Key Definitions

In TUI mode, the runtime system recognizes function keys F1 through F36. These keys correspond to the termcap capabilities k0 through k9, followed by kA through kZ.

The termcap entry for these capabilities is the sequence of ASCII characters your terminal sends when you press the function keys (or any other keys you choose to use as function keys).

This example shows some function key definitions for the xterm terminal:

```
k0=\E[11~:k1=\E[12~:k2=\E[13~:k3=\E[14~:\
...
k9=\E[21~:kA=\E[23~:kB=\E[24~:\
```

Defining dialog action keys

Dialog action keys for insert, delete and list navigation can be defined with the following capabilities:

- **ki** : Insert line (default is CTRL-J)
- **kj** : Delete line (default is CTRL-K)
- **kf** : Next page (default is CTRL-M)
- **kg** : Previous page (default is CTRL-N)

  Note: You can also use the OPTIONS statement to name other function keys or CTRL keys for these operations.

Specifying Characters for Window Borders

The runtime system uses the graphics characters in the termcap file when you specify a window border in an `OPEN WINDOW` statement.

The runtime system uses characters defined in the termcap file to draw the border of a window. If no characters are defined in this file, the runtime system uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners.

Steps to define the graphical characters for window borders for your terminal type:

1. Determine the escape sequences for turning the terminal graphics mode ON and OFF (Refer to the manual of your terminal). For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B, and the escape sequence for leaving graphics mode is ESC H^C.
2. Identify the ASCII equivalents for the six graphics characters that Genero requires to draw the window borders. The ASCII equivalent of a graphics character is the key you would press in graphics mode to obtain the indicated character. The six graphical characters needed by Genero are:
   a. The upper left corner
   b. The lower left corner
   c. The upper right corner
   d. The lower right corner
   e. The horizontal line
   f. The vertical line
3. Edit the termcap entry for your terminal, and define the following string capabilities:
   - **gs** : The escape sequence for entering graphics mode. In the termcap file, ESCAPE is represented as a backslash (\ ) followed by the letter E; CTRL is represented as a caret (^). For example, the Wyse 50 escape sequence ESC-H CTRL-B is represented as \EH^B.
   - **ge** : The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESC-H CTRL-C is represented as \EH^C.
   - **gb** : The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border. Using the order as listed in (2).

     For example, if you are using a Wyse 50 terminal, you would add the following, in a linear sequence:

     ```
     :gs=\EH^B:ge=\EH^C:gb=2135z6:\
     ```

For terminals without graphics capabilities, you must enter a blank value for the **gs** and **ge** capabilities. For **gb**, enter the characters you want Genero to use for the window border. The following example shows possible values for **gs**, **ge**, and **gb** in an entry for a terminal without graphics capabilities:

```
:gs=:ge=:gb=.|.|_|:
```

With these settings, window borders would be drawn using underscores (_) for horizontal lines, vertical bars (|) for vertical lines, periods (.) for the top corners, and vertical bars (|) for the lower corners.

Adding Color and Intensity

In TUI mode, a Genero program can be written either for a monochrome or a color terminal, and then you can run the program on either type of terminal. If you set up the termcap files as described, the color attributes and the intensity attributes are related.

Table 221: Relationship between color attributes and intensity attributes

Number	Color	Intensity	Note
0	WHITE	NORMAL	
1	YELLOW	BOLD	

Number	Color	Intensity	Note
2	MAGENTA	BOLD	
3	RED	BOLD (*)	If the keyword BOLD is indicated as the attribute, the field will be RED on a color terminal
4	CYAN	DIM	
5	GREEN	DIM	
6	BLUE	DIM (*)	If the keyword DIM is indicated as the attribute, the field will be BLUE on a color terminal
7	BLACK	INVISIBLE	

The background for colors is BLACK in all cases. In either color or monochrome mode, you can add the REVERSE, BLINK, or UNDERLINE attributes if your terminal supports them.

The ZA String Capability

Genero uses a parameterized string capability named **ZA** in the termcap file to determine color assignments. Unlike other termcap string capabilities that you set to a literal sequence of ASCII characters, ZA is a function string that depends on the following four parameters:

Table 222: ZA function parameters

Parameter	Name	Description
1	p1	Color number between 0 and 7 (see Table 221: Relationship between color attributes and intensity attributes on page 765).
2	p2	0 = Normal; 1 = Reverse.
3	p3	0 = No-Blink; 1 = Blink.
4	p3	0 = No-underscore; 1 = Underscore.

ZA uses the values of these four parameters and a stack machine to determine which characters to send to the terminal. The ZA function is called, and these parameters are evaluated, when a color or intensity attribute is encountered in a Genero program. Use the information in your terminal manual to set the ZA parameters to the correct values for your terminal.

The ZA string uses stack operations to push values onto the stack or to pop values off the stack. Typically, the instructions in the ZA string push a parameter onto the stack, compare it to one or more constants, and then send an appropriate sequence of characters to the terminal. More complex operations are often necessary; by storing the display attributes in static stack machine registers (named a through z), you can have terminal-specific optimizations.

The different stack operators that you can use to write the descriptions are summarized here. For a complete discussion of stack operators, see your operating system documentation.

Operators That Send Characters to the Terminal

- **%d** pops a numeric value from the stack and sends a maximum of three digits to the terminal. For example, if the value 145 is at the top of the stack, %d pops the value off the stack and sends the ASCII

representations of 1, 4, and 5 to the terminal. If the value 2005 is at the top of the stack, %d pops the value off the stack and sends the ASCII representation of 5 to the terminal.

- **%2d** pops a numeric value from the stack and sends a maximum of two digits to the terminal, padding to two places. For example, if the value 145 is at the top of the stack, %2d pops the value off the stack and sends the ASCII representations of 4 and 5 to the terminal. If the value 5 is at the top of the stack, %2d pops the value off the stack and sends the ASCII representations of 0 and 5 to the terminal.
- **%3d** pops a numeric value from the stack and sends a maximum of three digits to the terminal, padding to three places. For example, if the value 7 is at the top of the stack, %3d pops the value off the stack and sends the ASCII representations of 0, 0, and 7 to the terminal.
- **%c** pops a single character from the stack and sends it to the terminal.

Operators That Manipulate the Stack

- **%p[1-9]** pushes the value of the specified parameter on the stack. The notation for parameters is p1, p2, ... p9. For example, if the value of p1 is 3, %p1 pushes 3 on the stack.
- **%P[a-z]** pops a value from the stack and stores it in the specified variable. The notation for variables is Pa, Pb, ... Pz. For example, if the value 45 is on the top of the stack, %Pb pops 45 from the stack and stores it in the variable Pb.
- **%g[a-z]** gets the value stored in the corresponding variable (P[a-z]) and pushes it on the stack. For example, if the value 45 is stored in the variable Pb, %gb gets 45 from Pb and pushes it on the stack.
- **%'c'** pushes a single character on the stack. For example, %'k' pushes k on the stack.
- **%{n}** pushes an integer constant on the stack. The integer can be any length and can be either positive or negative. For example, %{0} pushes the value 0 on the stack.
- **%S[a-z]** pops a value from the stack and stores it in the specified static variable. (Static storage is nonvolatile since the stored value remains from one attribute evaluation to the next.) The notation for static variables is Sa, Sb, ... Sz. For example, if the value 45 is on the top of the stack, %Sb pops 45 from the stack and stores it in the static variable Sb. This value is accessible for the duration of the Genero program.
- **%G[a-z]** gets the value stored in the corresponding static variable (S[a-z]) and pushes it on the stack. For example, if the value 45 is stored in the variable Sb, %Gb gets 45 from Sb and pushes it on the stack.

Arithmetic Operators

Each arithmetic operator pops the top two values from the stack, performs an operation, and pushes the result on the stack.

- **%+** Addition.

  For example, %{2}%{3}%+ is equivalent to 2+3.
- **%-** Subtraction.

  For example, %{7}%{3}%- is equivalent to 7-3.
- **%*** Multiplication.

  For example, %{6}%{3}%* is equivalent to 6*3.
- **%/** Integer division.

  For example, %{7}%{3}%/ is equivalent to 7/3 and produces a result of 2.
- **%m** Modulus (or remainder).

  For example, %{7}%{3}%m is equivalent to (7 mod 3) and produces a result of 1.

Bit Operators

The following bit operators pop the top two values from the stack, perform an operation, and push the result on the stack:

- **%&** Bit-and.

For example, %{12}%{21}%& is equivalent to (12 and 21) and produces a result of 4.

- **%|** Bit-or.

  For example, %{12}%{21}%| is equivalent to (12 or 21) and produces a result of 29.
- **%^** Exclusive-or.

  For example, %{12}%{21}%^ is equivalent to (12 exclusive-or 21) and produces a result of 25.

The following unary operator pops the top value from the stack, performs an operation, and pushes the result on the stack:

- **%~** Bitwise complement.

  For example, %{25}%~ results in a value of -26.

Logical Operators

The following logical operators pop the top two values from the stack, perform an operation, and push the logical result (0 for false or 1 for true) on the stack:

- **%=** Equal to.

  For example, if the parameter p1 has the value 3, the expression %p1%{2}%= is equivalent to 3=2 and produces a result of 0 (false).
- **%>** Greater than.

  For example, if the parameter p1 has the value 3, the expression %p1%{0}%> is equivalent to 3>0 and produces a result of 1 (true).
- **%<** Less than.

  For example, if the parameter p1 has the value 3, the expression %p1%{4}%< is equivalent to 3<4 and produces a result of 1 (true).

The following unary operator pops the top value from the stack, performs an operation, and pushes the logical result (0 or 1) on the stack.

- **%!** Logical negation.

  This operator produces a value of zero for all nonzero numbers and a value of 1 for zero. For example, %{2}%! results in a value of 0, and %{0}%! results in a value of 1.

Conditional Statements

The conditional statement has the following format:

```
%? expr %t thenpart %e elsepart %;
```

The *%e elsepart* is optional. You can nest conditional statements in the *thenpart* or the *elsepart* .

When Genero evaluates a conditional statement, it pops the top value from the stack and evaluates it as either true or false. If the value is true, the runtime performs the operations after the %t; otherwise it performs the operations after the %e (if any).

For example, the expression:

```
%?%p1%{3}%=%t;31%;
```

is equivalent to:

```
if p1 = 3 then print ";31"
```

Assuming that p1 in the example has the value 3, Genero would perform the following steps:

- **%?** does not perform an operation but is included to make the conditional statement easier to read.
- **%p1** pushes the value of p1 on the stack.
- **%{3}** pushes the value 3 on the stack.
- **%=** pops the value of p1 and the value 3 from the stack, evaluates the boolean expression p1=3, and pushes the resulting value 1 (true) on the stack.
- **%t** pops the value from the stack, evaluates 1 as true, and executes the operations after %t. (Since ";31" is not a stack machine operation, Genero prints ";31" to the terminal.)
- **%;** terminates the conditional statement.

ZA example

The ZA sequence for the ID Systems Corporation ID231 (color terminal) is:

```
ZA =
\E[0;                  # Print lead-in
%?%p1%{0}%=%t%{7}      # Encode color number (translate color number
                       # to number for the ID231 term)
%e%p1%{1}%=%t%{3}      #
%e%p1%{2}%=%t%{5}      #
%e%p1%{3}%=%t%{1}      #
%e%p1%{4}%=%t%{6}      #
%e%p1%{5}%=%t%{2}      #
%e%p1%{6}%=%t%{4}      #
%e%p1%{7}%=%t%{0}%;    #
%?%p2%t30;%{40}%+%2d # if p2 is set, print '30' and '40' + color number
 (reverse)
%e40;%{30}%+%2d%;      #  else print '40' and '30' + color number (normal)
%?%p3%t;5%;            # if p3 is set, print 5 (blink)
%?%p4%t;4%;            # if p4 is set, print 4 (underline)
m                      # print 'm' to end character sequence
```

Form definitions

This section describes how to define application forms and program resources related to the presentation layer.

- Windows and forms on page 769
- Using images on page 782
- Accessibility guidelines on page 791
- Message files on page 794
- Action defaults files on page 796
- Presentation styles on page 799
- Form specification files on page 853
- Form rendering on page 1002
- Toolbars on page 1021
- Topmenus on page 1027

Windows and forms

The section describes the concept of windows and forms in the language.

- Understanding windows and forms on page 770
- OPEN WINDOW on page 772
  - Window position and size on page 773
  - OPEN WINDOW attributes on page 774
  - The WITH FORM clause on page 775

  - Window styles on page 775
  - Window titles on page 776
  - Window icons on page 776
  - Window types on page 777
- CLOSE WINDOW on page 777
- CURRENT WINDOW on page 778
- OPEN FORM on page 779
- DISPLAY FORM on page 780
- CLOSE FORM on page 781
- CLEAR WINDOW on page 779
- CLEAR SCREEN on page 781
- DISPLAY AT on page 781

Understanding windows and forms

Programs manipulate windows and forms, to define display and input areas controlled by interactive instructions such as the `INPUT` dialog. When a dialog is started, it uses the form associated with the current window. Forms are defined in .42f compiled form files and are loaded and displayed in windows.

Window objects

The windows are created from programs; they define a display context for sub-elements like forms, menus, message and error lines. A window can contain only one form at a time, but you can display different forms successively in the same window.

When using a the text mode (FGLGUI=0), windows are displayed in the character terminal as fixed-size boxes, at a given line/column position, width and height. When using a graphical desktop front end (FGLGUI=1), windows are displayed as independent re-sizeable windows by default. Note that a GUI application can run in traditional mode (`gui.uiMode="traditional"` FGLPROFILE setting), displaying windows as simple static areas inside a real graphical parent window. When using a mobile device front-end, only one window is visible at the time, because of device platform GUI standards and the limited screen sizes (smartphones). Split views is the exception, and allows to display two windows side by side for a typical list-detail display on tablets.

A program creates a new window with the `OPEN WINDOW` instruction, which also defines the window identifier. A window is destroyed with the `CLOSE WINDOW` instruction:

```
OPEN WINDOW mywindow WITH FORM "myform"
...
CLOSE WINDOW mywindow
```

If there is a current window, it is possible to display several forms successively in that same window. The previous form is removed automatically by the runtime system when displaying a new form to the window:

```
OPEN WINDOW mywindow WITH FORM "form1"
INPUT BY NAME ... -- uses form1 elements
...
OPEN FORM f1 FROM "form2"
DISPLAY FORM f1  -- removes "form1" from the window
INPUT BY NAME ... -- uses form2 elements
...
```

When a program starts, the runtime system creates a default window named `SCREEN`. This default window can be used as a regular window: it can hold a menu and a form. If needed, it can be closed with `CLOSE WINDOW SCREEN`. You typically display the main form of your program in the `SCREEN` window, by using `OPEN FORM` + `DISPLAY FORM`:

```
MAIN
```

```
    -- The SCREEN window exists by default
    ...
    OPEN FORM f_main FROM "customers"
    DISPLAY FORM f_main -- displays in SCREEN
    ...
END MAIN
```

Several windows can be created, but there can be only one current window when using modal dialogs (only one dialog is active at the time, thus only the current window can be active). By using parallel dialogs, several windows can be active concurrently. Parallel dialogs were introduced to implement split views, for mobile devices.

There is always a current window. The last created window becomes the current window. When the last created window is closed, the previous window in the window stack becomes the current window. Use the `CURRENT WINDOW` instruction to make a specific window current before executing the corresponding dialog that is controling the window content:

```
OPEN WINDOW w_customers ...
OPEN WINDOW w_orders ...
...
CURRRENT WINDOW IS w_customers
...
CLOSE WINDOW w_customers
CURRRENT WINDOW IS w_orders
...
```

By default, a window has no particular type and displays as a modal window on the front-end, to be controlled by a modal dialog instruction. In some situations, you must specify the type of the window to get a specific rendering and behavior. This is achieved by defining the `TYPE` attribute in the `ATTRIBUTES` clause of the `OPEN WINDOW` instruction:

```
OPEN WINDOW w_cust WITH FORM "f_cust" ATTRIBUTES(TYPE=LEFT)
...
OPEN WINDOW w_pref WITH FORM "f_pref" ATTRIBUTES(TYPE=POPUP)
...
```

Specify decoration options with a presentation style for the window, identified the `STYLE` attribute of the `ATTRIBUTES` section of `OPEN WINDOW`. Window styles can also be specified at form level, with the `WINDOWSTYLE` form attribute in the `LAYOUT` of the form definition:

```
OPEN WINDOW w_cust WITH FORM "f_cust" ATTRIBUTES(STYLE="dialog2")
```

The `ui.Window` built-in class can be used to manipulate windows as objects. The common practice is to get the current form of the window and use it as `ui.Form` object to manipulate its content.

The windows can be displayed in an WCI container application, by using the `ui.Interface` methods to define parent / child relationship.

Form objects

Forms define the layout and presentation of areas used by the dialogs (`INPUT`), to display or input data. Forms are loaded by programs from external files with the .42f extension, the compiled version of .per form specification files.

Forms can be stamped with the `VERSION` attribute. The form version attribute is used to indicate that the form content has changed. The front end is then able to distinguish different form versions and avoid saved settings being applied for new form versions.

Forms can be loaded with the `OPEN FORM` instruction followed by a `DISPLAY FORM`, to display the form into the current window, or forms can be used directly as window creation argument with the `OPEN WINDOW ... WITH FORM` instruction:

```
OPEN FORM f_cust FROM "f_cust"
DISPLAY FORM f_cust -- into current window
...
OPEN WINDOW w_cust WITH FORM "f_cust"
```

The form that is used by interactive instructions like `INPUT` is defined by the current window containing the form. Switching between existing windows (and thus, between forms associated to the windows) is done with the `CURRENT WINDOW` instruction.

Several forms can be successively displayed in the same (current) window. The last displayed form will be used by the next dialog, while the form displayed before will be automatically removed from the window:

```
OPEN WINDOW w_common WITH 20 ROW, 60 COLUMNS
...
OPEN FORM f1 FROM "f_cust"
DISPLAY FORM f1 -- f_cust is shown
INPUT BY NAME rec_cust.* ...
...
OPEN FORM f2 FROM "f_ord"
DISPLAY FORM f2 -- f_ord is shown (f_cust is removed)
INPUT BY NAME rec_ord.* ...
```

The `ui.Form` built-in class is provided to handle form elements. You can, for example, hide some parts of a form with the `setElementHidden()` method. Get a `ui.Form` object with the `ui.Window.getForm()` method.

OPEN WINDOW
Creates and displays a new window.

Syntax

```
OPEN WINDOW identifier
   [ AT line, column ]
   WITH { FORM form-file
        | height ROWS, width COLUMNS
        }
   [ ATTRIBUTES ( window-attributes ) ]
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
| BORDER
| TEXT = "string"
| TYPE = { RIGHT | LEFT | POPUP | NAVIGATOR }
| STYLE = "string"
| PROMPT LINE = integer
| MENU LINE = integer
| MESSAGE LINE = integer
| ERROR LINE = integer
| COMMENT LINE = { OFF | integer }
}
```

1. *identifier* is the name of the window. It is always converted to lowercase by the compiler.

2. *line* is the integer defining the top position of the window. The first line in the screen is 1, while the relative line number inside the window is zero.
3. *column* is the integer defining the position of the left margin. The first column in the screen is 1, while the relative column number inside the window is zero.
4. *form-file* defines the .42f compiled form specification file to be used, without the file extension.
5. *height* defines the number of lines of the window in character units; includes the borders in character mode.
6. *width* defines the number of lines of the window in character units; includes the borders in character mode.

Usage

An `OPEN WINDOW` statement can have the following effects:

- Declares a name (the *identifier*) for the window.
- Indicates which form has to be used in that window.
- Specifies the display attributes of the window.
- When using character mode, specifies the position and dimensions of the window, in character units.

For graphical applications, use this instruction without the `AT` clause, and with the `WITH FORM`clause.

The window identifier must follow the rules for identifiers and be unique among all windows defined in the program. Its scope is the entire program. You can use this identifier to reference the same Window in other modules with other statements (for example, `CURRENT WINDOW` and `CLOSE WINDOW`).

The compiler converts the window identifier to lowercase for internal storage. When using functions or methods receiving the window identifier as a string parameter, the window name is case-sensitive. We recommend that you always specify the window identifier in lowercase letters.

The runtime system maintains a stack of all open windows. If you execute `OPEN WINDOW` to open a new window, it is added to the window stack and becomes the current window. Other statements that can modify the window stack are `CURRENT WINDOW` and `CLOSE WINDOW`.

Example

```
MAIN
  OPEN WINDOW w1 WITH FORM "customer"
  MENU "Test"
    COMMAND KEY(INTERRUPT) "exit" EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

Window position and size

Window objects can be created with a position and size for the TUI mode.

When using the full GUI mode (without the traditional mode), the `AT` *`line, column`* clause is optional and if used, the `WITH` *`lines`* `ROWS,` *`characters`* `COLUMNS` clause is ignored, because the size of the window is automatically calculated according to its contents.

When using the TUI mode, the `AT` *`line, column`* clause defines the position of the top-left corner of the window on the terminal screen and `WITH` *`lines`* `ROWS,`*`characters`* `COLUMNS` clause specifies explicit vertical and horizontal dimensions for the window. The expression at the left of the `ROWS` keyword specifies the height of the window, in character unit lines. This must be an integer between 1 and max, where max is the maximum number of lines that the screen can display. The integer expression after the comma at the left of the `COLUMNS` keyword specifies the width of the window, in character unit columns. This must return a whole number between 1 and length, where length is the number of characters that your monitor can display on one line. In addition to the lines needed for a form, allow room for the `COMMENT` line, the `MENU` line, the `MENU` comment line and the `ERROR` line. The runtime system issues a runtime error if the window

does not include sufficient lines to display both the form and these additional reserved lines. The minimum number of lines required to display a form in a window is the number of lines in the form, plus an additional line below the form for prompts, messages, and comments.

OPEN WINDOW attributes
List if attributes for the OPEN WINDOW instruction.

Table 223: Window-attributes supported by the OPEN WINDOW statement

Attribute	Description
`TEXT = string`	Defines the default title of the window. When a form is displayed, the form title (`LAYOUT(TEXT="mytitle")`) will be used as window title. **Tip:** We recommend that you define the window title in the form file.
`STYLE = string`	Defines the default style of the window. If the form defines a window style, (`LAYOUT(WINDOWSTYLE="mystyle")`), it overwrites the default window style. **Tip:** We recommend that you define the window style in the form file.
`TYPE = [LEFT|RIGHT|POPUP|NAVIGATOR]`	Defines the window type. According to the type specified, the window will appear differently, following front-end platform GUI standards. For example, on iOS devices, a window created with `TYPE=POPUP` will show up from the bottom of the screen. See Window types on page 777.
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	Default TTY color of the data displayed in the window.
`BOLD, DIM, INVISIBLE, NORMAL`	Default TTY font attribute of the data displayed in the window.
`REVERSE, BLINK, UNDERLINE`	Default TTY video attribute of the data displayed in the window.
`PROMPT LINE integer`	In character mode, indicates the position of the prompt line for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`FORM LINE integer`	In character mode, indicates the position of the form line for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`MENU LINE integer`	In character mode, indicates the position of the ring menu line for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.

Attribute	Description
`MESSAGE LINE integer`	In character mode, indicates the position of the message line for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`ERROR LINE integer`	In character mode, indicates the position of the error line for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`COMMENT LINE {OFF|integer}`	In character mode, indicates the position of the comment line or no comment line at all, for this window. The position can be specified with `FIRST` and `LAST` predefined line positions.
`BORDER`	Indicates if the window must be created with a border in character mode. A border frame is drawn outside the specified window. This means, that the window needs 2 additional lines and columns on the screen.

The following list describes the default line positions in character mode:

- First line: Prompt line (output from `PROMPT` statement) and Menu line (command value from `MENU` statement).
- Second line: Message line (output from `MESSAGE` statement; also the descriptions of `MENU` options).
- Third line: Form line (output from `DISPLAY FORM` statement).
- Last line: Error line (output from `ERROR` statement). Also comment line in any window except `SCREEN`.

The WITH FORM clause
Creating a window object with a form.

As an alternative to specifying explicit dimensions for a window, the `WITH FORM` clause can specify the name of a compiled form file, without the `.42f` file extension. A window object is automatically opened and sized to the screen layout of the form. When using the TUI mode, the width of the window is from the left-most character on the screen form (including leading blank spaces) to the right-most character on the screen form (truncating trailing blank spaces). The length of the window is calculated as (form line) + (form length).

```
OPEN WINDOW w1 WITH FORM "custlist"
```

It is recommended that you use the `WITH FORM` clause, especially in the default GUI mode, because the window is created in accordance with the form. If you use this clause, you do not need the `OPEN FORM`, `DISPLAY FORM`, or `CLOSE FORM` statement to open and close the form. The `CLOSE WINDOW` statement closes the window and the form.

Window styles
Use the `STYLE` attribute to set a style for a window.

By default, windows are displayed as normal application windows, but you can use the *window style* to show a window at the top of all other windows, as a "modal window".

The window style defines the type of the window (normal, modal) and its decoration, via a presentation style. The presentation style specifies a set of attributes in an external file (.4st).

There are different ways to define the style of a window: The `STYLE` attribute can be used in the `OPEN WINDOW` instruction to define the default style for a window, but it is better to specify the window style in the

form file, with the WINDOWSTYLE attribute of the LAYOUT section. This avoids decoration-specific code in the programs.

Table 224: Standard window styles defined in the default presentation style file

Style name in 4st file	Description
Window	Defines presentation attributes for common application windows. When using MDI containers, normal windows are displayed as MDI children.
Window.main, Window.main2	Defines presentation attributes for starter applications, where the main window shows a startmenu if one is defined by the application.
Window.dialog, Window.dialog2, Window.dialog3, Window.dialog4	Defines presentation attributes for typical OK/Cancel modal windows. On iOS mobile devices, opening a new window with the predefined style 'dialog' causes the window to slide up from the bottom: `OPEN WINDOW w_opt WITH FORM "f_opt" ATTRIBUTES(STYLE="dialog")`
Window.naked	Defines presentation attributes for windows that should not show the default view for ring menus and action buttons (OK/Cancel).
Window.viewer	Defines presentation attributes for viewers as the report pager (fglreport.per).

It is recommended that you not change the default settings of windows styles in the FGLDIR/lib/default.4st file. If you create your own style file, copy the default styles into your own file in a different directory.

It is not possible to change the presentation style attributes of windows dynamically in the AUI tree. The style is applied when the window and form are loaded.

If you open and display a second form in an existing window, the window style of the second form is not applied.

Window titles
Use the TEXT attribute to define a title for a window.

The TEXT attribute in the ATTRIBUTE clause of OPEN WINDOW defines the default title of the window. If the window is opened with a form (WITH FORM clause) that defines a TEXT attribute in the LAYOUT section, the default is ignored. Subsequent OPEN FORM / DISPLAY FORM instructions may change the window title if the new form defines a different title in the LAYOUT section.

It is recommended that you specify the window title in the form file, instead of using the TEXT attribute of the OPEN WINDOW instruction.

If you want to set a window title dynamically, you can use the setText() method of the ui.Window built-in class.

Window icons
Use a IMAGE attribute to define the icon for a window.

If the window is opened with OPEN WINDOW WITH FORM, by using a form file that defines an IMAGE attribute in the LAYOUT section, the window will use this image as icon. Subsequent OPEN FORM /

`DISPLAY FORM` instructions may change the window icon if the new form defines a different image in the `LAYOUT` section.

If you want to set a window icon dynamically, you can use the `setImage()` method of the `ui.Window` built-in class.

Window types
Use the `TYPE` attribute to define the type of a window.

> **Important:** This feature is only for mobile platforms.

The type of a window can be specified with the `TYPE` attribute in the `OPEN WINDOW` instruction:

```
OPEN WINDOW w_main WITH FORM "navi"
     ATTRIBUTES( TYPE = NAVIGATOR )
```

This attribute was introduced to implement split-views on mobile front-ends.

Possible values for the `TYPE` attribute are described in the following table:

Table 225: Supported window types

Name	Description
`LEFT`	Defines the window as the left pane when implementing split views. The window will be the parent window of a window cascade displayed on the left-hand side.
`NAVIGATOR`	Defines the window as the action pane (i.e. iOS Tab bar) when implementing split views. This type of window will be used as top-level navigator window, showing the options to switch between different windows controlled by parallel dialogs.
`POPUP`	Defines the window as popup (modal) window, to open on the top of other windows.
`RIGHT`	Defines the window as the right pane when implementing split views. The window will be the parent window of a window cascade displayed on the right-hand side.

CLOSE WINDOW
Closes and destroys a window.

Syntax

```
CLOSE WINDOW { identifier | SCREEN }
```

1. *identifier* is the name of the window.

Usage

If the `OPEN WINDOW` statement includes the `WITH FORM` clause, it closes both the form and the window.

Closing a window has no effect on variables that were set while the window was open.

Closing the current window makes the next window on the stack the new current window. If you close any other window, the runtime system deletes it from the stack, leaving the current window unchanged.

If the window is currently being used for input, `CLOSE WINDOW` generates a runtime error.

You can close the default screen window with the `CLOSE WINDOW SCREEN` instruction.

Example

```
MAIN
  OPEN WINDOW w1 WITH FORM "customer"
  MENU "Test"
    COMMAND KEY(INTERRUPT) "exit" EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

CURRENT WINDOW
Makes a specified window the current window.

Syntax

```
CURRENT WINDOW IS { identifier | SCREEN }
```

1. *identifier* is the name of the window.

Usage

Programs with multiple windows might need to switch to a different open window so that input and output occur in the appropriate window. To make a window the current window, use the `CURRENT WINDOW` statement.

When a program starts, the screen is the current window. Its name is SCREEN. To make this the current window, specify the keyword `SCREEN` instead of a window identifier.

If the window contains a form, that form becomes the current form when a `CURRENT WINDOW` statement specifies the name of that window. All interactive instruction such as `CONSTRUCT`, `INPUT` use only the current window for input and output. If the user displays another form (for example, through an `ON KEY` clause) in one of these statements, the window containing the new form becomes the current window. When an interactive instruction resumes, its original window becomes the current window.

The `CURRENT WINDOW` instruction is typically used in TUI based applications, when distinct areas of the screen are reserved for different usage. In a GUI application, windows are rather opened and closed sequentially or on a stack of windows.

Example

```
MAIN
  OPEN WINDOW w1 WITH FORM "customer"
  ...
  OPEN WINDOW w2 WITH FORM "custlist"
  ...
  CURRENT WINDOW IS w1
  ...
  CURRENT WINDOW IS w2
  ...
  CLOSE WINDOW w1
  CLOSE WINDOW w2
END MAIN
```

CLEAR WINDOW

Clears the contents of a window.

Syntax

```
CLEAR WINDOW { identifier | SCREEN }
```

1. *identifier* is the name of the window.

Usage

The `CLEAR WINDOW` instruction clears the content of the specified window that was declared in an `OPEN WINDOW`. If the window was created with borders, these are left untouched (only the content of the window is cleared).

If you specify `CLEAR WINDOW SCREEN`, the root screen will be cleared, except areas occupied by an existing window. `CLEAR WINDOW SCREEN` will not change the current window setting.

The `CLEAR WINDOW` instruction is typically used in TUI based applications, as it clears the whole content of the window, including static labels and messages.

OPEN FORM

Declares a compiled form in the program.

Syntax

```
OPEN FORM identifier FROM filename
```

1. *identifier* is an identifier that defines the name of the form object.
2. *filename* is a string expression defining the name of the compiled form file, without .42f extension.

Usage

In order to use a .42f compiled version of a form specification file, the programs must declare the form with the `OPEN FORM` instruction and then display the form in the current window by using the `DISPLAY FORM` instruction.

`OPEN FORM` / `DISPLAY FORM` are typically used at the beginning of programs to display the main form in the default `SCREEN` window:

```
OPEN FORM custform FROM "customer"
DISPLAY FORM custform
```

The form identifier does not need to match the name of the form specification files, but it must be unique among form names in the program. Its scope of reference is the entire program.

The quoted string that follows the `FROM` keyword must specify the name of the file that contains the compiled screen form. This filename can include a pathname, but this is not recommended.

Form files are found by using the directory paths defined in the DBPATH or FGLRESOURCEPATH environment variable. It is not recommended that you provide a path for *filename*; Instead, use simple file names in programs and put the compiled forms in a directory defines in DBPATH / FGLRESOURCEPATH environment variable.

If you execute an `OPEN FORM` with the name of an open form, the runtime system first closes the existing form before opening the new form.

The scope of reference of form identifier is the entire program.

When the window is dedicated to the form, use the `OPEN WINDOW WITH FORM` instruction to create the window and the form object in one statement.

In TUI mode, the form is displayed in the current window at the position defined by the FORM LINE attribute that can be specified in the ATTRIBUTE clause of OPEN WINDOW or as default with the OPTIONS instruction.

After the form is loaded, you can activate the form by executing a CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or DIALOG statement. When the runtime system executes the OPEN FORM instruction, it allocates resources and loads the form into memory. To release the allocated resources when the form is no longer needed, the program must execute the CLOSE FORM instruction. This is a memory-management feature to recover memory from forms that the program no longer displays on the screen. If the form was loaded with a window by using the WITH FORM clause, it is automatically closed when the window is closed with a CLOSE WINDOW instruction.

Example

```
MAIN
  OPEN FORM f1 FROM "customer"
  DISPLAY FORM f1
  CALL input_customer()
  CLOSE FORM f1
  OPEN FORM f2 FROM "custlist"
  DISPLAY FORM f2
  CALL input_custlist()
  CLOSE FORM f2
END MAIN
```

DISPLAY FORM
Displays and associates a form with the current window.

Syntax

```
DISPLAY FORM identifier
[ ATTRIBUTES ( display-attributes ) ]
```

1. *identifier* is the name of the form.
2. *window-attributes* defines the display attributes of the form.

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

Usage

The DISPLAY FORM instruction creates a form element in the current window, from a form resource loaded by the OPEN FORM instruction.

> **Important:** The INVISIBLE display attribute is ignored.

The runtime system applies display attributes that you specify in the ATTRIBUTES clause to any fields that have not been assigned attributes by the ATTRIBUTES section of the form specification file, or by the database schema files, or by the OPTIONS runtime configuration statement. If the form is displayed in a window, color attributes from the DISPLAY FORM statement supersede any from the OPEN WINDOW statement. If subsequent CONSTRUCT, DISPLAY, or DISPLAY ARRAY statements that include an

ATTRIBUTES clause reference the form, however, their attributes take precedence over those specified in the DISPLAY FORM instruction.

CLOSE FORM
Closes the resources allocated by OPEN FORM.

Syntax

```
CLOSE FORM identifier
```

1. *identifier* is the name of the form.

Usage

The CLOSE FORM instruction releases the memory allocated to the form.

A form associated with a window by the OPEN WINDOW WITH FORM instruction is automatically closed when the program closes the window with a CLOSE WINDOW instruction.

CLEAR SCREEN
Clears the complete application screen.

Syntax

```
CLEAR SCREEN
```

Usage

The CLEAR SCREEN instruction is typically used in TUI mode to clear the complete screen and make the root screen window the current window on the stack.

The whole screen will be cleared, including prompt, error and message lines (the menu line is not cleared).

DISPLAY AT
Displays text at a given line/column position in the current window.

Syntax

```
DISPLAY text AT line, column [ ATTRIBUTES ( display-attributes ) ]
```

1. *text* is any expression to be evaluated and displayed at the given position in the current window.
2. *line* is an integer expression defining the line position in the current window.
3. *column* is an integer expression defining the column position on the screen.
4. *display-attributes* defines the display attributes for the *text*.

Usage

The DISPLAY AT instruction evaluates a string expression and displays the result at a given line and column in the current window. This instruction is typically used in text-based applications to display static text on the screen such as messages or decoration lines with - (dash) or _(underscore) characters.

The DISPLAY AT instruction should only be used in TUI mode. To display data at a given place in a graphical form, use form fields and the DISPLAY BY NAME or DISPLAY TO instructions, or use interative instructions with the UNBUFFERED mode to automatically display program variable data to form fields.

When using DISPLAY AT in GUI mode, the text will only be displayed if the current window contains no form, or contains a form defined with the SCREEN layout.

Table 226: Display-attributes supported by the DISPLAY AT statement

Attribute	Description
`BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW`	The TTY color of the displayed text.
`BOLD, DIM, INVISIBLE, NORMAL`	The TTY font attribute of the displayed text.
`REVERSE, BLINK, UNDERLINE`	The TTY video attribute of the displayed text.

Using images

Describes how to use pictures in the forms of your application.

- Image handling basics on page 782
- Controlling the image layout on page 783
- Providing the image resource on page 784
- Static images on page 787
- Runtime images on page 787

Image handling basics

This is an introduction to image handling in Genero.

Purpose of images in applications

Graphical applications typically use images for different purpose:

- Application icon for the operating system taskbar / window manager.
- Icons in popup messages, menu options, form buttons, toolbars, list elements, treeview nodes.
- Decoration pictures in forms like background images, company logo, etc.
- Application photos, to get a visual identification for objects or people.

Images can be static (like toolbar icons, logos), or can change during the program execution (images related to application data).

In `.per` form definition files, specify static or dynamic image form items, with the IMAGE item type on page 888.

Sources for image data

An image can come from different sources:

- An image file located on the system where the program executes (available on the platform, or from your own application).
- An URL (or URI) resource: the image file is located on a web server an can be downloaded from the internet.
- Image data stored in a database within Binary Large Object (BLOB) typed columns.
- Pictures coming from a mobile device photo gallery, or camera.

In all cases, the image data must be available locally on the front-end platform to be displayed. Since the program can run on a different platform as the front-end, Genero provides several solutions to transmit the image data to the front-end, when the image is not available as a local file. For more details, see Providing the image resource on page 784.

Image triggering actions

If needed, it is possible to associate an action to an image by defining the ACTION attribute. The associated action handler will then be executed in the program code, for example to react on mouse clicks on the image for desktop front-ends:

```
IMAGE i1: logo,
   IMAGE = "genero_logo",
   ACTION = show_about_box;
```

For more details about action handling, see Dialog actions on page 1276.

Controlling the image layout

Explains how image form items can be sized, according to the type of front-end layout system.

Image sizing basics

How an IMAGE item renders on the front-end screen depends on these factors:

- The type of layout used (grid-based or stack-based layout).
- The size of the form item tag in the LAYOUT section, or the WIDTH and HEIGHT attributes defined for the IMAGE item.
- The combination of image item attributes (SIZEPOLICY, AUTOSCALE, STRETCH). These attributes may have a limited effect based on the front-end platform.
- The image resource (actual picture file) size when displayed (especially when SIZEPOLICY=DYNAMIC/INITIAL).
- The scaleIcon presentation style attribute, for elements using icons such as BUTTON or TOOLBAR items.

Image size in grid-based layout

The AUTOSCALE attribute indicates if the picture must be scaled to the available space in the image item. The space is defined by the SIZEPOLICY attribute, the STRETCH attribute, and the form item size (the form item tag in the layout or the WIDTH and HEIGHT attributes).

The STRETCH attribute defines how the image item adapts to the parent container when it is re-sized. The default is NONE.

The SIZEPOLICY attribute defines how the image widget gets its size, according to the context:

- When SIZEPOLICY is INITIAL (the default) and AUTOSCALE is not used, the size of the widget is defined by the first picture displayed in the form element. The size will not change if other pictures with different sizes display in the widget.
- When SIZEPOLICY is DYNAMIC, the size of the widget is automatically adapted to the size of the pictures displayed in the image form item. The AUTOSCALE attribute makes no sense and will have no effect.
- If SIZEPOLICY attribute is set to FIXED, the size of the widget is defined by the form specification file, either by the size of the *item-tag* in the layout, or by the WIDTH and HEIGHT attributes. With a fixed image widget size, if AUTOSCALE is not used, scrollbars may appear if the picture is greater than the widget.

By default, the size of the image widget defaults to the relative width and height defined by the *item-tag* in the form layout section. The size of an image widget can also be specified in the WIDTH and HEIGHT attributes, but these attributes will only have an effect when SIZEPOLICY=FIXED.

> **Note:** On some platforms, the image widgets automatically add a border to the source picture. For these platforms, if the image form item is the same size as the image, you may need to increase the size of the image form item to avoid automatic scrollbars. For example, if your image source has a size of 500x500 pixels and the widget displays a border with as size of 1 pixel, you will have to set WIDTH and HEIGHT to 502 pixels. If you do not, scrollbars will appear or the image will shrink

if AUTOSCALE is used. Alternatively, you can avoid the image border with the `border`presentation style attribute.

Image size in stack-based layout

With a stacked layout, where form items display vertically on each other, by default the image is auto-scaled with the correct aspect/ratio into the available form space.

The image size can be controlled by the HEIGHT attribute.

If the HEIGHT attribute is set, it is expressed in CHARACTERS as for grid-based layout, and the width is determined by the correct aspect/ratio.

Providing the image resource

There are several things you need to know about providing an image resource in a Genero program.

Supported image formats

Genero supports several image data formats, typically PNG, JPEG and SVG. Check the front-end platform documentation for supported image formats. True Type Font (TTF) files are also supported, the TTF format is used when image-to-font-glyph mapping is enabled by specifying a mapping file in the FGLIMAGEPATH environment variable.

Image resolution

Consider using the appropriate image resolution for the target front-end platform. For example, mobile devices have a much higher pixel density (a higher resolution) than desktop monitors. An image which looks nice on a desktop can appear small or as an upscaled image on a mobile device.

Static and dynamic image resources

The image resource specification is different for static and dynamic images:

- For static images (such as button icons), set the image resource in the image attribute (IMAGE, IMAGELEAF, and so on). See Static images on page 787.
- For dynamic images (such as image fields displaying photos from a database), the image resource is specified with the field/variable value, to be rendered in a form field. The form field is typically defined as an IMAGE item, or an IMAGECOLUMN in a table view. For more details, see Runtime images on page 787.

Image resource lookup

The image data can be provided in different ways, according to the image resource specification:

1. As a Uniform Resource Locator (URL).
2. As a simple image name (typical for icons).
3. As a simple file name, typically with a `.png` or `.jpg` extension, or a relative or absolute file path .

Using an URL image resource

If the image specification starts with a URL prefix, the front-end will try to download the image from the location specified by the URL.

The network access to the web server must exist and network bandwidth must be sufficient to rapidly download the images.

Table 227: Supported image resource locations (URLs)

Image resource location (URL)	Description
`http://location-specification`	HTTP server
`https://location-specification`	HTTPS server (HTTP over SSL)

Using a simple image name (centralized icons)

If the image specification is a simple name (without a file extension), and the FGLIMAGEPATH environment variable defines an icon mapping file for the runtime system, the image name is converted to a font file and font glyph according to the mapping file entries, and the image form item displays that glyph/icon. The mapping file and the font definition file are centralized on the application server.

A line in the image-to-font-glyph mapping file must have the following format:

```
image-name=font-file:hexa-ordinal[:color-spec]
```

For example, if the image mapping file defines the following lines:

```
smiley=FontAwesome.ttf:f118
red_smiley=FontAwesome.ttf:f118:#8B0000
```

An image resource (`IMAGE` attribute, `IMAGECOLUMN` value, and so on) with the name "`smiley`" will be mapped to the glyph with ordinal position `0xf118` in the `FontAwesome.ttf` font file, and the image resources using "`red_smiley`" will use the same glyph, but will get a red color.

> **Important:** The directory to the font file must be specified in FGLIMAGEPATH, except if the font file is located in the same directory as the mapping file.

A default color can be defined for all TTF icons of a window, by using the `defaultTTFColor` style attribute:

```
<StyleList>
  <Style name="Window.important">
     <StyleAttribute name="defaultTTFColor" value="red" />
  </Style>
  ...
```

A default mapping file named "`image2font.txt`" and the "`FontAwesome.ttf`" font file are provided in `FGLDIR/lib`. If FGLIMAGEPATH is not defined, the runtime system will use these files to make the image to font glyph mapping.

> **Important:** When providing your own customized font file, it must be a valid TTF file. For example, changing the file name is not sufficient to turn it into another different font: In order to produce a valid TTF file, use font management tools such as FontForge (http://fontforge.github.io/en-US/) or Fontello (http://fontello.com). Further, to target Microsoft Internet Explorer (version 11), you will need to patch the generated TTF file to remove embedding limitations from TrueType fonts, by setting the `fsType` field in the OS/2 table to zero. This modification can be done with freeware tools like ttembed

It is possible to mix several plain image file directories with several image-to-font glyph mapping files in FGLIMAGEPATH. The list of mapping files and directories defines the order of precedence, for example:

```
$ export FGLIMAGEPATH="/var/myapp/myimages:/var/myapp/myicons.txt:/var/
myapp/fontfiles:$FGLDIR/lib/image2font.txt:$FGLDIR/lib"
-- /var/myapp/myimages:           Directory where plain image files can be
 found
-- /var/myapp/myicons.txt:        custom image-to-font-glyph mapping file
 (icons)
```

```
-- /var/myapp/fontfiles:          Font files used by the myicons.txt mapping
 file
-- $FGLDIR/lib/image2font.txt:    Genero default icon mapping files
 (using FontAwesome.ttf)
```

Consider defining your own image mapping file and make FGLIMAGEPATH point to your own files.

> **Note:** When executing the application on a mobile device, you must define the FGLIMAGEPATH environment variable with the `mobile.environment.FGLIMAGEPATH` entry in FGLPROFILE. Use $FGLAPPDIR and $FGLDIR placeholders to include the current *`appdir`* (i.e. program file directory) and the FGL runtime system directory, respectively.

See FGLIMAGEPATH on page 182 for more details about this environment variable.

Using file names or paths

If the image specification is a simple file path (without an URL prefix, and typically with an image file extension), the front-end gets the image file from the runtime system. The image file is searched on the platform where the program executes. The runtime system uses FGLIMAGEPATH on page 182 environment variable when searching for the images. If FGLIMAGEPATH is not set, the current working directory is searched for the image files.

The front-end provides the name of the image file, and a list of supported file extensions. The runtime system searches for image files in different locations as described here: The search depends on the name of the image file, the list of directories defined in FGLIMAGEPATH, and the expected file extensions provided by the front-end.

For example:

- Name of the image file: "`mycalendar`"
- FGLIMAGEPATH="`/var/myapp/myicons/common:business`"
- Extensions provided by front-end: "`.jpg, .png`"

The search for the image file would be as follows:

1. `/var/myapp/myicons/common/mycalendar`
2. `/var/myapp/myicons/common/mycalendar.jpg`
3. `/var/myapp/myicons/common/mycalendar.png`
4. `business/mycalendar`
5. `business/mycalendar.jpg`
6. `business/mycalendar.png`

This search procedure using a proposal of file extensions was implemented to allow different type of front-ends to pass the type of image compression format required, so you can define the image name in your forms without any extension. However, it is much more efficient to specify the image file with a portable extension.

When FGLIMAGEPATH is defined, the current working directory is not searched. If you want to look for image files in the current working directory and in other directories, add "." to the FGLIMAGEPATH path list.

> **Note:** When specifying a file name as an image resource, consider using the extension (`.png`, `.jpg`), to avoid unnecessary file searching, trying different combinations with all supported formats (FGLIMAGEPATH). The file extension will also be used by the front-end to easily identify the compression format (for example, to define the Content-Type in an HTML entity).

Application images in Web Components

Web Components can display static images (part of the Web Component assets), and application images provided at runtime (for example, a photo gallery web component). In order to provide application images

to a Web Component, the program must use the `ui.Interface.filenameToURI()` method to convert the local file name to a URI that can be accessed by the front-end.

For more details, see Using image resources with the gICAPI web component on page 1430.

Static images
Describes how to decorate forms with icons.

Static image usage context

Static images are application pictures that do not change during program executing, like icons in toolbar buttons and window icons.

Static images can be defined in different contexts withing form definition, or configuration files:

- Global application icon for platform window managers (taskbars), by using the ui.Interface.setImage on page 1766 method. For mobile devices, the application icon should be provided in the installation package (.apk for Android™, .ipa for iOS).
- Window specific icons, with the `IMAGE` attribute in the `LAYOUT` definition of a form (recommended) or at runtime, with the ui.Window.setImage on page 1773 method (if it must be changed during program execution).
- As default icon for action action views, with the `IMAGE` action configuration attribute (in action defaults for example).
- As specific action view icons, directly in the form item definition with the `IMAGE` attribute (for toolbars, menu items, buttons, buttonedits, etc).
- Image form items (logos), defined by the `IMAGE item-tag : item-name` syntax, using the `IMAGE` attribute.
- Default treeview node icons, with the `IMAGEEXPANDED`, `IMAGECOLLAPSED`, `IMAGELEAF` attributes of a `TREE` container.

Static image examples

The following code example, defines an `ITEM` toolbar element using a icon, that is specified with the `IMAGE` attribute:

```
TOOLBAR
   ITEM print ( TEXT="Print", IMAGE="printer" )
```

Next example defines a `BUTTONEDIT` form field with an icon named "listchoice":

```
ATTRIBUTES
BUTTONEDIT f05 = customer.cust_city,
    ACTION=get_city,
    IMAGE="listchoice",
    ... ;
```

Runtime images
Explains how to display pictures at runtime.

Dynamic image usage context

Application images like photos or variable icons (in list views) are only known at runtime, and will be displayed during program execution. Such images are typically centralized on a server, as BLOBs in a database, or on the file system, as regular files.

For simple files (not URLs), images to be displayed are automatically handled by Genero: the program just needs to specify the name of the file to be displayed.

This section describes programming patterns to handle application images. For a complete description of the mechanisms to provide images to front-ends, see Providing the image resource on page 784.

IMAGE form fields

To display a picture dynamically in a form area, you must define a form field with the `IMAGE` item type:

```
LAYOUT
GRID
{
[img1                  ]
[                      ]
[                      ]
}
END
END
ATTRIBUTES
IMAGE img1 = FORMONLY.image_field, AUTOSCALE, ...
```

The program can then display an image dynamically by assigning the image resource to the form field, for example, with a `DISPLAY TO` instruction:

```
DEFINE image_field STRING
LET image_field = "local_image_file.png"
DISPLAY BY NAME image_field
```

It is also possible to use the program variable containing the image resource in a dialog using the `UNBUFFERED` option:

```
DEFINE rec RECORD
               pk INT,
               name VARCHAR(30),
               image_field VARCHAR(50)
           END RECORD
INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
   ON ACTION set_picture
      LET rec.image_field = "local_image_file.png"
...
```

IMAGECOLUMN attribute of TABLE/TREE

The `IMAGECOLUMN` attribute can be used to define a `PHANTOM` field that will hold the image resource for a `TABLE` or `TREE` column:

```
...
ATTRIBUTES
PHANTOM FORMONLY.item_icon;
EDIT FORMONLY.item_desc, IMAGECOLUMN=item_icon;
...
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.item_icon, FORMONLY.item_desc, ...);
...
```

In the program code, the image resource will be specified in the array member attached to the icon field. Each row can define a different image for the cell:

```
LET arr[1].item_icon = "honda_logo.png"
LET arr[1].item_desc = "Honda CB600 Hornet (red)"
LET arr[2].item_icon = "honda_logo.png"
```

```
LET arr[2].item_desc = "Honda CB1000r (black)"
LET arr[3].item_icon = "ducati_logo.png"
LET arr[3].item_desc = "Ducati Diavel Carbon"
DISPLAY ARRAY arr TO sr.*
   ...
```

Displaying images contained in BYTE variables

Application images managed by a program can be held in a `BYTE` variable. You need to use this data type to interface with databases storing images in Binary Large OBject (BLOB) columns.

When using an `IMAGE` field, if the BYTE variable holding the image data is located in a file (`LOCATE IN FILE`), the runtime system can automatically send the content of the `BYTE` file to the front-end when doing a `DISPLAY BY NAME`, `DISPLAY TO field`, or if the `BYTE` variable is controlled by a dialog using the `UNBUFFERED` option.

```
DEFINE pb BYTE
LOCATE pb IN FILE -- temp file used
...
OPEN FORM f1 FROM "myform"
DISPLAY FORM f1
...
SELECT image_col INTO pb FROM mytable WHERE pk = ...
DISPLAY pb TO image_field
...
```

Further, if the image data is modified, without changing the name of the file (i.e., without a new LOCATE IN FILE instruction), the runtime system detects the file modification time, and if needed, re-sends the image data to the front-end. For example, consider the following program flow:

```
DEFINE pb BYTE
LOCATE pb IN FILE -- temp file used
...
-- A first SELECT fetches image data from row 345 into the BYTE
SELECT image_col INTO pb FROM mytable WHERE pk = 345
-- And displays the BYTE image to a field
DISPLAY pb TO image_field
-- A second SELECT fetches new image data from row 672 into the BYTE
SELECT image_col INTO pb FROM mytable WHERE pk = 672
-- And displays the BYTE image to a field
DISPLAY pb TO image_field
-- The BYTE file name has not changed, only the image data has changed
...
```

Images on mobile devices

When executing the application on a mobile device, it is possible to use a front call to choose or take a photo. Those front calls return an opaque file identifier referencing an image in the device photo gallery (or database).

On all mobile platforms, you can directly display the returned opaque file path to an `IMAGE` form field:

```
DEFINE path STRING
-- Here we use "choosePhoto" front call, could be "takePhoto"
CALL ui.Interface.frontCall("mobile", "choosePhoto", [], [path])
DISPLAY path TO ff_image
```

Consider the path returned by such a front call as an opaque local file identifier, and do not use it as a persistent file name for the picture. For example, if you store such a path name in a database, and if the mobile photo gallery storage technology changes, the stored file names will no longer be valid.

If you need to keep the image data in the application (to store it in a local file or in the database), grab the image data into the runtime system context with a `fgl_getfile()` call. The mobile picture *path* can be used in a `fgl_getfile()` call to the photo from the mobile device into the file storage context where the runtime system executes. When the runtime system executes on the mobile device, the `fgl_getfile()` call will copy the picture to the application sandbox. If the program executes on an application server, the call will transfer the picture to the application server file system. It is possible to load the picture data into a BYTE variable, by transferring the image data directly into the file used by the BYTE variable locates in `byte_file`, by doing a `fgl_getfile(`*`mobile_path, byte_file`*`)`. It is also possible to keep the transferred files on the file system where the VM executes, if you do not want to use BYTE variables to store images in your database.

```
CONSTANT vm_fn = "mypic.tmp"
DEFINE md_fn STRING, image BYTE
CALL ui.Interface.frontCall(
        "mobile",
        "choosePhoto",  -- could be "takePhoto"
        [], [md_fn])
CALL fgl_getfile(md_fn,vm_fn)
LOCATE image IN FILE vm_fn
DISPLAY image TO ff_image
UPDATE mytab SET pic = image WHERE ...
```

> **Note:** When using `fgl_getfile()` in conjunction with BYTE variables located in files, pay attention to the fact that `INITIALIZE` *`byte_var`* `TO NULL` will set the internal null indicator of the BYTE variable, and a subsequent `fgl_getfile(`*`mobile_path, byte_file`*`)` will only modify the file without touching the null flag. The recommended pattern is to re-locate the BYTE variable after the `fgl_getfile()` call:
>
> ```
> CALL fgl_getfile(mobile_path, byte_file)
> LOCATE byte_var IN FILE byte_file
> ```

Videos on mobile devices

Let the user take videos or choose videos from the gallery with the takeVideo on page 1939 and chooseVideo on page 1927 front calls.

Similar to photo front calls, the video front calls return an opaque path to the video file, which can then be used in the `fgl_getfile()` function to transfer the video file from the device context to the runtime system context in a BYTE variable for persistent storage.

> **Note:** The opaque path can, however, be used to show the video with the `"launchURL"` front call.

For example:

```
IMPORT os

CONSTANT VM_MOVIES = "./movies"

MAIN
  DEFINE r INTEGER,
         mb_path STRING,
         vm_path STRING

  LET r = os.Path.delete(VM_MOVIES)
  LET r = os.Path.mkDir(VM_MOVIES)

  MENU
    COMMAND "take_video"
      CALL ui.Interface.Frontcall("mobile", "takeVideo", [], [mb_path])
      IF mb_path IS NOT NULL THEN
```

```
         LET vm_path = SFMT("%1/%2", VM_MOVIES, os.Path.baseName(mb_path) )
         CALL fgl_getfile(mb_path, vm_path)
      END IF
    COMMAND "choose_video"
      CALL ui.Interface.Frontcall("mobile", "chooseVideo", [], [mb_path])
      IF mb_path IS NOT NULL THEN
         LET vm_path = SFMT("%1/%2", VM_MOVIES, os.Path.baseName(mb_path) )
         CALL fgl_getfile(mb_path, vm_path)
      END IF
    COMMAND "show_video"
      IF mb_path IS NOT NULL THEN
         CALL ui.Interface.Frontcall("standard", "launchURL", [mb_path], [])
      END IF
    COMMAND "quit"
      EXIT MENU
  END MENU

END MAIN
```

Accessibility guidelines

This section describes the best practices to make a your application accessible to disabled people.

- Keyboard access on page 791
- Form description for screen readers on page 792
- Usability and ergonomics on page 793

Keyboard access

Defining keyboard accelerators for every action

Since a mouse or other pointing devices may not be used by people with reduced vision, an accessible application must be usable with the keyboard alone. Therefore, all the possible actions that could be triggered by a user must have a keyboard shortcut.

We strongly suggest that you define consistent keyboard shortcuts for all actions through the use of action defaults. Developers can avoid overriding the system default shortcuts by checking the target platform guidelines, especially for system shortcuts that trigger accessible actions (for example, Ctrl-Shift-Enter, which triggers spoken information about the currently selected item). Overriding system shortcuts is generally a bad practice, even for non-accessible applications, although overriding may be unavoidable due to compatibility issues.

Keyboard focus and action views

Generally, keyboard navigation in an application may be easier if you keep the `MENU` actions in the menu frame; the actions can have the keyboard focus and the user can navigate through them using the up and down arrows.

You can also use a `TOPMENU`, because you can pull it down with the keyboard (for example, the Alt key on Windows™) and then navigate using arrow keys, but it may be less accessible than the menu panel. You must also be sure that every item of the menu can be activated by a keyboard shortcut. You may use the & (ampersand) in menu items to specify character which letter should be used, to let the front-end automatically create a shortcut to trigger the action with that letter.

Avoid using toolbars only in an accessible application, because toolbars by default are not accessible using the keyboard. Toolbars cannot have the keyboard focus, and there is no way to navigate through all toolbar items or to activate one of them using the keyboard. If you do use toolbars, provide keyboard shortcuts and duplicate them in a topmenu.

Form description for screen readers

Understanding screen readers

Screen readers are special system applications that transform the application's graphical user interface into speech. The behavior may change between screen reader implementations, but, basically, each widget is named and described by speech. On some workstation operating systems, special keyboard shortcuts are available to trigger the complete enumeration of all the components of a window, or to describe only the component having the current focus.

Providing form item descriptions to screen readers

Screen readers use special bindings to get the information that they need (name, full description, hierarchy, triggered actions, and so on) about each graphical component of the entire graphical user interface. It is up to the programmer to provide these bindings to the screen reader, but most of the work is already done by the front-end.

Programmers can provide two things for each widget to provide speech information to screen readers:

- an accessible **name**, using the `TEXT` form attribute if available, otherwise with the `COMMENT` form attribute.
- an accessible **description**, with the `COMMENT` form attribute.

This can be tedious, but it absolutely must be done carefully, keeping in mind that the text will be spoken. As such, *customer's name* is preferable to *cust_name_str*.

Spaces and punctuation are allowed.

Most of the form items are supported: All kind of form field, static labels, static images, and action-based items (such as buttons); some containers (`GROUP` and `FOLDER`) should work out of the box as soon as their `TEXT` attributes are set.

Examples

In an action defaults file (mydefaults.4ad)

```
<ActionDefaultList>
  <ActionDefault name="new" text="New..." image="new.svg"
     comment="Create a new database"
   acceleratorName="control-n" />
  <ActionDefault name="open" text="Open..." image="open.svg"
     comment="Open an existing database"
   acceleratorName="control-o" />
  <ActionDefault name="save" text="Save" image="save.svg"
     comment="Save the current database"
   acceleratorName="control-s" />
  ...
```

In field definitions on a form specification file (myform.per)

```
ATTRIBUTES
  EDIT login_name = formonly.login_name, NOT NULL,
    COMMENT="Login name of the current user";
  EDIT password = formonly.password, NOT NULL, INVISIBLE,
 VERIFY,
    COMMENT="Password of the current user";
  EDIT first_name = formonly.first_name, NOT NULL,
    COMMENT="First name of the current user";
  EDIT last_name = formonly.last_name, NOT NULL,
    COMMENT="Last name of the current user";
  DATEEDIT birthdate = formonly.birthdate, FORMAT="mm/dd/yyyy",
    COMMENT="Date of birth of the current user";
```

```
  EDIT email = formonly.email,
    COMMENT="E-mail of the current user";
END -- ATTRIBUTES
```

In this form specification file, the `COMMENT` attribute is used for both the accessible name and the accessible description.

Usability and ergonomics

Design simple application forms

Keep your forms as simple as possible. Because everything will be described by the screen reader software, it is preferable to have a lot of small and concise forms with a few fields. With forms containing a lot of labels and fields, the screen reader will take a long time to enumerate every name and description. The end user must be able to make a picture of the form in their mind, according to the form description.

Make form content bigger

Consider using a special .4st presentation styles file defining big fonts, big icons, and high contrast color themes; This will make your application a lot more efficient for users who are partially sited. Forms will take more space on the screen, assuming that the forms have a limited number of fields to have sufficient room for large widgets.

Use large icons (such as 64x64 pixel icons), for people with impaired vision. Do not forget that most of the default sizes (font, icons, gui components, and so on) were set when the default resolution was 640*640 pixels in 16 colors. Now, even if the user has very good eyes, with the screen resolution available today, old-style icons look small.

Use a high contrast color theme. Although support of the system high contrast theme is only partial, nothing prevents you from setting up the correct theme using a specific presentation style attributes.

Example

Presentation styles file defining larger, bolder fonts and large icons:

```
<StyleList>

  <Style name="*" >
    <StyleAttribute name="fontSize" value="10" />
  </Style>

  <Style name="Action" >
    <StyleAttribute name="scaleIcon" value="28px"/>
    <StyleAttribute name="fontSize" value="12" />
  </Style>

  <Style name="Window" >
    <StyleAttribute name="actionPanelPosition" value="bottom"/>
    <StyleAttribute name="actionPanelButtonSpace" value="huge"/>
    <StyleAttribute name="actionPanelHAlign" value="center"/>
    <StyleAttribute name="ringMenuPosition" value="bottom"/>
    <StyleAttribute name="ringMenuButtonSpace" value="huge"/>
    <StyleAttribute name="ringMenuHAlign" value="center"/>
  </Style>

  <Style name="ToolBar" >
    <StyleAttribute name="scaleIcon" value="32px"/>
  </Style>

  <Style name="Edit:focus" >
    <StyleAttribute name="fontWeight" value="bold" />
```

```
    <StyleAttribute name="backgroundColor" value="darkBlue" />
    <StyleAttribute name="textColor" value="white" />
  </Style>

</StyleList>
```

Message files

Message files centralize strings and larger texts identified by a number, that can be used in programs.

- Understanding message files on page 794
- Syntax of message files (.msg) on page 794
- Using message files on page 795
  - Compiling message files on page 795
  - Using message files at runtime on page 795
- Examples on page 796
  - Example 1: Help message file used in a MENU on page 796

Understanding message files

Message files define text messages with a unique integer identifier.

Several message files can be created and loaded by the same program.

Message files are typically used to implement application help system, and are especially designed for the the TUI mode.

In order to use a message file, do the following:

1. Create the .msg source message file with a text editor.
2. Compile the source message file with fglmkmsg to create the .iem binary format.
3. Copy the binary file to a distribution directory.
4. In programs, specify the message file with the `OPTIONS HELP FILE` instruction.
5. Use a specific message with the `HELP` clause of dialogs, or load a given message with the `SHOWHELP( )` function.

Message files provide a simple way to implement a help system in your application.

For other application messages and texts, consider using localized strings instead of message files.

Syntax of message files (.msg)

A message file contains a set of messages identified by an integer number.

```
filename.msg
```

1. *filename* is the name of the message source file.

Syntax of a message file

```
{
  message-definition
| include-directive
}[...]
```

where *message-definition* is:

```
.message-number
message-line | new-page
```

```
[...]
```

where *include-directive* is:

```
.include filename
```

And where *new-page* is:

```
^L (Control-L, ASCII 12)
```

1. *message-number* is an integer in the range -2147483648 to 2147483647.
2. You can split the message into pages by adding the ^L (Control-L / ASCII 12) in a line.
3. Note that multi-line messages will include the newline (ASCII 10) characters.

Using message files
To use message files, you must understand how they work and how to structure the code.
Compiling message files

In order to use message files in a program, the message source files (with .msg extension) must be compiled with the fglmkmsg utility to produce compiled message files (with .iem extension).

The following command line compiles the message source file mess01.msg:

```
fglmkmsg mess01.msg
```

This creates the compiled message file mess01.iem.

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

The .iem compiled version of the message file must be distributed on the machine where the programs are executed.

Using message files at runtime

In order to use compiled message files (`.iem`) in programs, specify the current message file with the `OPTIONS HELP FILE` command:

```
OPTIONS HELP FILE "mymessages.iem"
```

The message file will first be searched with the string passed to the `OPTIONS HELP FILE` command (i.e. the current directory if the file is specified without a path), and if not found, the DBPATH / FGLRESOURCEPATH environment variable with be used.

After the message file is defined, you can start the help viewer by calling the `SHOWHELP()` function:

```
CALL showhelp(1242)
```

Use the `HELP` clause in a dialog instruction such as `INPUT` to define particular message number for that the dialog:

```
INPUT BY NAME ... HELP 455
```

The help viewer will automatically display the message text corresponding to the number when the user pressed the help key. By default, the help key is Ctrl-W in TUI mode and F1 in GUI mode.

Note that you can implement your own help viewer by overloading the `SHOWHELP()` function defined in `FGLDIR/src/fglhelp.4gl`. This allows you to customize the help system for your application.

Examples

Example 1: Help message file used in a MENU

The message source file help.msg:

```
.101
This is help about option 1
.102
This is help about help
.103
This is help about My Menu
```

Compiling the message file:

```
$ fglmkmsg help.msg
```

Program using the .iem compiled message file.

```
MAIN
    OPTIONS
        HELP FILE "help.iem"
    MENU "Sample"
        COMMAND "Option 1" HELP 101
            DISPLAY "Option 1 chosen"
        COMMAND "Help"
            CALL showhelp(103)
    END MENU
END MAIN
```

Action defaults files

Action defaults files allow to centralize action configuration parameters such as text, icon, accelerators and behavior options in XML format.

- Understanding action defaults files on page 796
- Syntax of action defaults file (.4ad) on page 796
- Action default attributes reference (.4ad) on page 797
- Examples on page 799
  - Example 1: Loading a global action defaults file on page 799

Understanding action defaults files

Action defaults files define the defaults for action attributes in an XML file. These defaults can be overwritten with form item attributes, or with dialog action handler attributes, when using default action views.

This section describes only the `.4ad` action defaults file reference, for more details see Configuring actions on page 1318.

Syntax of action defaults file (.4ad)

Action defaults are defined in the .4ad file with this syntax:

```
<ActionDefaultList>
  <ActionDefault name="action-name" [ attribute=value [...] ] />
  [...]
</ActionDefaultList>
```

1. *action-name* identifies the action.
2. *attribute* is the name of an attribute.

3. *value* defines the value to be assigned to *attribute*.

Action default attributes reference (.4ad)

Table 228: Action default attributes

Attribute	Description
`name = "action-name"`	This attribute identifies the action.
`text = "action-label"`	The default label to be displayed in action views (typically, the text of buttons). See also: TEXT attribute on page 987
`comment = "action-comment"`	The default help text for this action (typically, displayed as bubble help). See also: COMMENT attribute on page 959
`image = "action-icon"`	The default image file to be displayed in the action view. See also: IMAGE attribute on page 967
`acceleratorName = "key-name"`	The default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1343. See also: ACCELERATOR attribute on page 953
`acceleratorName2 = "key-name"`	The second default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1343. See also: ACCELERATOR2 attribute on page 953
`acceleratorName3 = "key-name"`	The third default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1343. See also: ACCELERATOR3 attribute on page 954
`acceleratorName4 = "key-name"`	The fourth default accelerator key that can trigger the action, as defined in Keyboard accelerator names on page 1343. See also: ACCELERATOR4 attribute on page 954
`defaultView = {"yes"|"no"|"auto"}`	Defines whether the front-end must show the default action view (buttons in control frame). Values can be: • `"no"` the default action view is never visible. • `"yes"` the default action view is always visible, if the action is visible (`ui.Dialog.setActionHidden`).

Attribute	Description
	• `"auto"` the default action view is visible if no other action view is explicitly defined and the action is visible (`ui.Dialog.setActionHidden`). The default is `"auto"`. See also: DEFAULTVIEW attribute on page 961
`contextMenu = {"yes"|"no"|"auto"}`	Defines whether the front-end must render the action in the default context menu. Values can be: • `"no"` the context menu option is never visible. • `"yes"` the context menu option is always visible, if the action is visible (`ui.Dialog.setActionHidden`). • `"auto"` the context menu option is visible if no other action view is explicitly defined and the action is visible (`ui.Dialog.setActionHidden`). The default is `"yes"`. See also: CONTEXTMENU attribute on page 958
`validate = "no"`	Defines the behavior of data validation when the action is invoked. Values can be: • `"no"` no data validation is done (field text only available in input buffer). By default, data validation is driven by the dialog mode (`UNBUFFERED` or default mode). For more details, see Data validation at action invocation on page 1331. See also: VALIDATE attribute on page 991

Using action defaults files

To use action default files, you must understand how they work and how to structure the code.

Using action defaults files at runtime

Global action defaults are defined in an XML file with the `4ad` extension. By default, the runtime system searches for a file named `default.4ad` in the current directory. If the file does not exist, it searches in the directories defined by the FGLRESOURCEPATH (or DBPATH) environment variable. If no file was found using the environment variable(s), standard action default settings are loaded from the `FGLDIR/lib/default.4ad` file.

Action defaults files usage is related to action configuration concepts. For more details, see Configuring actions on page 1318.

Examples
Example 1: Loading a global action defaults file

Some Action Defaults in XML format (exit action has Localized Strings):

```
<ActionDefaultList>
   <ActionDefault name="print" text="Print" image="printer"
       comment="Print report" />
   <ActionDefault name="modify" text="Update"
       comment="Update the record" />
   <ActionDefault name="exit" text="Quit" image="byebye"
       comment="Exit the program" validate="no" >
       <LStr text="common.exit.text" />
   </ActionDefault>
</ActionDefaultList>
```

The program loading the action defaults file:

```
MAIN
  CALL ui.Interface.loadActionDefaults("mydefaults")
  OPEN FORM f FROM "myform"
  DISPLAY FORM f
  ...
END MAIN
```

Presentation styles

Use presentation styles to specify decoration attributes for window and form elements.

- Understanding presentation styles on page 799
- Syntax of presentation styles file on page 801
- Using presentation styles on page 801
- Predefined attribute values on page 807
- Style attributes reference on page 818
- Examples on page 849

Understanding presentation styles

Presentation styles centralize the attributes related to the decoration of the graphical user interface elements.

The decoration attributes are defined in a separate file, which can be easily modified to customize the application.

Presentation styles are only supported for the GUI front-ends. If you design an application for the TUI mode, you can use TTY attributes.

Styles are applied implicitly by using global styles, or explicitly by naming a specific style in the `style` attribute of the element.

Common presentation attributes define font properties, foreground colors and background colors. Some presentation attributes are specific to a given class of widgets (like the first day of the week in a `DATEEDIT`).

Presentation styles are defined in a resource file having an extension of `.4st`. The `.4st` file must be distributed with the other runtime files.

Presentation styles are inspired from the *cascading style sheets* (CSS) used in HTML, with the following deviations:

1. The elements using style definitions are AUI tree elements; CSS styles apply to HTML elements.
2. To specify a style for an AUI tree element, you must use the "style" attribute; HTML/CSS use the "class" attribute.

3. Inline-style definition is not supported in the AUI tree.
4. Some pseudo selectors, such as "query, are specific to Genero.

Figure 41: Form without presentation styles (GDC) on page 800 shows a desktop application without presentation styles:

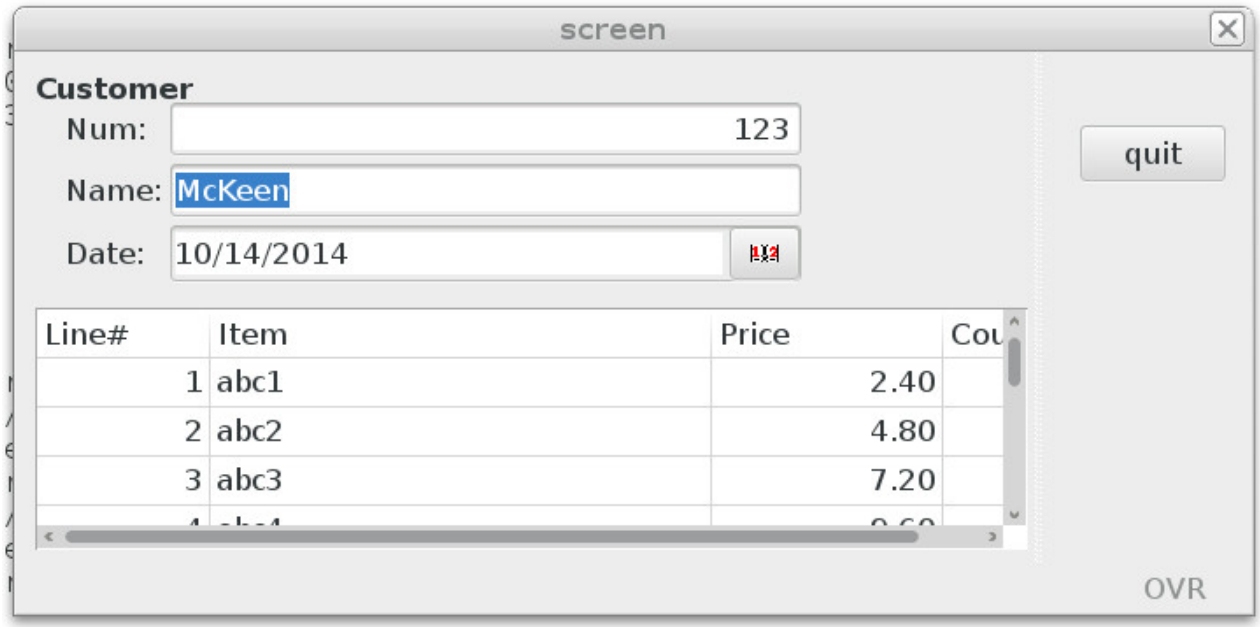


Figure 41: Form without presentation styles (GDC)

Figure 42: Form using presentation styles (GDC) on page 800 shows a desktop application without presentation styles:

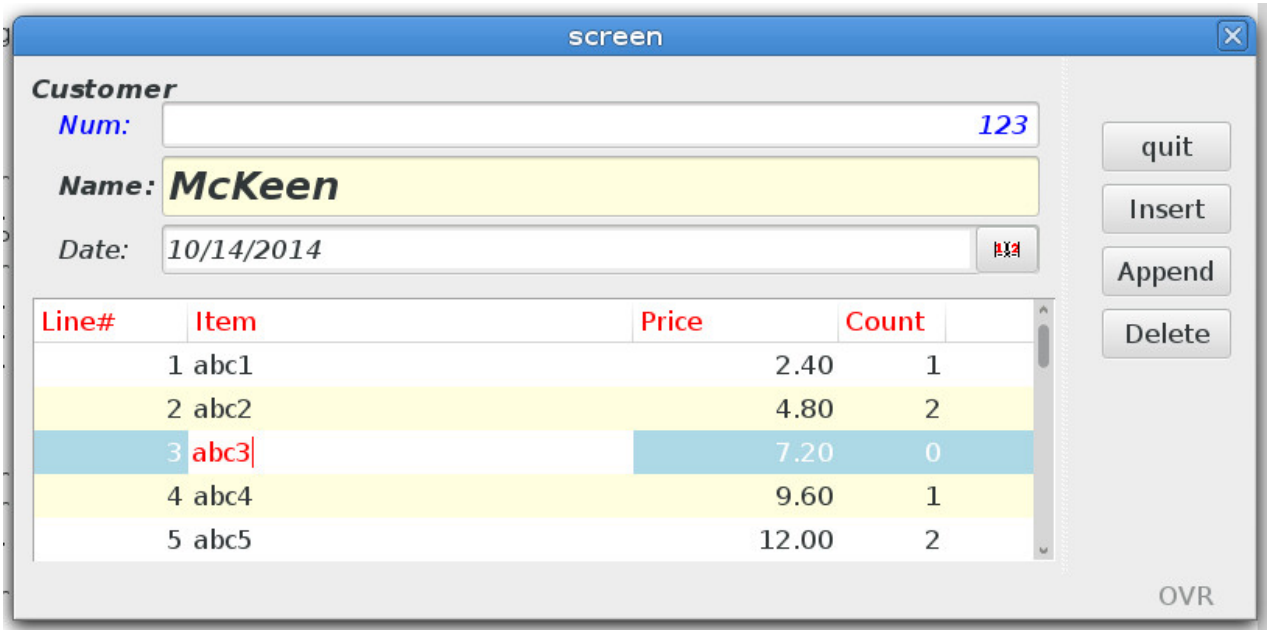


Figure 42: Form using presentation styles (GDC)

Syntax of presentation styles file

A presentation styles file (4st) is an XML file comprised of `StyleList`, `Style`, and `StyleAttribute` elements.

Syntax (.4st)

```
<StyleList>
  <Style name="style-identifier" >
    <StyleAttribute name="attribute-name" value="attribute-value" />
      [...]
  </Style>
  [...]
</StyleList>
```

where *style-identifier* is:

```
{ [ element-type ] [ .style-name ] [ :pseudo-selector ]
| *
}
```

1. *element-type* is a type of AUI tree element, such as `Edit`, `Window`.
2. *style-name* is an explicit style name, that can be referenced in `STYLE` attributes of form items.
3. *pseudo-selector* indicates in what context the style should apply.
4. *attribute-name* defines the name of the style attribute.
5. *attribute-value* defines the value to be assigned to *attribute-name*.

Syntax of attribute values

Presentation style attribute values are always specified as strings, for example:

```
<StyleAttribute name="fontFamily" value="Serif" />
```

Numeric values must be specified in quotes:

```
<StyleAttribute name="completionTimeout" value="60" />
```

Boolean values must be specified with the values `"yes"` or `"no"`:

```
<StyleAttribute name="forceDefaultSettings" value="yes" />
```

> **Note:** Some front-ends may also support the boolean values `0/1` and `true/false`. However, it is recommended to use `yes/no` values only.

Using presentation styles

Use presentation styles to centralize the decoration of your user interface.

- Understanding presentation styles on page 799
- Defining a style on page 802
- Pseudo selectors on page 802
- Using a style on page 804
- Order of precedence on page 804
- Combining styles on page 805
- Style attribute inheritance on page 805
- Presentation styles in the AUI tree on page 805
- Loading presentation styles on page 805
- Combining TTY and style attributes on page 806

- Element types on page 807

Defining a style
Styles can be defined to be global (for all elements), for an element in general, or for specific types of an element.

The style is identified by the `name` attribute, that can be a combination of element type, style name and pseudo selector, or the star character. See Syntax of presentation styles file on page 801 for a complete description of the presentation style definition syntax.

In the definition of a style, the `name` attribute is used as a selector to apply style attributes to graphical elements.

You can define a style as global or specific to a class of graphical object:

- A style identified by a star (`*`) is a global style that is automatically applied to all elements:

```
<Style name="*" >
```

- A style identified by an *element-type* is a global style that is automatically applied to all objects of this type:

```
<Style name="ComboBox" >
```

- A style identified by a *style-name* is a specific style that can be applied to any element types using that style name in a `STYLE` attribute:

```
<Style name=".important" >
```

- A style identified by an *element-type* followed by a dot and a *style-name* is a specific style that will only be applied to elements of the given type and using the style name in a `STYLE` attribute:

```
<Style name="Window.main" >
```

- A style identified by an *element-type* followed by a colon and a *pseudo-selector* is a style that will only be applied to elements of the given type, if the condition defined by the pseudo-selector is satisfied:

```
<Style name="Edit:focus" >
```

- A style identified by an *element-type* followed by a dot and a *style-name*, and a colon with a pseudo-selector, is a specific style that will only be applied to elements of the given type, using the style name in a `STYLE` attribute, if the condition defined by the pseudo-selector is satisfied:

```
<Style name="Edit.important:focus" >
```

- It is possible to combine pseudo-selectors:

```
<Style name="Edit:query:focus" >
```

Pseudo selectors
Pseudo selectors can be used to apply only when some conditions are fulfilled.

Pseudo selectors are preceded with a colon and can be combined:

```
<Style name="Table:even:input" >
<Style name="Edit:focus" >
<Style name="Edit.important:focus" >
```

When combining several pseudo selectors, the style will be applied if all pseudo selector conditions are fulfilled.

Note: Depending on the type of the front-end, some pseudo selectors are meaningless, or unsupported. See the table below to check which pseudo selectors are supported on your front-end platform.

Pseudo selectors have different priorities; the style with the most important pseudo selector will be used when several styles match.

Table 229: Pseudo selectors for presentation styles

Priority	Pseudo selectors	Condition	GDC	HTML5	GMA	GMI
1	`focus`	The widget has the focus	Yes	Yes	Yes	Yes
2	`query`	The widget is in construct mode	Yes	Yes	Yes	Yes
3	`display`	The widget is in a display array	Yes	Yes	Yes	Yes
4	`input`	The widget is in an input array, input or construct	Yes	Yes	Yes	Yes
5	`even`	This widget is on an even row if an list (Table or Tree)	Yes	Yes	No	No
6	`odd`	This widget is on an odd row if an list (Table or Tree)	Yes	Yes	No	No
7	`inactive`	The widget is inactive	Yes	Yes	Yes	Yes
8	`active`	The widget is active	Yes	Yes	Yes	Yes
9	`message`	Applies only to text displayed with the `MESSAGE` instruction	Yes	Yes	Yes	Yes
10	`error`	Applies only to text displayed with the `ERROR` instruction	Yes	Yes	Yes	Yes
11	`summaryLine`	Applies only to text displayed in `AGGREGATE` fields of tables	Yes	Yes	No	No

Pseudo selectors also define the priority of your styles. A more generic style will be used when the pseudo-selector has a higher priority.

For instance: you want all important edits to have red text, but you want the current field to be displayed in blue:

```
<Style name="Edit.important" >
<Style name=":focus" >
```

The style "`:focus`" is more generic than "`Edit.important`"; therefore, it will be used for the focused item, as the pseudo selector is more precise.

Using a style

To apply a specific style, set the *style-name* in the `style` attribute of the node representing the graphical element in the abstract user interface tree.

There are different ways to set the `style` attribute of a element:

- As a form element attribute, with a `STYLE` attribute in the form specification file.
- In the `ATTRIBUTES` clause of instructions such as `OPEN WINDOW`, `MESSAGE`, `ERROR`.
- Dynamically by a program, using the ui.Form.setElementStyle() method.

For example, to define a style in a form file for a input field:

```
EDIT f001 = customer.fname, STYLE = "info";
```

> **Note:** The string used to define the `STYLE` attribute must be a *style-name* only, it must not contain the *element-type* that is typically used to define the style in a `.4st` file (as `CheckBox.important` for example)

Order of precedence

Style definitions are applied according to the order of precedence.

If different styles can be applied to an element, the following priority is used to determine the style definition to be applied:

1. *element-type*.*style-name*:*pseudo-selector*
2. .*style-name*:*pseudo-selector*
3. *element-type*.*style-name*
4. *element-type*:*pseudo-selector*
5. :*pseudo-selector*
6. .*style-name*
7. *element-type*
8. *

> **Note:** The precedence rules to apply styles may vary according to the front-end type. As a general rule, Genero presentation styles precedence rules are similar HTML/CSS precedence rules.

For example, consider an `Edit` element with the style attribute set to '`mandatory`':

```
EDIT f1 = FORMONLY.cust_name, STYLE="mandatory"
```

With the following style definitions (`mystyles.4st`):

```
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <Style name="Edit.mandatory:focus">
     <StyleAttribute name="backgroundColor" value="yellow" />
  </Style>
  <Style name=".mandatory:focus">
     <StyleAttribute name="backgroundColor" value="blue" />
  </Style>
  <Style name="Edit.mandatory">
     <StyleAttribute name="backgroundColor" value="green" />
  </Style>
  <Style name="Edit:focus">
     <StyleAttribute name="backgroundColor" value="red" />
  </Style>
  <Style name=":focus">
     <StyleAttribute name="backgroundColor" value="cyan" />
  </Style>
  <Style name=".mandatory">
     <StyleAttribute name="backgroundColor" value="magenta" />
```

```
  </Style>
  <Style name="*">
     <StyleAttribute name="backgroundColor" value="orange" />
  </Style>
</StyleList>
```

The style definitions are scanned in the following order:

1. `Edit.mandatory:focus`
2. `.mandatory:focus`
3. `Edit.mandatory`
4. `Edit:focus`
5. `:focus`
6. `.mandatory`
7. `Edit`
8. `*`

If the Edit field `f1` has the focus, with the `mystyles.4st` definition file, the field background color will be yellow. If the Edit field `f1` does not have the focus, the field background color will be green.

Combining styles
You can combine several styles, by using the space character as a separator in the `STYLE` attribute.

In the following example, the `STYLE` attribute defines three different style names:

```
EDIT f001 = customer.fname, STYLE = "info highlight mandatory";
```

When several styles are combined, the same presentation attribute might be defined by different styles. In this case, the first style listed that defines the attribute takes precedence over the other styles.

For example, if the `textColor` presentation attribute is defined as follows by the `info`, `highlight` and `mandatory` styles:

- `info` style does not define `textColor`.
- `highlight` style defines `textColor` as `blue`.
- `mandatory` style defines `textColor` as `red`.

The widgets having a style set to `"info highlight mandatory"` will get a blue text color, because `highlight` is listed before `mandatory`.

Style attribute inheritance
A style attribute may be inherited by the descendants of a given node in the abstract user interface tree.

For example, when using a style defining a `fontFamily` in a window container, you would expect that all the children in that group box get the same font.

However, some style attributes should are not inherited, when specific to a given type of form element. Style inheritance is implicitly defined by the attribute.

Presentation styles in the AUI tree
Presentation styles are loaded in the abstract user interface tree, under the `UserInterface` node, in a `StyleList` node following the presentation style syntax.

The `StyleList` node holds a list of `Style` nodes that define a set of attribute values. Attribute values are defined in `StyleAttribute` nodes, with a `name` and a `value` attribute.

Loading presentation styles
Presentation styles are defined in an XML file with a `4st` extension. In order to load the presentation styles, the engine needs to locate the appropriate style file.

By default, the runtime system searches for a file named `default.4st` in the current directory. If this file does not exist, it searches in the directories defined by the `FGLRESOURCEPATH` / `DBPATH` environment

variables. If the file was not found using the FGLRESOURCEPATH / DBPATH environment variables, default presentation styles are loaded from the FGLDIR/lib/default.4st file.

Overwrite the default search by loading a specific presentation style file with the ui.Interface.loadStyles() method:

```
MAIN
  CALL ui.Interface.loadStyles("mystyles")
  ...
END MAIN
```

This method accepts an absolute path with the 4st extension, or a simple file name without the 4st extension. If you give a simple file name, for example "mystyles", the runtime system searches for the mystyles.4st file in the current directory. If the file does not exist, it searches in the directories defined by the FGLRESOURCEPATH environment variable. If FGLRESOURCEPATH is not defined, it searches in the directories defined by the DBPATH environment variable.

The presentation styles must be defined in a unique 4st file. When loading a styles file with the ui.Interface.loadStyles() method, current styles created from the default file or from a prior load will be replaced. The styles will not be combined when loading several files.

The default styles file located in FGLDIR/lib should not be modified directly: your changes would be lost if you upgrade the product. Make a copy if the original file into the program directory of your application, then modify the copied file.

Combining TTY and style attributes

TTY attributes can be specific to a form element or can be inherited by an element from a parent node (such as the form or window).

Specific element TTY attributes are directly set in the element node in the AUI tree; they can, for example, be defined with the COLOR attribute of form items. Inherited TTY attributes are taken from the parent nodes of the leaf element to be displayed. For example, when a form is displayed with DISPLAY FORM followed by an ATTRIBUTE clause containing TTY color, font option and/or video attributes, all static labels will be displayed with the TTY attributes of the form. Note however that the form elements controlled by interactive instructions (i.e. form fields) will explicitly get the TTY attributes defined by the ATTRIBUTE clause of OPEN WINDOW, OPEN FORM, DISPLAY TO / BY NAME or the current dialog statement, and must be considered specific TTY attributes for the element.

Specific TTY attributes defined for a form element have a higher priority than style attributes, while inherited TTY attributes (set on one of the parent elements) have a lower priority than style attributes defined for the element.

To illustrate this rule, imagine a form defining two static labels and two fields, with all items using the *mystyle* presentation style, and one of the labels and fields defining a specific TTY attribute with COLOR=BLUE:

```
LABEL lab01: TEXT="Field 1:", COLOR = BLUE, STYLE = "mystyle";
EDIT fld01 = FORMONLY.field01, COLOR = BLUE, STYLE = "mystyle";
LABEL lab02: TEXT="Field 2:", STYLE = "mystyle";
EDIT fld02 = FORMONLY.field02, STYLE = "mystyle";
```

The program displays the form (or window) with an ATTRIBUTES clause using a red color, and the fields are used by an INPUT dialog, with no ATTRIBUTES clause, so the default TTY attributes are gotten from the OPEN FORM instruction:

```
OPEN FORM f FROM "ttyform"
DISPLAY FORM f ATTRIBUTES(RED)
INPUT BY NAME field01, field02 WITHOUT DEFAULTS
```

The .4st styles file defines the *mystyle* attributes as follows:

```
<StyleList>
  <Style name="Edit.mystyle">
    <StyleAttribute name="textColor" value="green" />
  </Style>
  <Style name="Label.mystyle">
    <StyleAttribute name="textColor" value="magenta" />
  </Style>
</StyleList>
```

The text in the form field *fld01* is displayed in blue (from the specific `COLOR` attribute), while *fld02* is displayed in red (the TTY attribute of the form, the style *Edit.mystyle* being ignored).

Since labels are not used by the interactive instructions, *lab01* is displayed in blue (from the specific `COLOR` attribute), while *lab02* is displayed in magenta (from the style *Label.mystyle*, the form TTY attribute red being ignored).

Element types
Styles may apply to any graphical elements of the user interface, such as `Button`, `Edit`, `ComboBox`, `ButtonEdit`, `Table`, `Window`.

The name of the element when used in a style file is case-sensitive (use `CheckBox`, not `checkbox`).

For example, in the following style definition uses the "Window" element type in the style name:

```
<Style name="Window.dialog">
   <StyleAttribute name="position" value="center" />
</Style>
```

The supported element types is defined by the style attributes, for more details, see Style attributes reference on page 818.

Predefined attribute values
This section decribes the values that must be used for some style attributes.

- Colors on page 807
- Fonts on page 812
- Statusbar types on page 817

Colors
When providing a value for style attributes that define a color, you can specify a generic color name or its RGB value.

This section describes how to specify a value for style attributes defining colors, such as `textColor`.

Syntax

```
{ generic-color | #rrggbb }
```

1. *generic-color* is any of the predefined colors supported by the language.
2. *#rrggbb* is a numerical color defined by a red/green/blue specification.

Usage

In most cases it is not possible to know what a potential end-user might expect regarding the font family. Therefore, your application should avoid the usage of explicit font families and use only the `fontWeight`/`fontStyle`/`fontSize` properties. A specific font family should be used only if the client can't determine a proper default font family for the desired platform.

The language defines a set of generic color names, interpreted by the front end according to the graphical capability of the workstation.

Generic color names

Use generic color names, to keep your style definitions portable accros several front-end types.

Table 230: Generic color names

Generic color name	Visual result (1)	RGB value
`black`		`#000000`
`blue`		`#0000FF`
`cyan`		`#00FFFF`
`darkBlue`		`#00008B`
`darkCyan`		`#008B8B`
`darkGray`		`#A9A9A9`
`darkGreen`		`#006400`
`darkMagenta`		`#8B008B`

Generic color name	Visual result (1)	RGB value
darkOlive		#505000
darkOrange		#FF8C00
darkRed		#8B0000
darkTeal		#005050
darkYellow		#AAAA00
gray		#808080
green		#008000
lightBlue		#ADD8E6
lightCyan		#E0FFFF
lightGray		#D3D3D3

Generic color name	Visual result (1)	RGB value
`lightGreen`		`#90EE90`
`lightMagenta`		`#FFC0FF`
`lightOlive`		`#AAAA44`
`lightOrange`		`#FFCC00`
`lightRed`		`#FF8080`
`lightTeal`		`#33CCCC`
`lightYellow`		`#FFFFE0`
`magenta`		`#FF00FF`
`olive`		`#808000`

Generic color name	Visual result (1)	RGB value
`orange`		`#FFA500`
`red`		`#FF0000`
`teal`		`#008080`
`white`		`#FFFFFF`
`yellow`		`#FFFF00`

Note:

1. The exact rendered color depends on front-end type.

System color names

System color names can be used to get a color from the current theme of the workstation windowing system:

Table 231: System color names

System color name	Meaning
`appWorkSpace`	Background color of multiple document interface
`background`	Desktop background
`buttonFace`	Face color for three-dimensional display elements.
`buttonText`	Text on push buttons.
`grayText`	Grayed (disabled) text.
`highLight`	Item(s) selected in a control.
`highLightText`	Text of item(s) selected in a control
`infoBackground`	Background color for tooltip controls.
`infoText`	Text color for tooltip controls.
`systemAlternateBackground`	Background color of the alternate row in listviews
`window`	Window background.
`windowText`	Text in windows.

RGB notation

In some cases, you may also specify a color with the RGB notation, starting with a # hash character.

Each value of the RGB color specification must be provided in hexadecimal, in the range [00-FF].

Example

```
<StyleAttribute name="textColor" value="blue" />
<StyleAttribute name="textColor" value="#00FF45" />
```

Fonts

A graphical application should follow the front-end platform theme. The front-end tries to determine the default font for the application screens.

- Font families on page 813
- Font sizes on page 814

- Font styles on page 815
- Font weights on page 816

Font families
Use the `fontFamily` style attribute to define a generic or specific font family.

This section describes the possible values of the `fontFamily` style attribute.

Syntax

```
font-family [,...]
```

1. *font-family* defines a generic or a native font family.

Usage

A set of generic font families is supported, that are interpreted by the front end according to the graphical capability of the paltform.

If the `fontFamily` is not a generic font family, it is interpreted as a native font family, which identifies a local font supported by the front-end. Usually, it is one of the fonts installed on the platform operating system. See front-end documentation for a list of supported native fonts.

A native font family should be used only if the front-end can't determine a proper default font family for the desired platform.

> **Important:** A font family containing white-spaces must be single quoted. In the XML definition of the style, this leads to a single quoted string that is, in turn, enclosed in double quotes:
>
> ```
> <StyleAttribute name="fontFamily" value="'Courier New'" />
> ```

When specifying a comma-separated list of font families, the front-end will use the best matching font available on the platform. You can mix generic and native font families:

```
<StyleAttribute name="fontFamily" value="'Times New Roman',Times,serif" />
```

Table 232: Generic font families to front-end platform fonts

Generic font family name	GDC	HTML5	GMA	GMI
`serif`	Times	serif (CSS)	Serif	Times New Roman
`sans-serif`	Arial	sans-serif (CSS)	Sans-Serif	Helvetica Neue
`cursive`	Comic Sans Ms	cursive (CSS)	N/A (keeps defauft font)	Marker Felt
`fantasy`	Algerian	fantasy (CSS)	N/A (keeps defauft font)	Papyrus
`monospace`	Courier New	monspace (CSS)	Monospace	Courier

> **Note:**

- The HTML5 front-end used the font family as `font-family` property in a CSS style. For more details, see CSS generic-font-families
- The GMI front-end tries to find a font family in the available fonts of the application (i.e. the iOS built-in fonts and any application specific fonts) which matches the `fontFamily` given in the styles. If none is found, the fallback is "Helvetica Neue".
- The GMA front-end maps generic font family names to Android™ generic font names (Serif, Monospace), these are then mapped to real font names. The real font name depends from the Android brand. For example Sans-serif is usually implemented with the "Roboto" font.

Example

```
<StyleAttribute name="fontFamily" value="sans-serif" />
<StyleAttribute name="fontFamily" value="'Courier New'" />
<StyleAttribute name="fontFamily" value="'Times New
 Roman',Times,serif" />
```

Font sizes

Use the `fontSize` style attribute to influence the size of a font.

Syntax

```
{ generic-size | pointspt | sizeem }
```

1. *generic-size* is one of the generic font size names (such as '`small`' or '`xx-large`) listed in Table 233: Generic font sizes on page 815.
2. *points* defines an absolute size in points. Specify a number followed immediately by `pt`, e.g., `3pt`.
3. *size* defines an relative size. Specify a number followed immediately by `em`, e.g., `3em`.

Usage

Specify either a generic font size, an absolute size in points with the "pt" unit, or a relative size with the "em" unit.

Absolute sizes (using the "pt" suffix) define a font size in physical points. Physical points are much like pixels, in that they are fixed-size units and cannot scale in size. For example, on HTML pages using CSS styles, one point is equal to 1/72 of an inch.

Relative sizes (using the "em" suffix) define a font size in a scalable size unit that adapts to the front-end platform, where one "em" unit results in the same size as the size of the default font on the platform. For example, if the size of the platform default font is 16 points, `1em = 16pt`, `2em = 32pt`, etc.

Generic font sizes are interpreted by the front end according to the graphical capability of the platform.

> **Note:** Use generic font sizes such as `medium`, `large`, `small`, or sizes relative to the user-chosen font (using `em` units), rather than absolute point values. In an HTML browser you can choose two fonts (proportional/fixed), and a well-designed document should not use more than 2 fonts. This is also valid for applications.

Table 233: Generic font sizes

Generic font size name	Definition
`xx-small`	Tiny font size
`x-small`	Extra-small font size
`small`	Small font size
`medium`	Medium font size
`large`	Large font size
`x-large`	Extra-large font size
`xx-large`	Huge font size

You can also specify an absolute font size, by giving a numeric value followed by the units such as `pt` or `em`:

Example

```
<StyleAttribute name="fontSize" value="medium" />
<StyleAttribute name="fontSize" value="xx-large" />
<StyleAttribute name="fontSize" value="12pt" />
<StyleAttribute name="fontSize" value="1em" />
```

Font styles

Use the `fontSize` style attribute to define the style of a font.

Syntax

```
{ italic | roman | oblique }
```

Usage

The style of a font can be specified with a generic name, interpreted by the front end according to the graphical capabilities of the platform. For example, on "Android™ devices, `italic` and `oblique` result in the same font aspect.

Table 234: Generic font style

Generic font style name	Definition
`italic`	Specifies an italic font style, using a typefact that slants slightly to the right. Uses a different glyph as the roman style.
`oblique`	Specifies an oblique font style. This style is similar to italic, except that it uses the same glyphs as the roman type, but distorted.
`roman`	Specifies a roman font style. This is the typical default font style in Latin-script typography.

Example

```
<StyleAttribute name="fontStyle" value="italic" />
```

Font weights

Use the `fontWeight` style attribute to define the aspect of a font.

Syntax

```
{ black
| bold
| book
| condensed
| condensedbold
| condensedlight
| demibold
| extrablack
| heavy
| light
| medium
| normal
| regular
| semibold
| thin
}
```

Usage

The availability of the weight depends on the chosen font family. For example, if the font family is defined as `AmericanTypewriter`, and the front-end platform supports the following set of font names (for this font family): AmericanTypewriter, AmericanTypewriter-Light, AmericanTypewriter-Bold, AmericanTypewriter-CondensedLight, AmericanTypewriter-CondensedBold, AmericanTypewriter-Condensed, you can only use the `condensed`, `light` and `bold` font weights.

Before using a font weight, make sure that the target platform supports the value. For example, on "Android™ devices, only `normal` and `bold` are supported.

Example

```
<StyleAttribute name="fontWeight" value="bold" />
```

Statusbar types

Possible values for Window status bar type.

This section describes how to specify a value for the `Window.statusBarType` style attribute.

Syntax

```
{ statusbar-type }
```

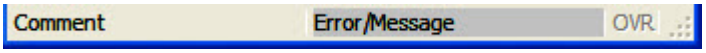
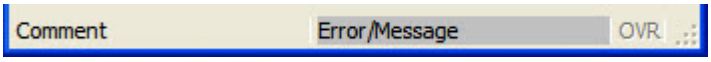
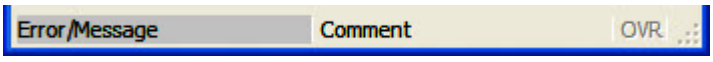
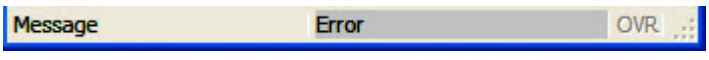
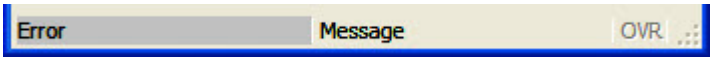
1. *statusbar-type* is a predefined status bar type name.

Usage

The `statusBarType` style attribute can get one of the values listed in the following table, to customize the rendering of error and message texts.

> **Important:** This style attribute is mainly used for desktop application using the GDC front-end.

Table 235: Possible status bar types for the statusBarType attribute

Value	Screenshot
`default`	
`lines1`	
`lines2`	
`lines3`	
`lines4`	
`lines5`	
`lines6`	
`panels1`	
`panels2`	
`panels3`	
`panels4`	
`panels5`	

Value	Screenshot
`panels6`	
`panels7`	
`none`	

Example

```
<StyleAttribute name="Windows.statusBarType" value="panels2" />
```

Style attributes reference

A presentation style attribute may be a common attribute that can be applied to any graphical element. Other presentation style attributes apply only to a specific graphical element.

- Common style attributes on page 818
- Button style attributes on page 821
- ButtonEdit style attributes on page 822
- CheckBox style attributes on page 822
- ComboBox style attributes on page 823
- DateEdit style attributes on page 824
- Default action view style attributes on page 825
- Edit style attributes on page 826
- Folder style attributes on page 827
- Grid style attributes
- Group style attributes
- HBox style attributes on page 827
- Image style attributes on page 828
- Label style attributes on page 828
- Menu style attributes on page 829
- Message style attributes on page 829
- ProgressBar style attributes on page 830
- RadioGroup style attributes on page 831
- Scrollgrid style attributes
- Table style attributes on page 831
- TextEdit style attributes on page 834
- Toolbar style attributes on page 838
- Window style attributes on page 839

Common style attributes

Common style presentation attributes apply to any graphical element, such as windows, layout containers, or form items.

For a complete list of AUI element types, refere to the `FGLDIR/src/aui.xa` definition file.

> **Important:** Common style attribute apply to basic layout elements such as containers (Group) and form widgets (Label, Button, Edit, CheckBox). According to the front-end platform, common style attributes typically do not apply to advanced graphical elements such as TopMenu or ToolBar, especially when such widget can be configured with the a user interface theme of the front-end platform. Consider using common style attribute only for elements inside the form layout.

Table 236: Common style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`backgroundColor` Defines the color to be used to fill the background of the object. For possible values, see Colors on page 807. Default is no value (default color of the object). **Note:** To set the background color of GMI/iOS specific GUI elements like toolbars, tab bars and navigation bars, use the `ios*TintColor` attributes for Windows.	Yes	Yes	Yes	Yes (see note)
`border` Defines the border for the widget. If Value is `"none"`, it removes the border. Default is no value (the widget gets its default appearance). This attribute especially applies to widgets such as Image, Edit, ButtonEdit, Button.	Yes	Yes	No	No
`fontFamily` Defines the name of the font. For possible values, see Font families on page 813. Default is no value (default object font or inherited font).	Yes	Yes	Yes	Yes
`fontSize` Defines the size of the characters. For possible values, see Font sizes on page 814. Default is no value (default object font or inherited font).	Yes	Yes	Yes	Yes
`fontStyle` Defines the style of characters. For possible values, see Font styles on page 815. Default is no value (default object font or inherited font).	Yes	Yes	Yes	Yes
`fontWeight` Defines the weight of the characters. Possible values for font weights depend from the front-end native font names, see Font weights on page 816 for details. Default is no value (default object font or inherited font).	Yes	Yes	Yes	Yes
`imageCache` For form items displaying an image, defines if the image can be cached of not by the front end.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
If value is `"yes"` the front-end can cache the image locally. By default, Image for image fields are not cached and image for form items (Button, TopMenu item, Toolbar item) are cached.				

`localAccelerators`
For form items using shortcuts, defines how the widget must behave regarding keyboard accelerators.
If value is `"yes"` (default), the local accelerators have higher priority.
Ex: "HOME" key moves the cursor to the first position.
If value is `"no"`, the application accelerators have higher priority.
Ex: "HOME" key selects the first row of the current array.
The following keys are managed "locally" if attribute defined to `"yes"`.
`TEXTEDIT`: `left, right, up, down, (control+)home, (control+)end, (control+)backspace, (control+)delete`
`EDIT`, `BUTTONEDIT`, `DATEEDIT`, etc: `left, right, home, end, (control+)backspace, (control+)delete`
`TABLE`, `TREE`: `(control+)left, (control+)right`
	Yes	No	No	No

`showAcceleratorInToolTip` Defines if the accelerator key(s) for an action should be shown in the tooltip of the corresponding action view (Button, Toolbar Item, and so on.) If value is `"yes"` the tooltip shows the accelerator key(s) after the action name, between brackets. By default, the tooltip only shows the action name.	Yes	No	No	No
`textColor` Defines the color to be used to paint the text of the object. For possible values, see Colors on page 807. Default is no value (default object color or inherited color). **Note:** In GMI, `textColor` affects the widgets they are defined on, not the labels in the form used to display the widgets. It is also used to set the tint of checkbox, radio group (horizontal) and spin edit.	Yes	Yes	Yes	Yes (see note)
`textDecoration` Defines the decoration for the text. Values can be `"overline"`, `"underline"` or `"line-through"`. Default is no value (default object font or inherited font).	Yes	Yes	No	No

Button style attributes

Button style presentation attributes apply to a button element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 237: Button style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`buttonType` Defines the rendering of a button. Values can be: • `"normal"` (default): The button is rendered as a regular push button. • `"link"`: the button is rendered as an HTML hyper-link. In contrast to the label hyper-link support, clicking on a "link" button does not start the default browser, but triggers the corresponding action, like a normal button. • `"commandLink"`: the button is rendered as a "Command Link" button on Microsoft™ Windows™ Vista and Windows™ 7.	Yes	Yes	No	No
`scaleIcon` Defines the scaling behaviors of the associated icon, if the source image size is bigger than the place reserved for it in the widget. **Note:** On GDC and GWC, if the `scaleIcon` attribute is undefined, the behavior depends on the kind of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`. Values can be: • `"no"`: No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GWC. • `"yes"`: Image are scaled down according to the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI. • `"`*nnn*`px"`: Image are scaled down according to the specified size. For example, scaleIcon="128px" will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not. Independently of the style value, the source image is never upscaled to avoid pixelization or blurring of the image. The exception is when the image come from an SVG file which can be upscaled without any penalty. If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment.	Yes	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMI
If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.				

ButtonEdit style attributes

ButtonEdit style presentation attributes apply to a buttonedit element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 238: ButtonEdit style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`scaleIcon` Defines the scaling behaviors of the associated icon, if the source image size is bigger than the place reserved for it in the widget. > **Note:** On GDC and GWC, if the `scaleIcon` attribute is undefined, the behavior depends on the kind of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`. Values can be: • `"no"`: No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GWC. • `"yes"`: Image are scaled down according to the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI. • `"`*nnn*`px"`: Image are scaled down according to the specified size. For example, scaleIcon="128px" will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not. Independently of the style value, the source image is never upscaled to avoid pixelization or blurring of the image. The exception is when the image come from an SVG file which can be upscaled without any penalty. If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment. If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.	Yes	Yes	No	No

CheckBox style attributes

CheckBox style presentation attributes apply to a checkbox element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 239: CheckBox style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`iosCheckBoxOnTintColor` On iOS devices, defines the color for the checkbox marker when on. This is different from backgroundColor, which is used for the tint of the whole switch.	N/A	N/A	N/A	Yes

ComboBox style attributes

ComboBox style presentation attributes apply to a combobox element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 240: ComboBox style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`autoSelectionStart` Defines the item from which the auto-selection will start, when pressing keys. Valid can be: • `"current"` (default): the auto-selection looks for the first corresponding item after the current item of the object. • `"first"`: the auto-selection looks for the first corresponding item after the first item of the object.	Yes	No	No	No
`comboboxCompleter` Activate the ComboBox completer mode. Possible values are `"yes"` and `"no"` (default). When this attribute is set to yes, the ComboBox will have the following behavior: • The ComboBox is editable, but only characters that match an item in the list are allowed (if the list contains the item `"aa"` and the item `"ab"`, you can type `"a"`, `"aa"`, `"ab"`, but nothing else. If you paste text in the field, it will be truncated until the rule is fulfilled. • The drop-down list will only display item which starts with the same characters as the edit field. It is dynamically updated as you type (if the list contains the item `"aa"` and the item `"ab"` and you type `"a"`, you will see both item displayed, but if you continue to type another `"a"`, you will only see `"aa"` in the list. • The best match is automatically selected when leaving the field (thus performing an `"on change"`) as soon as you hit `"TAB"` key, even if the input is not complete.	Yes	No	No	No
`completionTimeout` Defines the timeout (in milliseconds) to build the character sequence for item lookup when the user presses several keys successively. When pressing multiple keys, a character	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
sequence is build for item lookup. After the timeout delay has expired, the character sequence is reset.				

DateEdit style attributes

DateEdit style presentation attributes apply to a dateedit element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 241: DateEdit style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`buttonIcon` Defines the icon name to use for the button.	Yes	Yes	No	No
`daysOff` Defines the days of the week that are grayed out. Possible values are `"monday"`, `"tuesday"`, `"wednesday"`, `"thursday"`, `"friday"`, `"saturday"`, `"sunday"`. Default is `"saturday sunday"`. The days of week can be combined, as shown.	Yes	Yes	No	No
`firstDayOfWeek` Defines the first day of the week to be displayed in the calendar. Possible values are `"monday"`, `"tuesday"`, `"wednesday"`, `"thursday"`, `"friday"`, `"saturday"`, `"sunday"`. Default depends on the front-ends platform language settings: For example, the default first day of week will be Sunday for an English/US locale, Monday for a French or German locale.	Yes	Yes	No	No
`showCurrentMonthOnly` Defines if dates of the previous and next months are shown. Values can be `"yes"`, `"no"` (default).	Yes	Yes	No	No
`showGrid` Indicates if the grid lines between dates must be visible in the calendar. Values can be `"yes"`, `"no"` (default).	Yes	No	No	No
`showWeekNumber` Defines if the week numbers are displayed. Values can be `"yes"`, `"no"` (default).	Yes	No	No	No

Default action view style attributes

These style attributes apply to default action views (`MenuAction` and `Action` classes).

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 242: Action style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`androidActionPosition` On Android™, defines if the option corresponding to the action must be displayed in the menu bar. Values can be: • `"button"`: The action view will be displayed in the Android action bar as a button, if there is no room in the action bar, the action view is not displayed. • `"overflow"`: The action view will be displayed in the Android action bar overflow dropdown list. • `"default"`: The action view will be displayed in the Android action bar, or in the overflow dropdown, if there is no room in the action bar. **Note:** See also Default action views decoration on Android devices on page 1288.	N/A	N/A	Yes	N/A
`androidActionWithIcon` On Android, defines if the icon (default icon or icon specified with the `IMAGE` attribute) must be displayed for the action view. Values can be: • `"yes"` (default): The icon is visible (default). • `"no"`: The icon is not shown.	N/A	N/A	Yes	N/A
`androidActionWithText` On Android, defines if a label (specified with the `TEXT` attribute) must be displayed for the action view. Values can be: • `"yes"` (default): Option text is visible (if there is an icon) • `"no"`: Option text is not shown. **Note:** If the device orientation is in portrait mode, Android may not display the text, even if you force it with this attribute.	N/A	N/A	Yes	N/A
`scaleIcon` Defines the scaling behaviors of the associated icon, if the source image size is bigger than the place reserved for it in the widget. **Note:** On GDC and GWC, if the `scaleIcon` attribute is undefined, the behavior depends on the	Yes	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMI
kind of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`. Values can be: • `"no"`: No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GWC. • `"yes"`: Image are scaled down according to the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI. • `"`*nnn*`px"`: Image are scaled down according to the specified size. For example, scaleIcon="128px" will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not. Independently of the style value, the source image is never upscaled to avoid pixelization or blurring of the image. The exception is when the image come from an SVG file which can be upscaled without any penalty. If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment. If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.				

Edit style attributes

Edit style presentation attributes apply to an edit element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 243: Edit style attributes

Attribute	GDC	GWC-JS	GMA	GMI

`dataTypeHint`
 	
Defines the type of the input, to let the front-end render a field behavior suitable for the particular data type. This attribute is especially useful on mobile devices.
 	
Values can be:
• `"email"` (for email addresses)
• `"url"` (for URLs)
• `"tel"` (for telephone numbers)
• `"search"` (for search box fields)
 	
For example, on a smart phone, entering data into an edit field with `datatypeHint="tel"` makes the numeric keyboard appear.
 	
`<Style name="Edit.hintPhone">`
	No	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMI
<StyleAttribute name="dataTypeHint" value="tel"/> </Style>				
`spellCheck` Defines if the edit field includes a spelling checker. **Note:** • With GDC, the possible values are the two dictionary files needed for each language (one .aff and one .dic). These files can be downloaded here. Only the files available for OpenOffice.org 2.x are working (files for OpenOffice.org 3.x are not supported yet). Specify in the style the two files for the "spellCheck" StyleAttribute, using one of the file formats. The local directory of dictionary files can be asked to the GDC with the `standard.feInfo` frontcall with the `dictionariesDirectory` parameter. • With GWC-JS, the attribute is not applicable: Edit fields use the web browser spellchecker. • With GMI, available values are `"yes"`, `"no"`. If this attribute is not set, iOS will decide if spellchecking is enabled, depending on the global auto-correction setting on the device.	No (see note)	No (see note)	No	Yes (see note)

HBox style attributes
HBox style presentation attributes apply to an HBox element.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 244: HBox style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`splitViewRendering` Indicates if the HBox must be displayed as a splitview. • in landscape mode, panes are side by side and scroll independently • in portrait mode, user navigates between the panes by swiping left or right Values can be `"yes"`, `"no"` (default is no)	No	No	Yes	No

Folder style attributes
Folder style presentation attributes apply to a folder tab element.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 245: Folder style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`position` Defines the position of the folder tabs. Values can be `"top"` (default), `"left"`, `"right"`, `"bottom"`.	Yes	Yes	No	No

Image style attributes

Image style presentation attributes apply to an image element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 246: Image style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`alignment` Defines the image alignment when the container is bigger than the image itself. Possible values are a pair of horizontal (`"left"`, `"horizontalCenter"`, `"right"`) and vertical alignments (`"top"`, `"verticalCenter"`, `"bottom"`). To combine alignment options, use a space as separator. Value can also be `"center"`, which is equivalent to `"horizontalCenter verticalCenter"`. The default value is `"top left"`.	Yes	Yes	No	No
`imageContainerType` **Important:** This attribute is deprecated. Consider using URL-based Web Components instead of `IMAGE` fields with the `imageContainerType` style attribute: URL Web Components are much easier to use and more powerful. When set to `"browser"`, defines an image container as a browser. To use the image field as a browser, set a URL instead of an image name. **Note:** This feature uses the WebKit Open Source project as provided with Qt, and has limitations such as no Java™ or ActiveX support. It will display HTML / rich text, but may encounter difficulties with more complex Web pages.	Yes	No	No	No

Label style attributes

Label style presentation attributes apply to a label.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 247: Label style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`textFormat` Defines the rendering of the content of the label widget. Possible values are: • `"plain"` (default): the value assigned to this widget is interpreted as plain text. • `"html"`: it is interpreted as HTML (with hyperlinks).	Yes	Yes	No	No

Menu style attributes

Menu style presentation attributes apply to a menu element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

This table shows the presentation attributes for Menu:

Table 248: Menu style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`position` Defines the position of the automatic menu for "popup" menus. Values can be: • `"cursor"` (default): the popup menu appears at the cursor position. • `"field"`, the popup menu appears below the current field. • `"center"`, the popup menu appears at the center of the screen. • `"center2"`, the popup menu appears at the center of the current window.	Yes	No	No	No

Message style attributes

Message style presentation attributes apply to an error or message.

The element type for both ERROR and MESSAGE is `Message`. To distinguish `ERROR` from `MESSAGE`, the `":error"` or `":message"` pseudo-selectors can be used to specify a different style for the rendereing of each instruction: `"Message:error"` corresponds to the `ERROR` instruction, and `"Message:message"` corresponds to the `MESSAGE` instruction.

The `ERROR` and `MESSAGE` instructions can get a `STYLE` attribute in the `ATTRIBUTES` clause, to specify a particular style name:

```
MESSAGE "No rows have been found." ATTRIBUTES(STYLE="info")
```

A limited set of common style attributes are supported for error/message display. In addition to the attributes described in the section, you can only define font style attributes for messages.

Like simple form fields, TTY attributes have a higher priority than style attributes. By default, `ERROR` has the TTY attribute `REVERSE`, which explains why `ERROR` messages have a reverse background, even

when you use a backgroundColor style attribute. Use the NORMAL attribute in ERROR, to avoid the default REVERSE TTY attribute and define your own background color with a style.

Consider centralizing your ERROR and MESSAGE instruction calls in a function, to simplify global modifications:

```
FUNCTION my_error(m, s)
   DEFINE m, s STRING
   IF s IS NULL THEN
       ERROR m ATTRIBUTES(NORMAL)
   ELSE
       ERROR m ATTRIBUTES(NORMAL, STYLE=s)
   END IF
END FUNCTION
```

This table shows the presentation attributes for ERROR and MESSAGE instructions:

Table 249: Presentation attributes for ERROR and MESSAGE instructions

Attribute	GDC	GWC-JS	GMA	GMI
`position` Defines the output type of the status bar message field. Values can be: • `"statusbar"` (default): will display the text in the regular statusbar of the window. • `"popup"`: will bring a window popup to the front; it should be used with care, since it can annoy the user. • `"statustip"`: will add a small "down" arrow button that will show the popup once the user clicks on it. This can be useful to display very long text. • `"both"`: will display the text in a popup window and then in the status bar.	Yes	No	No	No
`textFormat` Defines the rendering of the content of the widget. Possible values are: • `"plain"` (default): the value assigned to this widget is interpreted as plain text. • `"html"`, it is interpreted as HTML (with hyper-links).	Yes	No	No	No

ProgressBar style attributes
ProgressBar style presentation attributes apply to a progressbar element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 250: ProgressBar style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`percentageVisible` Defines whether the current progress value is displayed. Possible values are:	Yes	Yes (see note)	No	No

Attribute	GDC	GWC-JS	GMA	GMI
• `"no"` (default): no progress value is displayed. • `"center"`: the progress will be displayed in the middle of the progressbar. • `"system"`: it will follow the system theme. **Note:** GWC-JS: This attribute is only supported if the browser allows this option in the progressbar widget.				

RadioGroup style attributes

RadioGroup style presentation attributes apply to a radiogroup element.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 251: RadioGroup style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`autoSelectionStart` Defines the item from which the auto-selection will start, when pressing keys. Possible values are: • `"current"` (default): it will look for the first corresponding item after the current item of the object. • `"first"`, the auto-selection will look for the first corresponding item after the first item of the object.	Yes	No	No	No
`completionTimeout` Defines the timeout (in milliseconds) to build the character sequence for item lookup when the user presses several keys successively. When pressing multiple keys, a character sequence is build for item lookup. After the timeout delay has expired, the character sequence is reset.	Yes	No	No	No

Table style attributes

Table style presentation attributes apply to a table element.

Note: This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 252: Table style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`forceDefaultSettings` By default, tables are reopened with column positions, visibility and sizes they had when the window was closed. By setting this attribute to true, the saved settings are ignored and the table gets the initial column layout. Note that the saved settings include also the sort columns, that will impact on the order of the rows in the table.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
Values can be `"yes"`,`"no"` (default).				
`headerAlignment` Defines the column header alignment in a table. Values can be: • `"default"` (default): will use the system default. In most case it is left aligned. • `"left"` will force all column headers to be left aligned. • `"center"` will force all column headers to be centered. • `"right"` will force all column headers to be right aligned. • `"auto"` will first try to align each column header according to the `"justify"` attribute of the column. If no `"justify"` attribute is set, the column header will be aligned according to the type of data: right for numeric data, left for text data.	Yes	Yes	No	No
`headerHidden` Defines if the horizontal header must be visible in a table. Values can be `"yes"`,`"no"` (default).	Yes	Yes	No	No
`highlightColor` Defines the highlight color of rows for the table, used for selected rows. For possible values, see Colors.	Yes	Yes	No	No
`highlightCurrentCell` Indicates if the current cell must be highlighted in a table. Values can be `"yes"`,`"no"` (default). By default the current edit cell in table has a white background. You can change this behavior by setting this attribute to `"yes"`, to use the same color as when highlightCurrentRow is used. Only some type of cells, checkboxes for example, can be highlighted. Normal editor cells stay in white, because this is the editor background color.	Yes	Yes	No	No
`highlightCurrentRow` Indicates if the current row must be highlighted in a table during an `INPUT ARRAY`. Values can be `"yes"`,`"no"` (default). By default, when a table is in read-only mode (`DISPLAY ARRAY`), the front-end automatically highlights the current row. But in editable mode (`INPUT ARRAY`), no row highlighting is done by default. You can change this behavior by setting this attribute to `"yes"`.	Yes	Yes	No	No
`highlightTextColor`	Yes	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMI
Defines the highlighted text color of rows for the table, used for selected rows. For possible values, see Colors.				
`leftFrozenColumns` Requires `"tableType"` set to `"frozenTable"`. Defines how many columns are frozen, starting from the left of the Table. Values can be any numeric value matching with the number of columns. Default is `"0"`.	Yes	Yes	No	No
`resizeFillsEmptySpace` Defines if the resize of the table adapts the size of the last column to avoid unused space. Values can be `"yes"`,`"no"` (default).	Yes	No	No	No
`rightFrozenColumns` Requires `"tableType"` set to `"frozenTable"`. Defines how many columns are frozen, starting from the right of the Table. Values can be any numeric value matching with the number of columns. Default is `"0"`.	Yes	Yes	No	No
`showGrid` Indicates if the grid lines must be visible in a table. Values can be `"yes"` (default when `INPUT ARRAY`),`"no"` (default when `DISPLAY ARRAY`). By default, when a Table is in editable mode (`INPUT ARRAY`), the front-end displays grid lines in the table. You can change this behavior by setting this attribute to `"no"`. By default, when a Table is in editable mode (`DISPLAY ARRAY`), the front-end does not display grid lines in the table. You can change this behavior by setting this attribute to `"yes"`.	Yes	Yes	No	No
`summaryLineAlwaysAtBottom` Defines the placement of the summary row containing aggregate fields. When set to "yes", the row containing aggregate fields is rendered in the last line of the table. When set to "no", the row containing aggregate fields is rendered immediately after the values being aggregated. This is the default.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
`tableType` Defines the rendering type of the table. Values can be: • `"normal"` (default): Regular table rendering. • `"pictureFlow"`: The first column of the table will be used to define the list of images to be used in the picture flow. • `"frozenTable"`: Users can "freeze" some columns from scrolling, so that they always remain visible. Default frozen columns can be defined with `"leftFrozenColumns"` and `"rightFrozenColumns"` attributes.	Yes	Yes	No	No

TextEdit style attributes
Textedit style presentation attributes apply to a textedit element.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 253: TextEdit style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`customWidget` Defines a specific widget to be used by the front end for the textedit field. Values can be: • `"TinyMCE"`: Uses a specific HTML text editor (for HTML5, uses the TinyMCE™ HTML editor when set).	No	Yes	No	No
`integratedSearch` Defines if the textedit field allows search facility (Control-F). Values can be `"yes"`, `"no"` (default).	Yes	No	No	No
`showEditToolBox` Defines if the toolbox for the rich text editing should be shown. Possible values are `"auto"`(default), `"yes"`, `"no"`. Only available if `textFormat` style attribute is set to `"html"`.	Yes	Yes	No	No
`spellCheck` Defines if the textedit field includes a spelling checker. **Note:** • With GDC, the possible values are the two dictionary files needed for each language (one .aff and one .dic). These files can be downloaded here. Only the files available	Yes (see note)	No (see note)	No	Yes (see note)

Attribute	GDC	GWC-JS	GMA	GMI
for OpenOffice.org 2.x are working (files for OpenOffice.org 3.x are not supported yet). Specify in the style the two files for the "spellCheck" StyleAttribute, using one of the file formats. The local directory of dictionary files can be asked to the GDC with the `standard.feInfo` frontcall with the `dictionariesDirectory` parameter. • With GWC-JS, the attribute is not applicable: Edit fields use the web browser spellchecker. • With GMI, available values are `"yes"`, `"no"`. If this attribute is not set, iOS will decide if spellchecking is enabled, depending on the global auto-correction setting on the device.				
`textFormat` Defines the rendering of the content of the widget. Values can be: • `"plain"` (default): the value assigned to this widget is interpreted as plain text. • `"html"`, the value is interpreted as HTML (with hyper-links), with rich text input feature enabled. Note that a specific HTML editor widget can be specified with the `customWidget` style attribute.	Yes	Yes	No	No
`wrapPolicy` Defines where the text can be wrapped in word wrap mode. Values can be: • `"atWordBoundary"` (default): the text will wrap at word boundaries. • `"anywhere"`: the text breaks anywhere, splitting words if needed.	Yes	Yes	No	No

File Formats for `spellCheck`:

- `"my_affix_file.aff|my_dictionnary_file.dic"`
- an absolute path such as `"file:///c:/dics/my_dictionnary_file.aff|file:///c:/dics/my_dictionnary_file.dic"`
- a Web server path such as `http://mywebserver.com/my_affix_file.aff|http://mywebserver.com/my_dictionnary_file.dic`

Rich Text Editing

Some Genero clients support a rich text editing interface, which can display a toolbox with classic editing actions (bold, italic, font size, and so on). Local actions are also created.

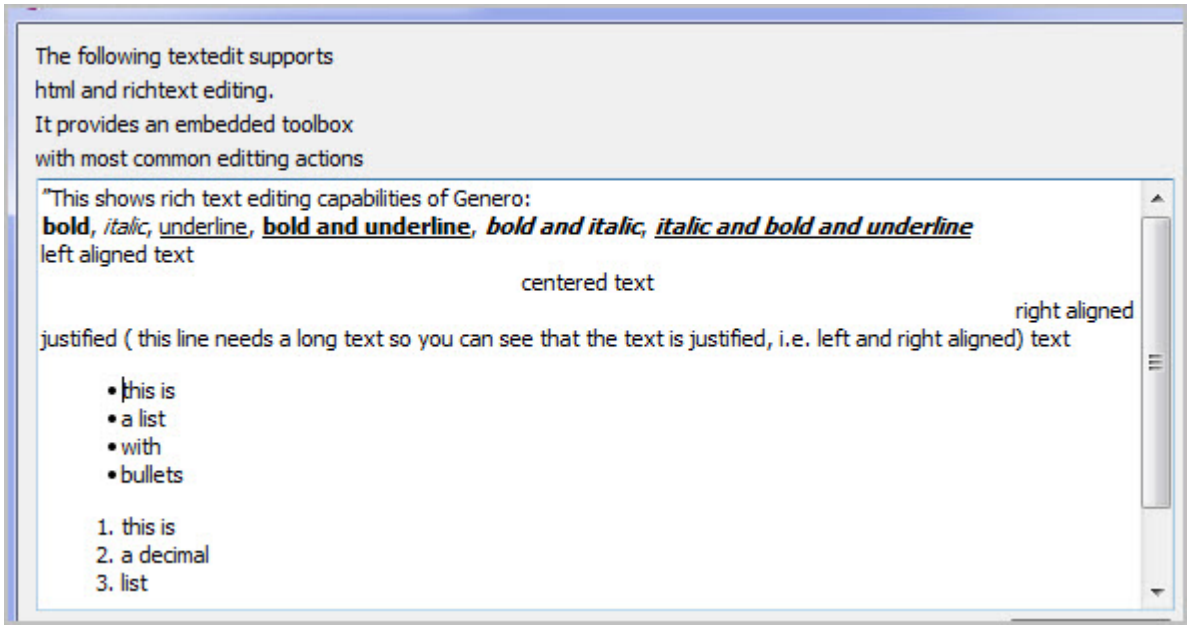


Figure 43: Rich text editing interface

Rich text editing provides:

- Text format: bold, italic, underline
- Paragraph alignment: left, center, right, justify
- Lists: bullet, decimal
- Paragraph indentation
- Font size

To enable rich text editing, set the `textFormat` styleAttribute to `html` .

```
<Style name="TextEdit.richText">
  <StyleAttribute name="textFormat" value="html" />
</Style>
```

If you are using the Genero Web Client for HTML5, you can specify the TinyMCE™ editor for rich text editing with the `customWidget` style attribute. If the `customWidget` attribute is not specified, the default editor is used.

```
<Style name="TextEdit.richText">
  <StyleAttribute name="textFormat" value="html" />
  <StyleAttribute name="customWidget" value="TinyMCE" />
</Style>
```

Richtext toolbox

By default, when the mouse reaches the top border of the textedit field where rich text editing has been enabled, a toolbox appears. The toolbox disappears when the mouse leaves the top border area. This implementation is useful if you only use the textedit field to display rich text, as the toolbox is only visible in input.

If you want always display the toolbox, you can set the `showEditToolBox` styleAttribute.

```
<Style name="TextEdit.richText">
```

```
  <StyleAttribute name="textFormat" value="html" />
  <StyleAttribute name="showEditToolBox" value="yes" />
</Style>
```

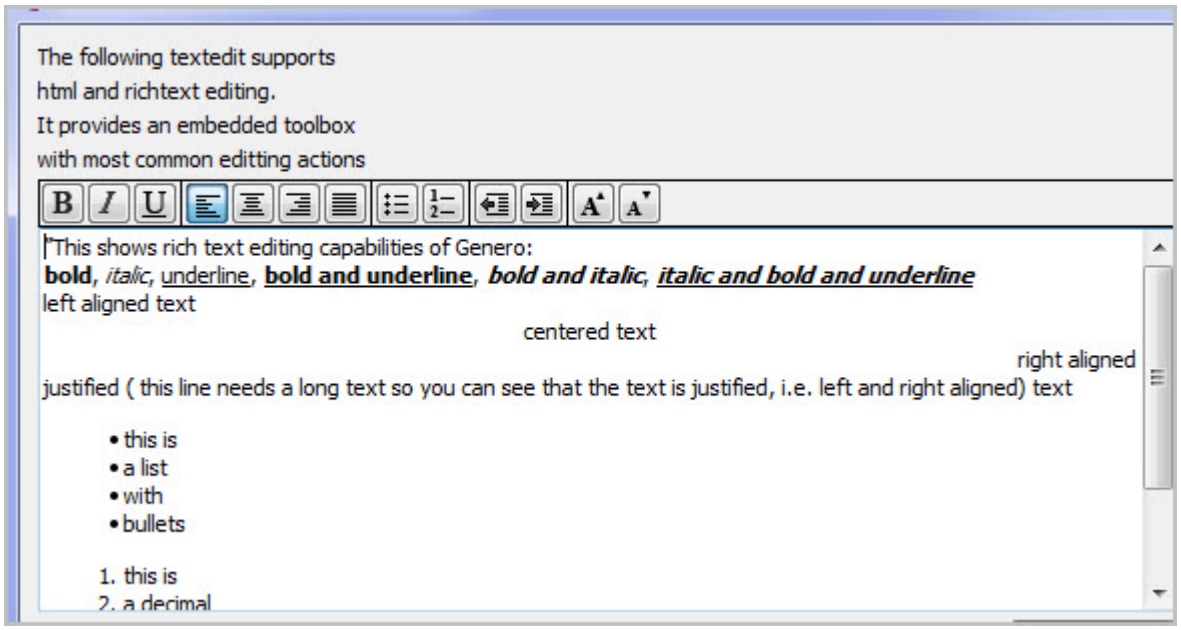


Figure 44: Rich text editing interface with toolbox always displayed.

> **Tip:** The textedit will be wide enough to display the toolbox in its entirety, even if you define a small width in your form definition file. Take this in account when designing your form.
>
> **Tip:** The textedit will be high enough to display the number of lines defined in the form definition file (using the textedit font) and the toolbox when required. A textedit with a height of 1 will display the toolbox and one line, which is much higher than without the toolbox.
>
> **Important:** The behavior of the attribute `showEditToolBox` with the value `auto` differs between the Genero Desktop Client and the Genero Web Client. With the Genero Desktop client, '`auto`' is interpreted as '`no`'. With the Genero Web Client, '`auto`' is interpreted as '`yes`'.

Rich text local actions

Local actions have been created for each rich text capability. As with any local action, you can configure accelerator keys, or you can bind them to action views like toolbar buttons.

Table 254: Local action names, accelerators, and icons

Name	Default Accelerator	Icon Name	Icon
`richtextbold`	Ctrl-b	textbold	B
`richtextitalic`	Ctrl-i	textitalic	*I*
`richunderline`	Ctrl-u	textunder	U
`richtextalignleft`	Ctrl-l	textleft	
`richtextaligncenter`	Ctrl-e	textcenter	

Name	Default Accelerator	Icon Name	Icon
`richtextalignright`	Ctrl-r	textright	
`richtextalignjustify`	Ctrl-j	textjustify	
`richtextlistbullet`	None	textlistbullet	
`richtextlistdecimal`	None	textlistnumbered	
`richtextdecreaseident`	None	textindentdecrease	
`richtextincreaseident`	None	textindentincrease	
`richtextdecreasefontsize`	None	textfontsizedown	
`richtextincreasefontsize`	None	textfontsizeup	

You can hide the toolbox using the `showEditToolBox` styleAttribute.

```
<StyleAttribute name="textFormat" value="html" />
<StyleAttribute name="showEditToolBox" value="no" />
```

Toolbar style attributes
Toolbar style presentation attributes apply to a toolbar.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 255: ToolBar style attributes

Attribute	GDC	GWC-JS	GMA	GMi
`iosSeparatorStretch` Stretches the SEPARATORs between toolbar items on iOS devices. When this attribute is set to `yes`, separators are acting like springs between the individual toolbar items. Values can be: • `"no"` (default): do not stretch toolbar item separators. • `"yes"`: stretch toolbar item separators.	N/A	N/A	N/A	Yes
`scaleIcon` Defines the scaling behaviors of the associated icon, if the source image size is bigger than the place reserved for it in the widget. **Note:** On GDC and GWC, if the `scaleIcon` attribute is undefined, the behavior depends on the kind of action view: toolbar button icons and action panel button icons are scaled down to match the size of the widget. For other widgets, by default no scaling occurs, as for `scaleIcon="no"`. Values can be:	Yes	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMi
• `"no"`: No scaling occurs and the image is taken as-is. It is up to the developer to resize the source image to avoid misalignment. This is the default on GDC/GWC. • `"yes"`: Image are scaled down according to the height of the widget (button or edit field). Setting a big font can result in a big icon. This is the default on GMA/GMI. • `"`*nnn*`px"`: Image are scaled down according to the specified size. For example, scaleIcon="128px" will make every icon a maximum of 128*128 pixels. At least one side equal to 128 pixels, depending if the source image is square or not. Independently of the style value, the source image is never upscaled to avoid pixelization or blurring of the image. The exception is when the image come from an SVG file which can be upscaled without any penalty. If the icon must be enlarged, the image is centered and a transparent border is added to "fill" the empty space. This allows a mix of larger and smaller icons while keeping widget alignment. If scaling takes place, the aspect ratio of the original image is kept. A non-square source image displays as a non-square scaled icon.				
`toolBarTextPosition` Defines the text position of a ToolBarItem. Values can be: • `"textBesideIcon"` • `"textUnderIcon"` (default)	Yes	Yes	No	No

Window style attributes
Window style presentation attributes apply to a window.

> **Note:** This topic lists presentation style attributes for a specific class of form element, common presentation style attributes can also be used for this type of element.

Table 256: Window style attributes

Attribute	GDC	GWC-JS	GMA	GMI
`actionPanelButtonSize` Defines the width of buttons. Values can be `"normal"`, `"shrink"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. When using `"normal"` and `"shrink"`, buttons are sized according to the text or image, where `"shrink"` uses the minimum size needed to display the content of the button. Default is `"normal"`.	Yes	No	No	No
`actionPanelButtonSpace` Defines the space between buttons.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
Values can be "none", "tiny", "small", "medium", "large" or "huge". Default is "medium".				
actionPanelButtonTextAlign Defines the text alignment inside buttons. Values can be "left", "center", "right". Default is "left" when the button have an icon, "center" otherwise.	Yes	Yes	No	No
actionPanelButtonTextHidden Defines the text visibility inside buttons. Values can be "yes" or "no". Default is "yes".	Yes	Yes	No	No
actionPanelDecoration Defines the decoration of the action panel. Values can be "auto", "yes", "no" and "dockable". Default is "auto".	Yes	No	No	No
actionPanelHAlign Defines the alignment of the action panel when actionPanelPosition is "top" or "bottom". Values can be "left", "right" or "center". Default is "left".	Yes	No	No	No
actionPanelPosition Defines the position of the action button frame (OK/Cancel). Values can be "none", "top", "left", "bottom" or "right". Default is "right".	Yes	Yes	No	No
actionPanelScroll Defines if the action panel is "ring" - that is, when the last button is shown, pressing on the "down" button will show the first one again. Values can be "0" or "1". Default is "1".	Yes	No	No	No
actionPanelScrollStep Defines how the action panel should scroll when clicking the "down" button, to shown the next visible buttons. Values can be: • "line" (default): the panel will scroll by one line, and then show only the next button.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
• `"page"`: the scrolling will be done page by page.				
`allowedOrientations` Defines possible orientations for modile device. Values can be: • `"all"`: Any orientations are allowed. • `"landscape"`: Landscape orientation (the display is wider than it is tall). • `"portrait"`: Portrait orientation (the display is taller than it is wide). • `"landscape_reverse"`: Landscape orientation in the opposite direction from normal landscape. • `"portrait_reverse"`: Portrait orientation in the opposite direction from normal portrait. • `"landscape_all"`: Normal and reverse landscape orientations are allowed. • `"portrait_all"`: Normal and reverse portrait orientation are allowed. Default is `"all"`. **Note:** This attribute is supported at the Window level only by GMA.	N/A	N/A	Yes	No
`border` Defines the border type of the window. Values can be: • `"normal"` (default): the border is standard, with a normal window header with a caption. • `"frame"`: only a frame appears, typically without a window header. • `"tool"`: a small window header is used. • `"none"`: the window gets no border. On Mac platforms, using `"tool"` is not effective.	Yes	No	No	No
`commentPosition` Defines the rendering for field comments. Values can be: • `"statusbar"` (default): displays the comment in the window status bar. • `"popup"` will bring a window popup to the front; to be used with care, as it can annoy the end user. • `"statustip"` will add a small "down" arrow button that will show the popup once the user clicks on it; useful to display very long text. • `"both"` will display the comment text in a popup window and then in the status bar.	Yes	No	No	No
`defaultTTFColor`	Yes	Yes	Yes	Yes

Attribute	GDC	GWC-JS	GMA	GMI
Defines the default color to be used for TTF icons. All icons displayed in the window using this style will by default get the color specified in the defaultTTFColor attribute. The value for this attribute must and RGB specification or a named color as listed in Colors on page 807. For more details about TTF icon usage see Using a simple image name (centralized icons) on page 785.				
`errorMessagePosition` Defines the rendering of program errors displayed with the ERROR instruction. Values can be: • `"statusbar"` (default): displays the comment in the window status bar. • `"popup"` will bring a window popup to the front; to be used with care, as it can annoy the end user. • `"statustip"` will add a small "down" arrow button that will show the popup once the user clicks on it; useful to display very long text. • `"both"` will display the comment text in a popup window and then in the status bar.	Yes	No	No	No
`forceDefaultSettings` Indicates if the window content must be initialized with the saved positions and sizes. By default, windows are reopened at the position and with the size they had when they were closed. You can force the use of the initial settings with this attribute. This applies also to column position and width in tables. Values can be `"yes"` or `"no"`. Default is `"no"`.	Yes	Yes	No	No
`formScroll` Defines if scrollbars should always be displayed when the form is bigger than the screen, or only when the window is maximized. Values can be `"yes"` or `"no"`. Default is `"yes"`.	Yes	No	No	No
`ignoreMinimizeSetting` Defines if the stored settings "state=minimize" must be ignored when loading settings. To be used when minimized windows should not be shown minimized when reopened. Values can be `"yes"` or `"no"`.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
Default is `"no"`.				
`iosTintColor` On iOS devices, defines the color for items used in the navigation bar, toolbar, and some items in the forms (Buttons, SpinEdit, Radiogroups, and row checkmark and disclosure indicators in list views). This style attribute does not apply to `MENU` with `STYLE=dialog/popup`.	N/A	N/A	N/A	Yes
`iosNavigationBarTextColor` On iOS devices, defines the text color of the navigation bar.	N/A	N/A	N/A	Yes
`iosNavigationBarTintColor` On iOS devices, defines the background color of the navigation bar.	N/A	N/A	N/A	Yes
`iosToolBarTintColor` On iOS devices, defines background color of the toolbar.	N/A	N/A	N/A	Yes
`iosTabBarTintColor` On iOS devices, defines the background color of the tab bar. The iOS tab bar is created with a `TYPE=NAVIGATOR` window.	N/A	N/A	N/A	Yes

`materialFABActionList`
Defines a comma-separated list of action names that will be bound to the Floating Action Button (FAB button), on a device following the material design guidelines. To be used in conjunction with the `materialFABType` attribute. The order of the actions will define which action is triggered when the FAB button is tapped, and several matching actions are active.
The default list of actions is:
`"new,append,insert,update,edit"`
	No	No	Yes	No

`materialFABType` Controls the Floating Action Button (FAB button), on a device following the material design guidelines. Possible values are: • `"single"` (default) - the FAB button is shown and maps to the first active action defined in the `materialFABActionList` attribute. • `"none"` - no FAB button must be displayed.	No	No	Yes	No
`menuPopupPosition` Defines the position of the automatic menu for "popup" menus. Values can be:	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
• `"cursor"` (default) - the popup menu appears at the cursor position. • `"field"` - the popup menu appears below the current field. • `"center"` - the popup menu appears at the center of the screen. • `"center2"` - the popup menu appears at the center of the current window.				
`messagePosition` Defines the rendering for program messages displayed with the `MESSAGE` instruction. Values can be: • `"statusbar"` (default): displays the comment in the window status bar. • `"popup"` will bring a window popup to the front; to be used with care, as it can annoy the end user. • `"statustip"` will add a small "down" arrow button that will show the popup once the user clicks on it; useful to display very long text. • `"both"` will display the comment text in a popup window and then in the status bar.	Yes	No	No	No
`position` Indicates the initial position of the window. Values can be: • `"default"` (default): the windows are displayed according to the window manager rules. • `"field"`: the window is displayed below the current field (works as `"default"`, when current field does not exist). • `"previous"` the window is displayed at the same position (top left corner) as the previous window. (works as `"default"` when there is no previous window). • `"center"`: the window is displayed in the center of the screen. • `"center2"`: the window is displayed in the center of the current window. For front-ends using stored settings, `"field"`, `"previous"` and `"previous"` have higher priority than the settings.	Yes	No	No	No
`ringMenuButtonSize` Defines the width of buttons. Values can be `"normal"`, `"shrink"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. When using `"normal"` and `"shrink"`, buttons are sized according to the text or image, where `"shrink"` uses the minimum size needed to display the content of the button.	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
Default is `"normal"`.				
`ringMenuButtonSpace` Defines the space between buttons. Values can be `"none"`, `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. Default is `"medium"`.	Yes	No	No	No
`ringMenuButtonTextAlign` Defines the text alignment inside buttons. Values can be `"left"`, `"center"`, `"right"` Default is `"left"` when the button have an icon, `"center"` otherwise.	Yes	Yes	No	No
`ringMenuButtonTextHidden` Defines the text visibility inside buttons. Values can be "`yes`" or "`no`". Default is "`yes`".	Yes	Yes	No	No
`ringMenuDecoration` Defines the decoration of the menu panel. Values can be `"auto"`, `"yes"`, `"no"` and `"dockable"`. Default is `"auto"`.	Yes	No	No	No
`ringMenuHAlign` Defines the alignment of the ring menu when `ringMenuPosition` is `"top"` or `"bottom"`. Values can be `"left"`, `"right"` or `"center"`. Default is `"left"`.	Yes	No	No	No
`ringMenuPosition` Defines the position of the ring menu frame for a MENU instruction. Values can be `"none"`, `"top"`, `"left"`, `"bottom"` or `"right"`. Default is `"right"`.	Yes	Yes	No	No
`ringMenuScroll` Defines if the focus can wrap in the ring menu default actions when pressing up or down keys. Values can be `"0"` or `"1"`. Default is `"1"`.	Yes	No	No	No
`ringMenuScrollStep`	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
Defines how the ring menu must scroll when moving to a next button that is not visible. Values can be: • `"line"` (default): the menu will scroll by one line, and show only the next button. • `"page"`, the scrolling will be done page by page.				
`sizable` Defines if the window can be resized by the user. Values can be `"yes"`, `"no"` or `"auto"`. With the GDC, when using `"auto"`, the window becomes resizeable if the content of the first displayed form has resizeable elements, for example when using a form with a TABLE container or an TEXTEDIT with STRETCH attribute. With GWC, the behavior is applied to the form instead of the window. When set to `"no"`, the form content is not stretched even if the form contains stretchable items. **Note:** On Linux™ and Mac platforms, most of window managers don't take into account `sizable` when it is set to `"no"`. Default is `"yes"`.	Yes	Yes	No	No
`startMenuAccelerator` Defines the shortcut keys to execute the selected start menu item, when the position is defined as `"tree"` or `"poptree"`. By default, `"space"`, `"enter"` and `"return"` start the application linked to the current item.	Yes	No	No	No
`startMenuExecShortcut2` Defines the shortcut keys to execute the selected start menu item, when the position is defined as `"tree"` or `"poptree"`. By default, `"space"`, `"enter"` and `"return"` start the application linked to the current item.	Yes	No	No	No
`startMenuPosition` Indicates the position of the start menu, when one is defined. Values can be: • `"none"` (default): the startmenu is not displayed. • `"tree"`: the start menu is displayed as a treeview, always visible on the right side of the window. • `"menu"`: the start menu is displayed as a pull-down menu, always visible at the top of the window. • `"poptree"`: the start menu is displayed as a tree view in a popup window that can be opened with a shortcut (see `startMenuShortcut`).	Yes	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMI
`startMenuShortcut` Defines the shortcut key to open a start menu, when the position is defined as `"poptree"`. Default is `"control-shift-F12"`.	Yes	No	No	No
`startMenuSize` Defines the size of the start menu, when one is defined and the position is defined as `"tree"` or `"poptree"`. Values can be `"tiny"`, `"small"`, `"medium"`, `"large"` or `"huge"`. Default is `"medium"`. **Note:** The size will also depend on the font used for the startmenu.	Yes	No	No	No
`statusBarType` Defines the type of status bar the window will display. See Statusbar types on page 817 for all possible values. Default is `"default"`.	Yes	No	No	No
`tabbedContainer` For the GDC, defines if the WCI container must display the child application windows in a folder tab. For the GWC, this attribute specifies whether child applications are to be displayed inside the same browser window as the parent, or in a new window. WCI is not supported by GWC. Values can be `"yes"` or `"no"`. Default is `"no"`.	Yes	Yes	No	No
`tabbedContainerCloseMethod` Defines the folder tab method of the container when `tabbedContainer` is set to `"yes"`. Values can be: • `"container"` (default): container gets a close button in the tab. • `"page"`: each page has its own close button. • `"both"`: each page and the container has its close button. • `"none"`: no close button is shown.	Yes	No	No	No
`thinScrollbarDisplayTime` Defines the display time (in seconds) of the automatic scrollbar displayed when scrolling on fixed screen array (a.k.a. "Matrix") and SCROLLGRIDs (for some front-ends). After the delay, the scrollbar will disappear. A value of zero specifies an infinite time: The thin scrollbar remains	Yes	Yes	No	No

Attribute	GDC	GWC-JS	GMA	GMI
visible while the record list can be scrolled (i.e. during dialog execution). Default is 1 second.				
`toolBarDocking` Defines if the toolbar is movable and floatable. Values can be `"yes"` or `"no"`. Default is `"yes"`.	Yes	No	No	No
`toolBarPosition` Indicates the position of the toolbar, when a toolbar is defined. Values can be `"none"`, `"top"`, `"left"`, `"bottom"` or `"right"`. Default is `"top"`.	Yes	No	No	No
`windowMenu` Defines if the WCI container should display an automatic "Window" menu, with *Cascade* and *Tile* features, and list of child windows. Values can be `"yes"` or `"no"`. Default is `"no"`.	Yes	No	No	No
`windowOptionClose` Defines if the window can be closed with a system menu option or window header button. Values can be `"yes"`, `"no"` or `"auto"`. Default is `"auto"`. When value is `"auto"`, the option is enabled according to the window type. This attribute may have different behavior depending on the front end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.	Yes	Yes	No	No
`windowOptionMaximize` Defines if the window can be maximized with a system menu option or window header button. Values can be `"yes"`, `"no"` or `"auto"`. Default is `"auto"`. When value is `"auto"`, the option is enabled according to the window type. This attribute may have different behavior depending on the front end operating system. For example, when no system	Yes	No	No	No

Attribute	GDC	GWC-JS	GMA	GMI
menu is used, it may not be possible to have this option enabled.				
`windowOptionMinimize` Defines if the window can be minimized with a system menu option or window header button. Values can be `"yes"`, `"no"` or `"auto"`. Default is `"auto"`. When value is `"auto"`, the option is enabled according to the window type. This attribute may have different behavior depending on the front end operating system. For example, when no system menu is used, it may not be possible to have this option enabled.	Yes	No	No	No
`windowState` Defines the initial state of a window. Values can be `"normal"`, `"maximized"` or `"minimized"`. Default is `"normal"`.	Yes	No	No	No
`windowSystemMenu` Defines if the window shows a system menu. Values can be `"yes"`, `"no"` or `"auto"`. Default is `"auto"`. When value is `"auto"`, the system menu is enabled according to the window type. **Note:** HTML5: Only `"auto"` is supported.	Yes	No (see note)	No	No
`windowType` Defines the basic type of the window. Values can be: • `"normal"` (default): Normal windows are displayed as typical application windows. • `"modal"`: Modal windows are displayed at the top of all other windows, typically used for temporary dialogs.	Yes	Yes	No	No

Examples

Code examples that use style file entries, form definition files, and source code to illustrate how presentation styles are applied.

Example 1: Defining styles for grid elements

This example shows how to define styles for grid elements.

The presentation style definition file:

```
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <!-- Applies to all type of elements -->
```

```
  <Style name=".bigfont">
     <StyleAttribute name="fontSize" value="large" />
  </Style>
  <!-- Default text color and font family for all labels -->
  <Style name="Label">
     <StyleAttribute name="textColor" value="blue" />
     <StyleAttribute name="fontFamily" value="sans-serif" />
  </Style>
  <!-- Background color for Edits having focus -->
  <Style name="Edit:focus">
     <StyleAttribute name="backgroundColor" value="yellow" />
  </Style>
  <!-- Text color for Edits with STYLE="mandatory" -->
  <Style name="Edit.mandatory">
     <StyleAttribute name="textColor" value="red" />
  </Style>
</StyleList>
```

The form definition file:

```
LAYOUT
GRID
{
[l1    ][f1            ]
[l2    ][f2            ]
[l3    ][f3            ]
}
END
ATTRIBUTES
LABEL l1: TEXT="Label 1:";
EDIT  f1 = FORMONLY.field1;
LABEL l2: TEXT="Label 2:";
EDIT  f2 = FORMONLY.field2;
LABEL l3: TEXT="Label 3:", STYLE="bigfont";
EDIT  f3 = FORMONLY.field3, STYLE="bigfont mandatory";
END
```

Program source file:

```
MAIN
    DEFINE rec RECORD
                field1 STRING,
                field2 STRING,
                field3 STRING
           END RECORD

    LET rec.field1 = "Field 1"
    LET rec.field2 = "Field 2"
    LET rec.field3 = "Field 3"

    CALL ui.Interface.loadStyles("styles")

    OPEN FORM f1 FROM "form"
    DISPLAY FORM f1

    INPUT BY NAME rec.* WITHOUT DEFAULTS

END MAIN
```

Graphical result:

Figure 45: Form displayed based on styles applied

How the styles were applied

1. All labels get a blue text color and sans-serif font family because of the `name="Label"` style.
2. Label 3 and Edit 3 defined with the `bigfont` style name get a large font because of the `name=".bigfont"` style.
3. The Edit field having the focus gets a yellow background color because of the `name="Edit:focus"` style (using the `focus` pseudo-selector).
4. Edit fields defined with the `mandatory` style name get a red text color because of the `name="Edit.mandatory"` style.

Example 2: Defining styles for table rows
This example shows how to define styles for tables and table rows.

The presentation style definition file:

```
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <!-- Applies to all type of elements -->
  <Style name=".bigfont">
     <StyleAttribute name="fontSize" value="large" />
  </Style>
  <!-- Background color form odd rows in tables -->
  <Style name="Table:odd">
     <StyleAttribute name="backgroundColor" value="yellow" />
  </Style>
</StyleList>
```

The form definition file:

```
LAYOUT
TABLE
{
[c1   |c2            |c3                ]
[c1   |c2            |c3                ]
[c1   |c2            |c3                ]
[c1   |c2            |c3                ]
}
END
ATTRIBUTES
EDIT c1 = FORMONLY.col1, TITLE="C1";
EDIT c2 = FORMONLY.col2, TITLE="C2";
EDIT c3 = FORMONLY.col3, TITLE="C3", STYLE="bigfont";
END
```

```
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

Program source file:

```
MAIN
    DEFINE arr DYNAMIC ARRAY OF RECORD
                col1 INTEGER,
                col2 STRING,
                col3 STRING
            END RECORD,
            i INTEGER

    FOR i=1 TO 20
        LET arr[i].col1 = i
        LET arr[i].col2 = "Item #"||i
        LET arr[i].col3 = IIF(i MOD 2, "odd", "even")
    END FOR

    CALL ui.Interface.loadStyles("styles")

    OPEN FORM f1 FROM "form"
    DISPLAY FORM f1

    DISPLAY ARRAY arr TO sr.*

END MAIN
```

Graphical result:

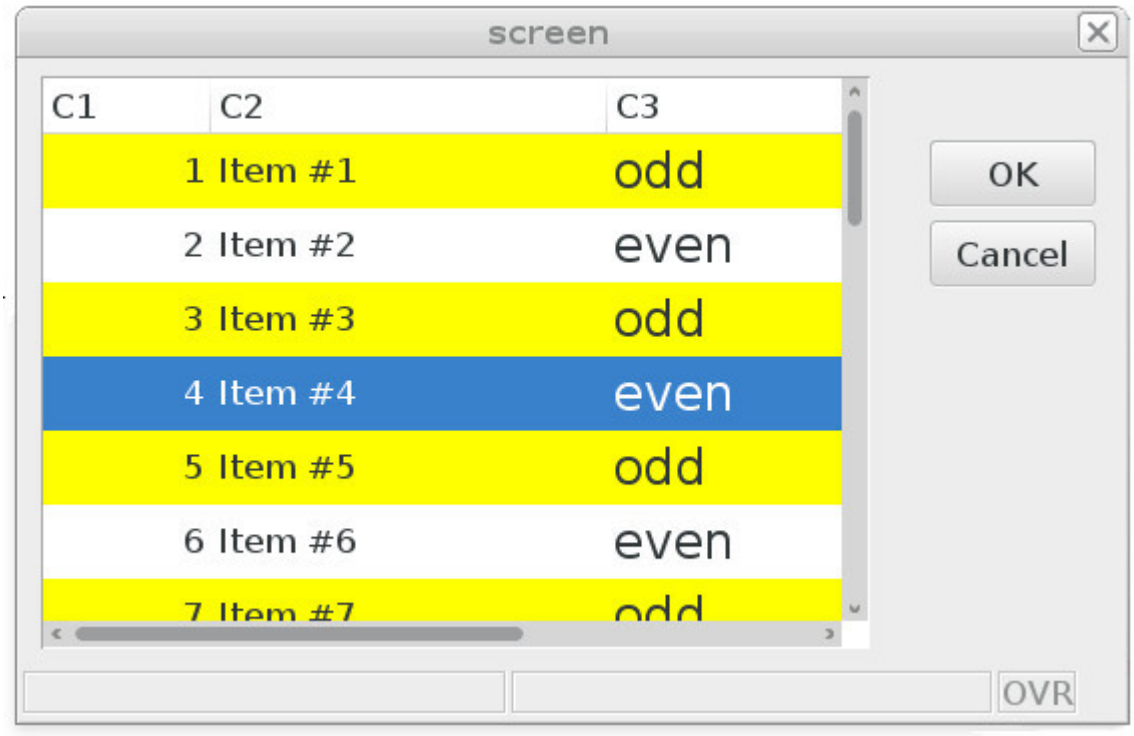


Figure 46: Form displayed based on styles applied

hat has the focus and

How the styles were applied

1. The odd rows get a yellow background because of the `name="Table:odd"` style (using the `odd` pseudo-selector).
2. Column 3 defined with the `bigfont` style name gets a large font because of the `name=".bigfont"` style.

Form specification files

Form specification files are the source files defining the layout and content of application forms.

- Understanding form files on page 853
- Form file concepts on page 853
- Form file structure on page 901
- Form item attributes on page 951
- Form rendering on page 1002
- Examples on page 1001

Understanding form files

A *form specification file* is a source file that defines an application form, to let the end user interact with the program.

The form file defines the disposition, presentation (i.e. decoration), and behavior of screen elements called form items.

The source file must have the `.per` file extension: `myform.per`. Programs load the `.42f` compiled version of the form files, and use interactive instructions (dialogs) to control the form.

To compile a `.per` source file to a `.42f` format, use the `fglform` form compiler. When a `SCHEMA` is specified in the form file, fglform requires that the database schema files already exist. Compiled form files depend on both the source files and the database schema files.

Compiled forms will be loaded by the programs with the `OPEN FORM` or the `OPEN WINDOW WITH FORM` instructions. The FGLRESOURCEPATH environment variable must contain the directory where the compiled form files are located at runtime, if the form file is not in the current directory.

Once a form is loaded, the program can manipulate forms to display or let the user edit data, with interactive instructions such as `INPUT` or `DISPLAY ARRAY`. Program variables are used as display and/or input buffers.

The content of a `.per` form file must follow a specific syntax as described in Form file structure on page 901.

Form file concepts

To write a form specification file, you need to understand the concepts described in this section.

- Form items on page 853
- External form inclusion on page 900
- Boolean expressions in forms on page 901

Form items

The concept of *form item* includes all elements used in the definition of a form.

Definition

A form item can be an input field such as an `EDIT` field, a push `BUTTON` or a `GROUPBOX` or `TABLE` container. A form item can also be an element of a `TOOLBAR`, `TOPMENU` and `ACTION DEFAULTS` definition.

A form item can be:

- A satellite item
- A static item

- A layout item
- A stack item
- An action view
- A form field

Form item types

A form item is defined by its type, called a *form item type*. For example, a form field can be an `EDIT`, or a `COMBOBOX`, a form layout container can be a `GROUP`, or a `GRID`, a toolbar item can be an `ITEM` or a `SEPARATOR`.

For a detailed description, see Form item types on page 878.

Form items in grid-based containers

In a grid-based container such as `GRID`, form items (typically, form fields) must be defined with a form tag in the `LAYOUT` section, bound by the tag name to a definition in the `ATTRIBUTES` section.

The form tag defines the position and length of the form item, while the appearance and the behavior the form item is defined by a set of attributes in the `ATTRIBUTES` section:

```
LAYOUT
GRID
{
   [f1            ]
   ...
}
END
END
...
ATTRIBUTES
EDIT f1 = customer.cust_name, ... ;
END
```

Form items in stack-based forms

In a stack-based container (`STACK`), form items (typically, form fields), are grouped and arranged in a given order, that will define their position in the stacked layout. The appearance and the behavior the form item is defined by a list of attributes in the stack item definition:

```
LAYOUT
 STACK
  GROUP
    EDIT customer.cust_name, ... ;
  END
 END
END
```

Satellite form items

Other kind of form items are defined in the section it belongs to (for example, an `ITEM` element of a `TOOLBAR` definition).

Satellite items
Satellite items are display elements defined outside the `LAYOUT` section.

Satellite items like the TOOLBAR section are form elements independent from the main form layout, and are defined additionally to the `LAYOUT` section.

```
TOOLBAR -- Toolbar section
```

```
...
END
LAYOUT -- Main layout section
...
END
```

Static items
A *static item* defines a simple form item as a final grid element (i.e. that does not change).

A static item is a form element that is defined directly in a grid of the form `LAYOUT` section, such as a text (typically, a field label).

Static items are identified by the `fglform` compiler and converted to a AUI tree node element in the resulting `.42f` file.

Simple texts

It is possible to define simple texts and field labels in the form layout:

```
LAYOUT
GRID
{
A simple text
}
END
END
```

> **Note:** To simplify internationalization, consider using named static labels instead of hard-coded text in the form layout.

Horizontal lines

You define a horizontal line with a sequence of hyphen-minus (`-`) characters in a grid:

```
LAYOUT
GRID
{
This is a horizontal line: ------------
}
END
END
```

> **Note:** Horizontal lines are mainly provided for TUI mode applications. While horizontal lines will be represented by some GUI front-ends, it is not a typical practice in common graphical applications.

Layout items
Layout items are containers with a body that can hold other form items, in a grid-based layout form.

Layout items can be specified as a tree of nested containers, or as layout tags within a single `GRID` container.

The next example shows a tree of nested containers, where a `GRID` and `TABLE` are included in a `VBOX`:

```
LAYOUT
VBOX
 GRID ...
 {
 }
 END
 TABLE ...
 {
 }
```

```
 END
END
```

The next example shows a GRID container including layout tags. The layout tags group form fields in dedicated areas. This syntax is usually more convenient to describe application forms:

```
LAYOUT
GRID
{
<g g1                       >
 Name: [f01                ]
<                           >
<t t1                       >
[c1  |c2                    ]
<                           >
}
END
END
```

Stack items
Stack items are form elements used to define a stack-based layout in a STACK container.

To define a stacked layout within a STACK container, leaf stack items (typically, form fields, labels, buttons) are specified inside grouping stack items such as GROUP or TABLE.

The next example shows a stack-based form definition with a GROUP stack item containing two EDIT stack items:

```
LAYOUT
 STACK
   GROUP g1
       EDIT customer.cust_num, NOENTRY;
       EDIT customer.cust_name, REQUIRED;
   END
 END
END
```

Action views
An *action view* defines a form item that can trigger an action in the program.

Action views as satellite items

Below is TOOLBAR section defining a toolbar button using the close action name. Here no layout tag is used because the toolbar item is part of the toolbar graphical object (it will not appear in the form layout area):

```
TOOLBAR
  ITEM close (TEXT="Close")
END
```

Action views in grid-based container

The position and size of the element is defined with an item tag, while the rendering and behavior is defined in the ATTRIBUTES section. Both parts are bound by the name of the item tag. The item tag name is local to the .per file and is not available at runtime.

The next example defines a BUTTON form item, where the item tag name is "b_close", and the button name (and the action name) is "close":

```
LAYOUT
GRID
```

```
{
  ...
  [b_close     ]
}
END
END
...
ATTRIBUTES
BUTTON b_close: close, TEXT="Close";
END
```

Action views in stack-based layout

In a stack-based container, action views are defined as stack items, with the attribute defining the rendering and behavior:

```
LAYOUT
STACK
 GROUP group1 ( TEXT="Customer" )
  ...
  BUTTON print, TEXT="Print Report", IMAGE="printer";
  ...
```

Form fields
Form fields are form elements designed for data input and/or data display.

Purpose of form fields

A form field is a form item dedicated to data management. It associates a form item with a screen record field. The screen record field will be used to bind program variables in interaction instructions (i.e. dialogs). The program variables will be the data models for the form fields.

There are different sort for form fields:

- Database column fields on page 858
- Formonly fields on page 860
- Phantom fields on page 861
- Aggregate fields on page 863

Form fields can be used in a grid-based layout or in a stack-based layout.

Form fields are identified by the field name in programs, and are grouped in screen records (or screen arrays in case of list containers). The interactive instruction must mediate between screen record fields and database columns by using program variables.

Form fields are usually related to database column, which types are defined in the database schema file.

Forms fields in grid-based containers

In a grid-based container, the position and size of a form field is defined with an item tag in the form layout, while the rendering and behavior is defined in the `ATTRIBUTES` section. Both parts are bound by the name of the item tag. The item tag name is local to the `.per` file and is not available at runtime: It is just the key to bind the item tag (position) with the item definition (attributes).

In the next example, the "`f1`" item tag (in the `LAYOUT` section) is linked to the "`vehicle.num`" form field definition (in the `ATTRIBUTES` section), which references a column of the "`vehicle`" table, defined in the "`carstore`" database schema:

```
SCHEMA carstore
LAYOUT
GRID
```

```
{
Number:   [f1           ]
Name:     [f2                       ]
}
END
END
TABLES
  vehicle
END
ATTRIBUTES
  EDIT f1 = vehicle.num, STYLE="keycol";
  EDIT f2 = vehicle.name, UPSHIFT;
END
```

Forms fields in stack-based containers

In a stack-based container, the visual position of a form field is defined by the ordinal position of the stack item in the stack definition, while the rendering and behavior are defined with stack item attributes.

In the next example, the "`vehicle.num`" form field definition references a column of the "`vehicle`" table, defined in the "`carstore`" database schema:

```
SCHEMA carstore
LAYOUT
  STACK
  GROUP
    EDIT vehicle.num, REQUIRED, STYLE="keycol";
  END
  END
END
TABLES
  vehicle
END
```

Database column fields

Form fields defined with a table and column name get data type from the database schema file.

Syntax 1: In grid-based container

```
item-type item-tag = [table.]column
    [ , attribute-list ]  ;
```

Syntax 2: In stack-based container

```
item-type [table.]column
    [ , attribute-list ]  ;
```

1. *item-type* references an item type like `EDIT`.
2. *item-tag* identifies the layout location of the field.
3. *table* is the name or alias of a table, synonym, or view, as declared in the `TABLES` section.
4. *column* is the name of a database column.
5. *attribute-list* is a list of field attributes.

Usage

A form field is typically based on the definition of a database column found in the database schema specified with the `SCHEMA` clause at the beginning of the form file. The database column defines the data type of the form field.

> **Important:** The data type of a form field is only used by the `CONSTRUCT` interactive statement to do database queries. When using the form field with an `INPUT`, `INPUT ARRAY` or `DISPLAY ARRAY` dialog, the type of the program variable defines the data type of the form field.

In order to reference database columns, the table name must be listed in the `TABLES` section of the form.

Fields are associated with database columns only during the compilation of the form specification file: The form compiler examines the database schema file to identify the data type of the column, and defines the form field with this type. This technique allows to centralize form field data types in the schema files: If the data type of a column changes, extract the schema again and recompile your forms to take the new type into account.

> **Note:** The compilers do also grab other field attributes like validation rules and video display attributes from `.val` and `.att` schema files. However, this is supported for backward compatibility only (formerly stored in syscolval and syscolatt database tables). Consider reviewing programs using this feature.

After the form compiler identifies data types from the schema file, the association between fields and database columns is broken, and the form cannot distinguish the name or synonym of a table or view from the name of a screen record.

The programs only have access to screen record fields, in order to display or input data using program variables. Regardless of how you define them, there is no implicit relationship between the values of program variables, form fields, and database columns. Even, for example, if you declare a variable `lname LIKE customer.lname`, the changes that you make to the variable do not imply any change in the column value. Functional relationships among these entities must be specified in the program code, through screen interaction statements, and through SQL statements. It is up to the programmer to determine what data a form displays and what to do with data values that the user enters into the fields of a form. You must indicate the binding explicitly in any statement that connects variables to forms or to database columns.

If a form field is declared with a table column using the SERIAL, SERIAL8 or BIGSERIAL SQL type, the field will automatically get the `NOENTRY` attribute, except if the field is defined with the `TYPE LIKE` syntax.

Example

Grid-based container database form field definition:

```
SCHEMA stores  -- Database schema
LAYOUT
GRID
{
 [f001             ]
 ...
}
END
END
TABLES
  customer  -- Database table
END
ATTRIBUTES
EDIT f001 = customer.fname,  -- DB-col form field
     REQUIRED, COMMENTS="Customer name";
...
```

Stack-based container database form field definition:

```
SCHEMA stores  -- Database schema
TABLES
  customer  -- Database table
END
LAYOUT
```

```
  STACK
  GROUP
    EDIT customer.fname, -- DB-col form field
         REQUIRED, COMMENTS="Customer name";
  ...
```

Formonly fields

FORMONLY form fields define their data type explicitly, with or without referencing a database columns.

Syntax 1: In grid-based container

```
item-type item-tag = FORMONLY.field-name
   [ TYPE
      { LIKE [table.]column
      | data-type [NOT NULL] }
   ]
     [ , attribute-list ]  ;
```

Syntax 2: In stack-based container

```
item-type FORMONLY.field-name
   [ TYPE
      { LIKE [table.]column
      | data-type [NOT NULL] }
   ]
     [ , attribute-list ]  ;
```

where *data-type* is one of:

```
{ CHAR
| DECIMAL [(p[,s])]
| SMALLFLOAT
| REAL
| FLOAT
| MONEY [(p[,s])]
| INTEGER
| SMALLINT
| DATE
| VARCHAR
| TEXT
| BYTE
| INTERVAL interval-qualifier
| DATETIME datetime-qualifier
| BIGINT
| BOOLEAN
}
```

1. *table* is the name or alias of a table, synonym, or view, as declared in the TABLES section.
2. *column* is the name of a database column.
3. *field-name* is the identifier that will be used in programs to handle the field.
4. *interval-qualifier* is an INTERVAL qualification clause such as HOUR(5) TO SECOND.
5. *datetime-qualifier* is a DATETIME qualification clause such as DAY TO SECOND.

Usage

Form fields can be specified with the FORMONLY prefix, when there is no corresponding database column, or when the field must be defined with another name as the database column.

Important: The data type of a form field is only used by the CONSTRUCT interactive statement to do database queries. When using the form field with an INPUT, INPUT ARRAY or DISPLAY ARRAY dialog, the type of the program variable defines the data type of the form field.

When using the LIKE [table.]column syntax, the form field will get the data type of the specific table column as defined in the database schema. The table name must be specified in the TABLES section.

When using the TYPE *data-type* clause, you explicitly specify the type of the field. Note that for CHAR/VARCHAR data types, the size is defined by the item tag length in the layout.

If no data type is specified, and no database column is referenced, the default data type is CHAR.

Specifying a data type followed by the NOT NULL keywords is equivalent to the NOT NULL attribute.

The STRING data type is not supported in formonly form field definitions.

The definition of FORMONLY fields can be completed by using the DISPLAY LIKE and VALIDATE LIKE attributes, to get the display and validation attributes from the .att and .val database schema files.

Example

Grid-based container FORMONLY form field definition (in the ATTRIBUTES section):

```
LAYOUT
GRID
{
 [f001             ]
 [f002             ]
 ...
}
END
END
ATTRIBUTES
EDIT f001 = FORMONLY.total TYPE DECIMAL(10,2), NOENTRY ;
EDIT f002 = FORMONLY.name TYPE LIKE customer.cust_name,
            VALIDATE LIKE customer.cust_name ;
```

Stack-based container FORMONLY form field definition:

```
LAYOUT
  STACK
  GROUP
    EDIT FORMONLY.total TYPE DECIMAL(10,2), NOENTRY ;
    EDIT FORMONLY.name TYPE LIKE customer.cust_name, REQUIRED;
```

Phantom fields

A PHANTOM field defines a screen-record field which is not rendered in the layout (it acts as a hidden field).

Syntax

```
PHANTOM { [table.]column
        | FORMONLY.field-name
            [ TYPE
              { LIKE [table.]column
              | data-type [NOT NULL] }
            ]
    } ;
```

where *data-type* is one of:

```
{ CHAR
| DECIMAL [(p[,s])]
```

```
| SMALLFLOAT
| REAL
| FLOAT
| MONEY [(p[,s])]
| INTEGER
| SMALLINT
| DATE
| VARCHAR
| TEXT
| BYTE
| INTERVAL interval-qualifier
| DATETIME datetime-qualifier
| BIGINT
| BOOLEAN
}
```

1. *table* is the name or alias of a table, synonym, or view, as declared in the `TABLES` section.
2. *column* is the name of a database column.
3. *field-name* is the identifier that will be used in programs to handle the field.
4. *interval-qualifier* is an `INTERVAL` qualification clause such as `HOUR(5) TO SECOND`.
5. *datetime-qualifier* is a `DATETIME` qualification clause such as `DAY TO SECOND`.

Usage:

A `PHANTOM` field defines a form field listed in a screen-record or screen-array, that has no corresponding layout element. It is only used for the screen-record (or screen-array) definition, to bind with program variables used by dialogs, typically to match a given database table definition.

Phantom fields will be used by dialog instructions as regular form fields, but will not be displayed to the end user, and the end user will not be able to enter values for these fields. Data hold by phantom fields is never send to the front-ends: They can be used to store critical data that must not go out of the application server.

Phantom fields can be based on columns defined in a database schema file, or as `FORMONLY` field.

For example, if you want to implement a screen-array with all the columns of a database table defined in the database schema file, but you don't want to display all the columns in the `TABLE` container of the `LAYOUT` section, you must use `PHANTOM` fields. With the screen-array matching the database table, you can easily write program code to fetch all columns into an array defined with a `LIKE` clause.

Example (grid-based layout)

Form file:

```
SCHEMA carstore
LAYOUT( TEXT = "Vehicles" )
GRID
{
<T t1                                    >
 Num      Name            Price
[c1      |c2             |c3            ]
[c1      |c2             |c3            ]
[c1      |c2             |c3            ]
}
END
END
TABLES
  vehicle
END
ATTRIBUTES
  TABLE t1: table1;
  EDIT c1 = vehicle.num;
```

```
  EDIT c2 = vehicle.name;
  EDIT c3 = vehicle.price;
  PHANTOM vehicle.available;  -- not used in layout
END
INSTRUCTIONS
  SCREEN RECORD sr(vehicle.*);
END
```

Program code:

```
SCHEMA carstore
...
DEFINE vl DYNAMIC ARRAY OF RECORD LIKE vehicle.*
...
DISPLAY ARRAY vl TO sr.*
...
```

Aggregate fields

An AGGREGATE field defines a screen-record field to display summary information for a TABLE column.

Syntax

```
AGGREGATE item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Usage

An AGGREGATE field defines a form field that is used to display a summary cell for a given column of a TABLE container. The aggregate fields are displayed after the last data line of the table. Such fields are typically used to show computed values for the corresponding column which appears above the aggregate cell.

> **Important:** This feature is not supported on mobile platforms.

An aggregate field can be based on a database column defined in a schema file, or as FORMONLY field.

The AGGREGATETYPE attribute defines how the value of the field will be computed. For example, the SUM keyword (the default) can be used to instruct the runtime system to automatically compute the total of the associated column. By using the PROGRAM keyword, you indicate that the value of the aggregate field will be computed and displayed by program code. A simple DISPLAY BY NAME or DISPLAY TO can be used to show the summary value.

The value displayed in the AGGREGATE field follows the FORMAT attribute of the corresponding column, if defined. The FORMAT attribute is applied for automatically computed values, as well as for values displayed by user code with DISPLAYBY NAME or DISPLAY TO.

The label of an aggregate field can be specified with the AGGREGATETEXT attribute. The text defined with this attribute will be displayed on the left of the aggregate value (in the aggregate cell), except if there is no room to display the label (for example if the aggregate value is too large or if the column values are aligned to the left). An aggregate label can be a localized string with the %"..." string syntax. You can also specify an AGGREGATETEXT attribute at the TABLE level, to get a global label for the summary line. If no text is defined for an aggregate field, the global aggregate text will appear on the left in the summary line.

Table aggregate decoration can be modified with a presentation style. Use the summaryLine pseudo-selector to change the font type and color, as well as the background of the summary line. Use the

`summaryLineAlwaysAtBottom` table style attribute to force the summary line to stay on the bottom of the table.

Aggregate fields in grid-based layout

The item tag of an aggregate field must appear in the last line in the layout block of the`TABLE` container, and must be aligned vertically with a table column item tag. You can specify several aggregate item tags for the same table:

```
TABLE
{
[c1      |c2       |c3           |c4       |c5      ]
[c1      |c2       |c3           |c4       |c5      ]
[c1      |c2       |c3           |c4       |c5      ]
[c1      |c2       |c3           |c4       |c5      ]
         [cnt      ]             [tot_c4   |tot_c5  ]
}
END
```

Aggregate fields in stack-based layout

Important: Aggregate fields are not supported in tables defined in a `STACK` container.

Example (grid-based layout)

```
SCHEMA stores
LAYOUT( TEXT = "Orders" )
GRID
{
<T t1                                    >
 Num      Date            Order total
[c1      |c2             |c3           ]
[c1      |c2             |c3           ]
[c1      |c2             |c3           ]
[c1      |c2             |c3           ]
                         [total        ]
}
END
END
TABLES
  orders
END
ATTRIBUTES
  TABLE t1: table1;
  EDIT c1 = orders.o_num;
  EDIT c2 = orders.o_date;
  EDIT c3 = orders.o_tot;
  AGGREGATE total = FORMONLY.o_total,
                        AGGREGATETEXT = "Total:",
                        AGGREGATETYPE = SUM;
END
INSTRUCTIONS
  SCREEN RECORD sr(orders.*);
END
```

Identifying form items

Elements defined in a form file can be identified with a name, to be used in programs.

Form fields are implicitely identified by the `tabname.colname` specification after the equal sign, while other (non-field) form items such as static labels and group boxes can get an optional item name.

The form item name defined in the form file will be copied to the `name` attribute of the corresponding node in the `.42f` file. It can then be used by programs to select a form element at runtime, to introspect or modify its attributes.

For example, specify the name for a `GROUP` container by writing an identifier after the layout container type:

```
GROUP group1 (TEXT="Customer")
```

Here the group name is '`group1`', and it can be used in a program to identify the group element:

```
DEFINE w ui.Window
DEFINE g om.DomNode
LET w = ui.Window.getCurrent()
LET g = w.findNode("Group","group1")
CALL g.setAttribute("text","Another text")
```

Helper methods are provided for common tasks on form elements. For example, to hide a group with the identifier `group1`, you can use the `setElementHidden()` method on a `ui.Form` object:

```
DEFINE f ui.Form
...
   LET f = DIALOG.getForm()
   ...
   CALL f.setElementHidden("group1", TRUE)
```

> **Note:** Consider defining unique names to form elements to be identified, to simplify the search at runtime. A good practice is the use a prefix based on the type of form element (`g_` for groups, `l_` for labels for example).

Static items in a grid-based layout container cannot get a name, because these are self-defined with the layout part of the item:

```
GRID
{
Name: [f1              ]
...
}
END
```

In the above example, the label "Name:" cannot be identified. In order to give a name to such label, use an item tag and add a `LABEL` line in the `ATTRIBUTES` section, and specify the name of the label after the colon:

```
GRID
{
[l1   ][f1              ]
...
}
END
...
ATTRIBUTES
LABLE l1: l_name, TEXT="Name:";
...
```

Screen records

Form fields can be grouped in a *screen record* or *screen array* definition. A screen array is a screen record with a dimension, to handle a list of records.

Syntax

```
SCREEN RECORD record-name [ size ] ( field-list )
```

where *field-list* is:

```
{ table.*
| field-name
| first-field [ {THROUGH|THRU} last-field ]
[,...] }
```

1. *record-name* is the name of an explicit screen record or screen array.
2. *size* is an integer representing the number of records in the screen array.
3. *field-name* is a field identifier as defined in the right operand of a field definition in the `ATTRIBUTES` section.
4. *first-field* and *last-field* are field identifiers like *field-name*. This notation instructs the form compiler to take all the fields defined between the first and last field (inclusive).
5. *table* is the name or alias of a table, synonym, or view, as declared in the `TABLES` section. This notation instructs the form compiler to build the screen record with all fields declared in the `ATTRIBUTES` section for the given table.

Usage

Screen records and screen arrays are defined with the `SCREEN RECORD` keywords in the `INSTRUCTIONS` section of a form specification file to name a group of fields.

Screen records

A screen record is a named group of fields that screen interaction statements of the program can reference as a single object. By establishing a correspondence between a set of screen fields (the screen record) and a set of program variables (typically a program record), you can pass values between the program and the fields of the screen record. In many applications, it is convenient to define a screen record that corresponds to a row of a database table.

Like the name of a screen field, the identifier of a screen record must be unique within the form, and it has a scope that is restricted to when its form is open. Interactive statements can reference *record-name* only when the screen form that includes it is being displayed. The form compiler returns an error if *record-name* is the same as the name or alias of a table in the TABLES section.

```
SCHEMA videoshop
LAYOUT
GRID
{
  Customer id: [f001     ]
  Name:        [f002                          ]
  Create date: [f003     ]
}
END
END
TABLES
customer
END
INSTRUCTIONS
SCREEN RECORD sr_customer
(
```

```
 customer.cust_id,
 customer.cust_name,
 customer.cust_crea
);
END
```

Default screen records

The form compiler builds default screen records that consist of all the screen fields linked to the same database table within a given form. A default screen record is automatically created for each table that is used to reference a field in the `ATTRIBUTES` section.

The components of the default record correspond to the set of display fields that are linked to columns in that table. The name of the default screen record is the table name (or the alias, if you have declared an alias for that table in the `TABLES` section). For example, all the fields linked to columns of the "customer" table constitute a default screen record whose name is "customer".

If a form includes one or more `FORMONLY` fields, those fields constitute a default screen record called "formonly".

Screen arrays

A screen array is similar to a screen record, except that it defines a additional *size*. Screen arrays are typically used to reference rows in a static list of fields defined in the `LAYOUT` section. Each row of a screen array is a screen record. Each column of a screen array consists of fields with the same field tag in the `LAYOUT` section.

The *size* value must be equal to the number of lines of the static list of field tags in the layout of the form. For example, a `GRID` container might represent a set fields organized in columns like this:

```
LAYOUT
GRID
{
  OrdId      Date        Total Price
 [f001      |f002       |f003       ]
 [f001      |f002       |f003       ]
 [f001      |f002       |f003       ]
 [f001      |f002       |f003       ]
}
END
END
```

This example requires a *size* of 4 when defining the corresponding screen array:

```
INSTRUCTIONS
SCREEN RECORD sr_orders[4]
(
 order.ord_id,
 order.ord_date,
 order.ord_total
);
END
```

You cannot define multiple screen arrays for the same `TABLE` definition. Only one `SCREEN RECORD` specification is allowed.

Screen arrays must specify a size when referencing fields that define a static list in the layout. When referencing the columns of a variable-size record list container such as a TABLE or a TREE, the corresponding screen array must be defined without a size:

```
LAYOUT
```

```
TABLE
{
  OrdId      Date        Total Price
 [f001      |f002       |f003       ]
 [f001      |f002       |f003       ]
 [f001      |f002       |f003       ]
 [f001      |f002       |f003       ]
}
END
END
```

This TABLE layout does not require a *size* specification when defining the corresponding screen array:

```
INSTRUCTIONS
SCREEN RECORD sr_orders
(
 order.ord_id,
 order.ord_date,
 order.ord_total
);
END
```

Using screen records and screen arrays in programs

Screen records and screen arrays can display program records. If the fields in the screen record have the same sequence of data types as the columns in a database table, you can use the screen record to simplify operations that pass values between program variables and rows of the database.

Screen records are usually not referenced in programs within single record input statements, because program variable to form field binding is typically done by name with the `INPUT BY NAME` instruction.

Screen array names are typicaly referenced in programs within interactive dialog controlling a list of records such as `DISPLAY ARRAY` and `INPUT ARRAY`. The current form must include that named screen array.

Form tags
Form tags define layout elements inside a grid-based container.

Form tags are place holders used inside a grid of the layout section, to define the position and the relation between form items.

The syntax and purpose of a form tag depends on the type of form tag.

The different sort of form tags are:

- Layout tags on page 868
- Item tags on page 873
- Hbox tags on page 875

Layout tags
Layout tags define layout areas for containers inside the frame of a grid-based container.

Syntax

```
  <type  [identifier]      >
    content
[ <                        > ]
```

1. *type* defines the kind of layout tag to be inserted at this position.
2. *identifier* references a form item definition in the `ATTRIBUTES` section, it must be unique, but is optional.
3. *content* defines other form items inside the layout tag.

4. The (< >) ending the layout tag body is optional.

Usage

A layout tag defines a layout region of a container, in the body frame of a `GRID` container.

While complex layout with nested frames can be defined with `HBOX` and `VBOX` containers, it is sometimes more convenient to define a form with a complex layout by using layout tags within a `GRID` container.

A layout tag has a type that defines what kind of container will be generated in the compiled form.

A layout tag is delimited by angle braces (<>), and contains the tag type (`G`/`GROUP`, `T`/`TABLE`, etc) and an optional identifier.

Table 257: Types of layout tags

Tag Type	Abbr.	Description
`GROUP`	`G`	Defines a group box layout tag, resulting in the same presentation as the GROUP container.
`TABLE`	`T`	Defines a table view layout tag, resulting in the same presentation as the TABLE container.
`TREE`	*N/A*	Defines a tree-view list view layout tag, resulting in the same presentation as the TREE container.
`SCROLLGRID`	`S`	Defines a scrollable grid layout tag, resulting in the same presentation as the SCROLLGRID container.

The details of the layout tag definition are specified in the `ATTRIBUTE` section. Layout tags must be identified by an item tag name. In the next example, the layout tag named "g1" is defined in the `ATTRIBUTE` section with the `GROUP` form item type to set the name and text:

```
LAYOUT
GRID
{
<GROUP g1        >
[text1           ]
[                ]
[                ]
<                >
}
END
END
ATTRIBUTES
GROUP g1:group1, TEXT="Description";
TEXTEDIT text1=FORMONLY.text1;
END
```

The layout region is a rectangle, in which the width is defined by the length of the layout tag, and the height by a closing tag (< >) . In the next example, the layout region is defined by the layout tag named "group1".

```
<GROUP group1            >
```

```
<                        >
```

Form items must be placed inside the layout region. The [ ] square brackets are not part of the form item width and can be place at the same X position as the layout tag delimiters:

```
<GROUP group1             >
 Item:     [f001          ]
 Quantity: [f002    ]
 Date:     [f003          ]
<                         >
```

The [ ] square brace delimiters are not counted to define the width of an item tag. The width of the item is defined by the number of character between the square braces. Thus, this layout is valid and can be compiled:

```
<GROUP group1            >
[f001                    ]
[f002                    ]
 Static labels must fit!!
<                        >
<TABLE table1            >
[colA  |colB              ]
[colA  |colB              ]
[colA  |colB              ]
[colA  |colB              ]
```

You can place several layout tags on the same layout line in order to split the frame horizontally. This example defines six layout regions (four group boxes and two tables):

```
<GROUP group1      ><GROUP group2              ><GROUP group4 >
 FName: [f001      ] Phone: [f004              ][f012         ]
 LName: [f002      ] EMail: [f005              ][             ]
<                  ><                          >[             ]
<GROUP group3                                  >[             ]
[f010                                          ][             ]
<                                              ><             >
<TABLE table1               ><TABLE table2                    >
[c11   |c12   |c13          ][c21   |c22                      ]
[c11   |c12   |c13          ][c21   |c22                      ]
[c11   |c12   |c13          ][c21   |c22                      ]
[c11   |c12   |c13          ][c21   |c22                      ]
<                           ><                                >
```

The < > closing layout tag is optional. When not specified, the end of the layout region is defined by the underlying layout tag or by the end of the current grid. However, the ending tag must be specified if the form compiler cannot detect the end of the layout region. This is usually the case with group layout tags. In the next example, the table does not need an ending layout tag because it is defined by the starting tag of the group, but the group needs and ending tag otherwise it would include the last field (*field3*). Additionally, if *field3* would have a different size, the form compiler would raise an error because the group and the last field geometry would conflict.

```
<TABLE table1     >
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
<GROUP group2     >
[field1           ]
```

```
[field2           ]
<                 >
[field3           ]
```

It is possible to mix container layout tags with singular form items. You typically put form items using a large area of the form, such as `IMAGE` fields or `TEXTEDIT` fields. The `[ ]` square brace delimiters are not used to compute the size of the singular form items:

```
<GROUP group1             >[image1         ]
 FName: [f001             ][               ]
 LName: [f002             ][               ]
<                         >[               ]
[textedit1                  |               ]
[                           |               ]
[                           |               ]
```

Table layout tags can be embedded inside group layout tags:

```
<GROUP group1             >
 <TABLE table1           >
 [colA  |colB            ]
 [colA  |colB            ]
 [colA  |colB            ]
 [colA  |colB            ]
<                          >
```

Hbox or vbox containers with splitter are automatically created by the form compiler in these conditions:

- Hbox is created when two or more stretchable elements are stacked side by side and touch each other (no space between).
- Vbox is created when two or more stretchable elements are stacked vertically and touch each other (no space between).

Stretchable elements are containers such as `TABLE` containers, or form items like `IMAGE` fields with the `STRETCH` attribute.

No hbox or vbox object will be created if the elements are in a `SCROLLGRID` container.

This example defines two tables stacked vertically, generating a VBox with splitter (note that ending tags are omitted):

```
<TABLE table1     >
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
<TABLE table2     >
[colC  |colD      ]
[colC  |colD      ]
```

In this example, the layout defines two stretchable `TEXTEDIT` fields placed side by side which would generate an automatic hbox with splitter. To make both textedits touch you need to use a pipe delimiter in between:

```
[textedit1         |textedit2               ]
[                  |                        ]
[                  |                        ]
[                  |                        ]
```

The next layout example would make the form compiler create an automatic vbox with splitter to hold *table2* and *textedit1* , plus an hbox with splitter to hold *table1* and the first VBox (We must use a pipe character to delimit the end of *colB* and *textedit1* so that both tables can be placed side by side):

```
<TABLE table1     ><TABLE table2           >
[colA  |colB      ][colC|colD              ]
[colA  |colB      ][colC|colD              ]
[colA  |colB      ][colC|colD              ]
[colA  |colB      |textedit1                ]
[colA  |colB      |                         ]
[colA  |colB      |                         ]
```

If you want to avoid automatic hbox or vbox with splitter creation, you must add blanks between elements:

```
<TABLE table1     >  <TABLE table2           >
[colA  |colB      ]  [colC|colD              ]
[colA  |colB      ]  [colC|colD              ]
[colA  |colB      ]  [colC|colD              ]
[colA  |colB      ]
[colA  |colB      ] [textedit1                ]
[colA  |colB      ] [                         ]
[colA  |colB      ] [                         ]
```

Examples

The typical OK/Cancel window:

```
LAYOUT
GRID
{
<GROUP g1                    >
[com                         ]
<                            >
[            :bok   |bno     ]
}
END
END
ATTRIBUTES
LABEL com: comment;
BUTTON bok: accept;
BUTTON bno: cancel;
...
```

This example shows multiple uses of layout tags:

```
LAYOUT
GRID
{
<GROUP g1                              ><GROUP g2          >
 Ident: [f001   ] [f002                ] [text1            ]
 Addr:  [f003                          ] [                 ]
<                                      ><                  >
<GROUP g3                                                  >
[text2                                                     ]
[                                                          ]
[                                                          ]
<                                                          >
<TABLE t1                                                  >
  Num      Name                 State  Value
[col1    |col2                |col3  |col4                ]
[col1    |col2                |col3  |col4                ]
[col1    |col2                |col3  |col4                ]
```

```
[col1    |col2              |col3  |col4              ]
<                                                     >
}
END
END
ATTRIBUTES
GROUP g1:group1, TEXT="Customer";
GROUP g2:group2, TEXT="Comments";
TABLE t1:table1, UNSORTABLECOLUMNS;
...
```

Item tags

Item tags define the position and size in a grid-based container.

An item tag defines the position and size of a simple form item in a *grid-area* of a `GRID` or `SCROLLGRID` container. Form item defined with item tags are leafs in the structure of a form definition, such as a form field (i.e. it is not a container form item).

Syntax

```
[identifier [-] [|...]]
```

1. *identifier* references a form item definition in the `ATTRIBUTES` section.
2. The optional `-` dash defines the real width of the element.
3. The `|` pipe can be used as item tag separator (equivalent to `][`).

Usage

An item tag is delimited by square braces (`[]`) or pipes (`|`) and contains an identifier used to reference the description of the form item in the `ATTRIBUTES` section. In the next example, the identifier of the form item is "`f01`", and the form item type is `BUTTONEDIT`:

```
LAYOUT
GRID
{
   ...
   [f01            ]
   ...
}
END
...
ATTRIBUTES
BUTTONEDIT f01 = customer.cust_name, ACTION=zoom;
...
```

Each item tag must be indicated by left and right delimiters to show the length of the item and its position within the container layout. Both delimiters must appear on the same line. You must use left and right braces (`[]`) to delimit item tags. The number of characters and the delimiters define the width of the region to be used by the item:

```
GRID
{
  Name:  [f001                              ]
}
END
```

The form item position starts after the open square brace and the length is defined by the number of characters between the square braces. The following example defines a form item starting at position 3, with a length of 2:

```
GRID
{
1234567890
 [f1]
}
END
```

By default, the real width of the form item is defined by the number of characters used between the tag delimiters.

For some special items like `BUTTONEDIT`, `COMBOBOX` and `DATEEDIT`, the width of the field is adjusted to include the button. The form compiler computes the width as: *width=nbchars*-2 if *nbchars*>2:

```
GRID
{
 1234567
[f1     ]  -- this EDIT gets a width of 7
[f2     ]  -- this BUTTONEDIT gets a width of 5 (7-2)
}
END
```

If the default width generated by the form compiler does not fit, the `-` dash symbol can be used to define the real width of the item. In this example, the form item occupies 7 grid cells, but gets a real width of 5 (i.e. for an EDIT field, you would be able to enter 5 characters):

```
GRID
{
 1234567
[f1   - ]
}
END
```

To make two items appear directly next to each other, you can use the pipe symbol (`|`) to indicate the end of the first item and the beginning of the second item:

```
GRID
{
  Info:  [f001    |f002            |f003    ]
}
END
```

If you need the form to support items with a specific height (more that one line), you can specify *multiple-segment* item tags that occupy several lines of a *grid-area*. To create a multiple-segment item, repeat the item tag delimiters without the item *identifier* on successive lines:

```
GRID
{
  Multi-segment: [f001                            ]
                 [                                ]
                 [                                ]
                 [                                ]
                 [                                ]
}
END
```

The notation applies to the new LAYOUT section only. For backward compatibility (when using a SCREEN section), multiple-segment items can be specified by repeating the *identifier* in sub-lines.

If the same item tag (i.e. using the same *identifier*) appears more than once in the layout, it defines a column of a screen array:

```
GRID
{
  Single-line static screen array:
    [f001          ] [f002          ]  [f003          ]
    [f001          ] [f002          ]  [f003          ]
    [f001          ] [f002          ]  [f003          ]
    [f001          ] [f002          ]  [f003          ]
}
END
```

You can even define a multi-line list of fields:

```
GRID
{
  Multi-line static screen array:
    [f001          ]  [f002          ]
        [f003                                          ]
    [f001          ]  [f002          ]
        [f003                                          ]
    [f001          ]  [f002          ]
        [f003                                          ]
    [f001          ]  [f002          ]
        [f003                                          ]
}
END
```

Hbox tags

Hbox tags group several item tags within the same horizontal layout box, in a grid-based container.

An hbox tag defines the position and size in a GRID container for an horizontal box containing several leaf form items. The elements in the hbox tag can use additional alignment rules to get the required visual affect.

Syntax

```
[ element: [...] ]
```

where *element* can be:

```
{ identifier  [-] | string-list }
```

where *string-list* is:

```
{ string-literal | spacer } [...]
```

1. *identifier* references a form item definition in the ATTRIBUTES section.
2. The optional - dash defines the real width of the element.
3. string-list is a combination of string-literals
4. *string-literal* is a quoted text that defines a static label.
5. *spacer* is one or more blanks that define an invisible element that expands automatically.
6. The colon is the delimiter for hbox tag elements.

Usage

Hbox tags are provided to control the alignment of form items in a grid. Hbox tags allow you to stack form items horizontally without the elements being influenced by elements above or below. In an hbox, you can mix form fields, static labels and spacers. A typical use of the hbox is to have zip-code / city form fields side by side with predictable spacing in-between.

An hbox tag is delimited by square braces (`[]`) and must contain at least one *string-list* or an *identifier* preceded or followed by a colon (`:`). A *string-list* is combination of *string-literals* (quoted text) and *spacers* (blank characters). The delimiter for hbox tag elements is the colon.

Hbox tags are not allowed for fields of screen arrays; you will get a form compiler error. The client needs a matrix element directly in a grid or a scrollgrid to perform the necessary positioning calculations for the individual fields.

The following example shows simple hbox tags:

```
GRID
{
 ["Customer info:":    ]
 [f001       :         ]
 [  :f002              ]
 ["Name: "  :f003      ]
}
END
```

In this example:

1. The first hbox tag contains two elements: a static label and a spacer.
2. The second hbox tag contains two elements: a form field and a spacer.
3. The third hbox tag contains two elements: a spacer and a form field.
4. The fourth hbox tag contains two elements: a static label and a form field.

An hbox tag defines the position and width (in grid cells) of several form items grouped inside an horizontal box. The position and width (in grid cells) of the horizontal box is defined by the square braces (`[]`) delimiting the hbox tag.

When using an *identifier*, you define the position of a form item which is described in the `ATTRIBUTES` section. When using a *string-list*, you can define static labels and/or spacers. The following example defines an hbox tag generating 7 items (a static label, a spacer, a form item identified by `num`, a spacer, a static label, a spacer and a form item identified by `name`):

```
GRID
{
 ["Num:"  :num  :  :"Name:"  :name        ]
}
END
```

A *spacer* is an invisible element that automatically expands. It can be used to align elements left, right or center in the hbox. The following example defines 3 hboxes with the same width. Each hbox contains one field. The first field is aligned to the left, the second is aligned to the right and third is centered:

```
GRID
{
 [left :              ]
 [         :right     ]
 [     :centered:     ]
}
END

ATTRIBUTES
 LABEL left: label1, TEXT="LEFT";
```

```
 LABEL right: label2, TEXT="RIGHT";
 LABEL centered: label3, TEXT="CENTER";
END
```

When you use string literals, the quotes define where the label starts and stops. If there is free space after the quote that ends the label, then it is filled by a spacer. Consider this example:

```
GRID
{
 [    :"Label1"     ]
 [         :"Label2"]
}
END
```

In this example:

1. The first line contains a spacer, followed by the static label, followed by another spacer. The quotation marks end the string literal; a colon is not required to delimit the label from the final spacer.
2. The second line contains a spacer, followed by the static label. Because there is no empty space between the end of the static label and the closing bracket of the hbox Tag ( ]).

A typical use of hbox tags is to vertically align some form items - that must appear on the same line - with one or more form items that appear on the other lines:

```
GRID
{
 Id:      [num  :"Name: ":name        ]
 Address: [street                  : ]
          [zip-code:city             ]
 Phones:  [phone       :fax          ]
}
END
```

In this example, the form compiler will generate a grid containing 7 elements (3 labels and 4 hboxes):

1. The label "Id:",
2. A first hbox which defines 3 cells, where:
   - The field 'num' will occupy the cell (1,1),
   - The label "Name:" will occupy the cell (2,1),
   - The field 'name' will occupy the cell (3,1).
3. The label "Address:" will occupy cell (1,2),
4. A second hbox which defines 1 cell, where:
   - The field 'street' will occupy the cell (1,1).
5. A third hbox which defines 2 cells, where:
   - The field 'zip-code' will occupy the cell (1,1),
   - The field 'city' will occupy the cell (2,1).
6. The label "Phones:" will occupy cell (1,4),
7. A fourth hbox which defines 2 cells, where:
   - The field 'phone' will occupy the cell (1,1),
   - The field 'fax' will occupy the cell (2,1).

Inside an hbox tag, the positions and widths of elements are independent of other hboxes. It is not possible to align elements over hboxes. The position of items inside an hbox depends on the spacer and the real size of the elements. The following example does not align the items as you would expect, following the character positions in the layout definition:

```
GRID
```

```
{
  ["Num:     " :fnum :        ]
  ["Name:    " :fname         ]
}
END
```

A big advantage in using elements in an hbox is that the fields gets their real sizes according to the .per definition. The following example illustrates the case:

```
GRID
{
 MMMMM
[f1   ]
[f2 : ]
}
END
```

Here all items will occupy the same number of grid columns (5). The MMMMM static label will have the largest width and define the width of the 5 grid cells. The first field is defined with a normal item tag, and expands to the width of the 5 grid cells. The line 5 defines an hbox that will expand to the size of the 5 grid cells, according to the static label, but its child element - the field f2 - gets a size corresponding to the number of characters used before the ':' colon (i.e. 3 characters).

If the default width generated by the form compiler does not fit, the `-` dash symbol can be used to define the real width of the item. In this example, the hbox tag occupies 20 grid cells, the first form item gets a width of 5, and the second form item gets a width of 3:

```
GRID
{
 12345678901234567890
[f1   - :f2 -    :   ]
}
END
```

The `-` dash size indicator is especially useful in `BUTTONEDIT`, `DATEEDIT` and `COMBOBOX` form fields, for which the default width computed by the form compiler may not fit.

In the next example, a static label is positioned above a `TEXTEDIT` field. The label will be centered over the `TEXTEDIT` field, and will remain centered as the field expands or contracts with the resizing of the window.

```
GRID
{
 [ :"label": ]
 [textedit   ]
}
END

ATTRIBUTES
 TEXTEDIT textedit = formonly.textedit, STRETCH=BOTH;
END
```

Form item types

The *form item types* defines the purpose of form elements.

BUTTON item type

Defines a push-button that can trigger an action.

BUTTON item basics

The `BUTTON` form item type defines a standard push button with a label and/or an icon.

Defining a BUTTON

The label of a BUTTON form item is defined with the TEXT attribute. The COMMENT attribute can be used to define a hint for the button. Consider using localized strings for these attributes.

The picture is defined by the IMAGE attribute. Consider using centralized icons for button images.

```
BUTTON ...
   TEXT = %"common.button.text.ok",
   IMAGE = "accept",
   COMMENT = %"common.button.hint.ok";
```

BUTTON form items can inherit action default attributes, to avoid having to specify the TEXT, COMMENT and IMAGE attributes in all elements bound to the same action. For more details, see Configuring actions on page 1318.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and Button style attributes on page 821.

Detecting BUTTON action

A BUTTON form item acts as an action view for a dialog action, and is bound to the ON ACTION handler by name. The action name can be prefixed with a sub-dialog identifier and/or a field name, to define a qualified action view:

```
-- Form file (grid layout)
BUTTON b1: print;

-- Program file:
ON ACTION print
   -- Execute code related to the print action
```

> **Note:** When controlled by a COMMAND action handler in a DIALOG interactive instruction, form buttons can get the focus and thus be part of the tabbing list (TABINDEX attribute).

For more details, see Binding action views to action handlers on page 1278.

Where to use a BUTTON

A BUTTON form item can be defined in two different ways:

1. With an item tag and a BUTTON item definition on page 934 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a BUTTON stack item on page 915 in a STACK container.

BUTTONEDIT item type

Defines a line-edit with a push-button that can trigger an action.

BUTTONEDIT item basics

The BUTTONEDIT form item defines an edit field that get user input, with an additional push button that can fire an action.

This type of form field is typically used to open a secondary window, to let the user choose from a large list of items and set the field value.

Defining a BUTTONEDIT

The IMAGE attribute of a BUTTONEDIT form item defines the picture to be displayed on the button.

By default, the text editor of a BUTTONEDIT allows the user to change the field value. Use the NOTEDITABLE attribute to deny text modification. The field still gets the focus, and the action button remains active, if there is a corresponding action handler in the current dialog.

```
BUTTONEDIT ...
   IMAGE = "zoom",
   NOTEDITABLE;
```

The button of BUTTONEDIT form items can inherit action default attributes, to avoid having to specify the IMAGE attributes in all elements bound to the same action. For more details, see Configuring actions on page 1318.

Most of the attributes described in the EDIT item type on page 886 can also be used with the BUTTONEDIT.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and ButtonEdit style attributes on page 822.

Detecting BUTTONEDIT button action

The button of a BUTTONEDIT form element acts as an action view for a dialog action, and is bound to the ON ACTION handler by the ACTION attribute. The ACTION attribute defines the name of the action to be sent to the program when the user clicks on the button. It can be prefixed with a sub-dialog identifier and/or field name, to define a qualified action view:

```
-- Form file
BUTTONEDIT ...
   ACTION = open_city_list;

-- Program file:
ON ACTION open_city_list
   -- Execute code related to the buttonedit button
```

For more details, see Binding action views to action handlers on page 1278.

Where to use a BUTTONEDIT

A BUTTONEDIT form item can be defined in two different ways:

1. With an item tag and a BUTTONEDIT item definition on page 935 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a BUTTONEDIT stack item on page 916 in a STACK container.

Defining the widget size

In a grid-based layout, the size of a BUTTONEDIT widget is computed according to layout rules as described in Widget size within hbox tags on page 1016.

In a stack-based layout, the widget will take the full width available in the parent container.

Field input length

In grid-based layout, the input length in a BUTTONEDIT fields is defined by the item tag and may need to get the SCROLL attribute. For more details, see Field input length on page 1260.

Defines a boolean or three-state checkbox field.

CHECKBOX item basics

The CHECKBOX form item defines a field with a check box and a text label.

Defining a CHECKBOX

The TEXT attribute defines the label to be displayed near the check box. Consider using localized strings for this attribute.

The box shows a checkmark when the form field contains the value defined in the VALUECHECKED attribute (for example: "Y"), and shows no checkmark if the field value is equal to the value defined by the VALUEUNCHECKED attribute (for example: "N"). If you do not specify the VALUECHECKED or VALUEUNCHECKED attributes, they respectively default to TRUE (integer 1) and FALSE (integer 0).

By default, during an INPUT dialog, a CHECKBOX field can have three states:

- Grayed (NULL value)
- Checked (VALUECHECKED value)
- Unchecked (VALUEUNCHECKED value)

If the field is declared as NOT NULL, the initial state can be grayed if the default value is NULL; once the user has changed the state of the CHECKBOX field, it switches only between checked and unchecked states.

During a CONSTRUCT, a CHECKBOX field always has three possible states (even if the field is NOT NULL), to allow the end user to clear the search condition:

- Grayed (No search condition)
- Checked (Condition column = VALUECHECKED value)
- Unchecked (Condition column = VALUEUNCHECKED value)

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and CheckBox style attributes on page 822.

Detecting CHECKBOX modification

To inform the dialog immediately when the value changes, define an ON CHANGE block for the CHECKBOX field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
CHECKBOX cb1 = order.ord_valid,
   ITEMS = ... ;

-- Program file:
ON CHANGE ord_valid
   -- The checkbox field has been modified
```

For more details, see Reacting to field value changes on page 1267.

Where to use a CHECKBOX

A CHECKBOX form item can be defined in two different ways:

1. With an item tag and a CHECKBOX item definition on page 936 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a CHECKBOX stack item on page 916 in a STACK container.

COMBOBOX item type

Defines a line-edit with a drop-down list of values.

COMBOBOX item basics

The COMBOBOX form item defines a field that can open a list of possible values the end user can choose from.

> **Note:** Such form field should be used for a short list of possible values (10 to 50, maximum).

Defining a COMBOBOX

The values of the drop-down list are defined by the ITEMS attribute. Define a simple list of values like ("A","B","C","D", ...) or a list of key/label pairs like in ((1,"Paris"),(2,"Madrid"), (3,"London")). In the second case, the labels (i.e. the city names) display according to the key value (the city number) held by the field.

```
COMBOBOX ...
   ITEMS=((1,"Paris"),(2,"Madrid"),(3,"London"));
```

Consider using localized strings when defining key/value pairs in the combobox items:

```
COMBOBOX ...
   ITEMS=((1,%"cities.paris"),
          (2,%"cities.madrid"),
          (3,%"cities.london"));
```

The INITIALIZER attribute allows you to define an initialization function for the COMBOBOX. This function will be invoked at runtime when the form is loaded, to fill the item list dynamically, for example with database records. It is recommended that you use the TAG attribute, so you can identify in the program the kind of COMBOBOX form item to be initialized. The initialization function name is converted to lowercase by fglform.

```
COMBOBOX ...
   TAG = "city", INITIALIZER=cmb_init;
```

If neither ITEMS nor INITIALIZER attributes are specified, the form compiler automatically fills the list of items with the values of the INCLUDE attribute, when specified. However, the item list will not automatically be populated with include range values (i.e. values defined using the TO keyword). The INCLUDE attribute can be specified directly in the form or indirectly in the schema files.

```
COMBOBOX ...
   INCLUDE=("A","B","C","D","E");
```

During an input dialog, a COMBOBOX field value can only be one of the values specified in the ITEMS attribute. If the field allows NULL values, consider adding an item to reference the NULL value. However, the best practice is to deny nulls with the NOT NULL attribute, and add a special item such as (0,"<Undefined>") to specify a non-specified-value:

```
COMBOBOX ...
   NOT NULL,
   ITEMS=((0,"<Undefined>"),
          (1,"Red"),
          (2,"Yellow"),
          (3,"Green"));
```

> **Note:** If one of the items is explicitly defined with NULL; In INPUT, selecting the corresponding combobox list item sets the field value to null. In CONSTRUCT, selecting the list item corresponding to null will be equivalent to the = query operator, which will generate a "*colname* is null" SQL

condition. During an CONSTRUCT, a COMBOBOX field gets an additional 'empty' item (even if the field is NOT NULL), to let the user clear the search condition.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and ComboBox style attributes on page 823.

Detecting COMBOBOX item selection

To inform the dialog when the value changes, define an ON CHANGE block for the COMBOBOX field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
COMBOBOX cb1 = customer.cust_city,
   ITEMS = ... ;

-- Program file:
ON CHANGE cust_city
   -- An new item was selected in the combobox list
```

For more details, see Reacting to field value changes on page 1267.

Where to use a COMBOBOX

A COMBOBOX form item can be defined in two different ways:

1. With an item tag and a COMBOBOX item definition on page 937 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a COMBOBOX stack item on page 917 in a STACK container.

Defining the widget size

In a grid-based layout, the size of a COMBOBOX widget is computed according to the SIZEPOLICY and SAMPLE attributes, and according to layout rules as described in Widget size within hbox tags on page 1016.

In a stack-based layout, the widget will take the full width available in the parent container.

COMBOBOX on mobile devices

On a mobile devices, COMBOBOX form items should be used for a short list of values that can be displayed on a single page; for example, 4 to 6 elements. When a list will expand to more than one page, it is recommended that you use a BUTTONEDIT with a zoom, which you can improve with a search button to find an exact item or to reduce the list of items to scroll.

DATEEDIT item type

Defines a line-edit with a calendar widget to pick a date.

DATEEDIT item basics

The DATEEDIT form item defines a field that can open a calendar to ease date input.

To store the field value, use a DATE program variable with this form item.

> **Important:** DATEEDIT fields are dedicated for DATE value input. Some front-ends (especially on mobile devices) deny data types different from DATE. If the front-end does not support the data type used for the DATEEDIT field, the runtime system will raise an error and stop the program. Consider testing your application with all types of front-ends.

Defining a DATEEDIT

The DATEEDIT form item type allows the user to edit date values with a specific widget for date input. A DATEEDIT field typically provides a calendar widget, to let the end user pick a date from it.

When using a DATE variable as recommended, with desktop front-ends, the format of DATEEDIT fields is by default defined by the DBDATE environment variable as for other editor fields. Specific format can be defined with the FORMAT attribute, but it is recommended to use the default date formatting. On mobile platforms, the date format is defined by the device OS language settings.

On some front-end platforms, the native widget used for DATEEDIT fields allows only pure date value input, and therefore cannot be used with a CONSTRUCT instruction, where it must be possible to enter search filters like ">=24/03/2014".

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and DateEdit style attributes on page 824.

Detecting DATEEDIT calendar selection

To inform the dialog when a date is picked from the calendar widget, define an ON CHANGE block for the DATEEDIT field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
DATEEDIT de1 = order.ord_shipdate,
   NOT NULL;

-- Program file:
ON CHANGE ord_shipdate
   -- An new date value was picked from the calendar
```

For more details, see Reacting to field value changes on page 1267.

Where to use a DATEEDIT

A DATEEDIT form item can be defined in two different ways:

1. With an item tag and a DATEEDIT item definition on page 938 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a DATEEDIT stack item on page 917 in a STACK container.

Defining the widget size

In a grid-based layout, the size of a DATEEDIT widget is computed according to layout rules as described in Widget size within hbox tags on page 1016.

In a stack-based layout, the widget will take the full width available in the parent container.

Field input length

The input length in a DATEEDIT fields is defined by the (DATE) program variable. In a grid-based layout, define an item tag with 10 positions, to be able to display dates with 4 year digits. For more details, see Field input length on page 1260.

DATETIMEEDIT item type

Defines a line-edit with a calendar widget to pick a datetime.

DATETIMEEDIT item basics

The DATETIMEEDIT form item defines a field that can open a calendar to ease date-time input.

To store the field value, use a `DATETIME YEAR TO MINUTE` or `DATETIME YEAR TO SECOND` program variable with such form item.

> **Important:** `DATEEDIT` fields are dedicated for `DATETIME` value input. Some front-ends (especially on mobile devices) deny data types different from `DATE`. If the front-end does not support the data type used for the `DATEEDIT` field, the runtime system will raise an error and stop the program. Consider testing your application with all types of front-ends.

Defining a DATETIMEEDIT

The `DATETIMEEDIT` form item type allows the user to edit date-time values with a specific widget for date-time input. A `DATETIMEEDIT` field typically provides a calendar and clock widget, to let the end user pick a date and time from it.

The display and input precision (time part with or without seconds) of the `DATETIMEEDIT` widget depends from the front-end. On some platforms, native date-time editors do not handle the seconds. Further, some front-ends (especially on mobile devices) deny data types different from `DATETIME YEAR TO {MINUTE|SECOND}`.

On some front-end platforms, the native widget used for `DATETIMEEDIT` fields allows only pure date-time value input, and therefore cannot be used with a `CONSTRUCT` instruction, where it must be possible to enter search filters like "`>= 2014-01-23 11:00`".

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818.

Detecting DATETIMEEDIT calendar selection

To inform the dialog when a date-time is picked from the calendar widget, define an `ON CHANGE` block for the `DATETIMEEDIT` field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
DATETIMEEDIT dt1 = order.ord_shipdate,
   NOT NULL;

-- Program file:
ON CHANGE ord_shipdate
   -- An new date-time value was picked from the calendar
```

For more details, see Reacting to field value changes on page 1267.

Where to use a DATETIMEEDIT

A `DATETIMEEDIT` form item can be defined in two different ways:

1. With an item tag and a DATETIMEEDIT item definition on page 939 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a DATETIMEEDIT stack item on page 918 in a `STACK` container.

Defining the widget size

In a grid-based layout, the size of a `DATETIMEEDIT` widget is computed according to layout rules as described in Widget size within hbox tags on page 1016.

In a stack-based layout, the widget will take the full width available in the parent container.

Field input length

The input length in a `DATETIMEEDIT` fields is defined by the (`DATETIME`) program variable. In a grid-based layout, define an item tag with enough positions, to be able to display dates with 4 year digits. For more details, see Field input length on page 1260.

EDIT item type

Defines a simple line-edit field.

EDIT item basics

The EDIT form item defines a field to enter a single-line text, for any data type.

This item type is typically used to define character string and numeric form fields.

Defining an EDIT

The EDIT item type can be used for any data type that can be converted to an editable text.

To show a hint to the user when the field has the focus, use the COMMENT attribute.

If the field is mandatory for an input, combine the NOT NULL with the REQUIRED attribute.

The value accepted for the field can be limited with the INCLUDE attribute.

To provide a default value, define the DEFAULT attribute for the field.

Use the DOWNSHIFT or UPSHIFT attributes to force the letter case during input.

Input can be hidden (for example for password fields), with the INVISIBLE attribute.

Especially for mobile devices, use the KEYBOARDHINT attribute to get a specific keyboard when entering values into the field.

Input completion proposals can be implemented with the COMPLETER attribute.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and Edit style attributes on page 826.

Where to use an EDIT

An EDIT form item can be defined in two different ways:

1. With an item tag and a EDIT item definition on page 939 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a EDIT stack item on page 918 in a STACK container.

Field input length

In grid-based layout, the input length in an EDIT fields is defined by the item tag and may need to get the SCROLL attribute. For more details, see Field input length on page 1260.

FOLDER item type

Defines a layout area to hold folder pages.

FOLDER item basics

A FOLDER form item type groups folder pages together. Folder pages are defined with the PAGE form item.

Defining an FOLDER

The FOLDER form item is just a container for PAGE items.

```
FOLDER ...
   PAGE ...
      ...
   PAGE ...
      ...
```

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818.

Where to use a FOLDER

A FOLDER form item can be defined in two different ways:

1. In as a FOLDER container in a LAYOUT tree, within a grid-based layout.
2. As a FOLDER stack item, inside a STACK container, within a stack-based layout.

GRID item type

Defines a layout area based on a grid of cells.

GRID item basics

A GRID form item defines an area in the layout section to place children form items by X,Y position in layout cells.

Defining an GRID

The GRID container declares a formatted text block defining the dimensions and the positions of the form items contained in the grid.

You can specify the position of labels, form fields for data entry or additional interactive objects such as buttons.

A GRID container can hold static text, item tags, field tags, hbox tags, and layout tags to define other containers such as TABLE, TREE and SCROLLGRID.

A GRID can hold form items such as labels, fields, or buttons at a specific position. Form items are located with item tags in the grid layout area. You can use layout tags to place some type of containers inside a grid.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and Grid style attributes.

Where to use a GRID

A GRID form item can only be defined as a GRID container in a LAYOUT tree.

GRID layout definition

For more details about grid layout concept, see Grid-based layout on page 1004.

GROUP item type

Defines a layout area to group other layout elements together.

GROUP item basics

A GROUP form item type groups other form items together, typically in a groupbox widget.

Defining an GROUP

The GROUP form item typically gets a TEXT attribute, to define the title of the group. Consider using localized strings for this attribute:

```
GROUP ...
  TEXT=%"customer.info";
```

Note: Some front-ends render group containers as a groupbox widget, displaying a title on the top of the child elements, while other front-ends may not show a group title.

Consider identifying group elements with a name, in order to manipulate the group during program execution. For example use the `ui.Form.setElementHidden()` method to hide or show groups in a form:

```
GROUP g1: g_cust_info, ... ;   -- grid-based layout
GROUP g_cust_info, ... ;       -- stack-based layout
```

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818 and Group style attributes.

Groups in grid-based layout

In a `LAYOUT` tree with `GROUP` containers, if you want to include several children in a `GROUP`, you can add a `VBOX` or `HBOX` into the `GROUP`, to define how these form items are aligned.

> **Note:** When defining a `GROUP` container, you cannot set the `GRIDCHILDRENINPARENT` attribute. This attribute makes sense only for a group item defined with a layout tag contained in a `GRID` area.

Consider using a group layout tag inside a `GRID` container, this layout specification technique is often more appropriate to define forms:

```
GRID
{
<G g1              ><G g2        >
[l1  :f1           ][f4          ]
 ...
<G g3                             >
 ...
```

Groups in stack-based layout

In a `STACK` container, `GROUP` form items are one of the base concepts used to put stack items together. For more details see Stacked group rendering on page 1019.

Groups on mobile devices

On mobile devices, groups render according to the platform standards:

- With GMA/Android™, groups are visualized by a simple separator under the group title. Complex layout construction is supported: groups in groups, groups in a grid, and so on.
- With GMI/iOS, the layout is limited by the platform GUI standards. The only visible grouping container element is a group. Groups within groups are not allowed. GMI enforces each form item as a member in a group. There can be group headers and footers, but no elements in between groups.

Where to use a GROUP

A `GROUP` form item can be defined in three different ways:

1. As a `GROUP` container in a `LAYOUT` tree, within a grid-based layout.
2. As a `<GROUP >` layout tag with a `GROUP` item definition in the `ATTRIBUTES` section, within a grid-based layout.
3. As a `GROUP` stack item, inside a `STACK` container, within a stack-based layout.

IMAGE item type

Defines a area that can display an image resource.

IMAGE item basics

The `IMAGE` item type defines an area where a picture resource can be displayed.

Defining an IMAGE

An IMAGE form item can be defined as a form field image or as a static image. Use a form field image when the content of the image will change often during program execution (for example, to display images from the database). Use a static image if the image remains the same during program execution.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and Image style attributes on page 828.

Form field IMAGE item

Use a form field image item to display values that change often during program execution, for example if the image is stored in the database.

The picture resource is defined by the value of the field.

The value can be changed from the program by using the DISPLAY BY NAME / DISPLAY TO instruction, or just by changing the value of the program variable bound to the image field when using the UNBUFFERED mode in an interactive instruction.

When defining the IMAGE item in the form, use a field name to identify the element in programs:

```
-- Grid-based layout (ATTRIBUTES item definition)
IMAGE f001 = cars.picture, SIZEPOLICY=FIXED, AUTOSCALE;

-- Stack-based layout (STACK item)
IMAGE cars.picture, SIZEPOLICY=FIXED, AUTOSCALE;
```

Static IMAGE item

Use a static image item to display an image that does not change during program execution, such as form decoration pictures and logos.

The resource of the image is defined by the IMAGE attribute; the item is not a form field. This kind of item is not affected by instructions such as CLEAR FORM or the DISPLAY TO instruction.

```
-- Grid-based layout (ATTRIBUTES item definition)
IMAGE img1: logo, IMAGE="fourjs.png", STRETCH=BOTH;

-- Stack-based layout (STACK item)
IMAGE logo, IMAGE="fourjs.png", AUTOSCALE;
```

Providing the image resource

To display an image, the front-end needs the image data, which can be provided in different ways.

For example, you can specify an URL, a mapped icon, or a plain image file (centralized on the application server).

For more details about image resource specification, see Providing the image resource on page 784.

Detecting IMAGE clicks

To inform the dialog immediately when an image was clicked, define the ACTION attribute in the IMAGE item, and implement the corresponding ON ACTION handler in the dialog:

```
-- Form file (grid layout)
IMAGE : logo, IMAGE="fourjs.png",
   ACTION=show_about;

-- Program file:
```

```
ON ACTION show_about
   -- The image was clicked
```

The program can then react immediately when the user selects the image element.

Where to use a IMAGE

A IMAGE form item can be defined in two different ways:

1. With an item tag and a IMAGE item definition on page 941 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a IMAGE stack item on page 920 in a STACK container.

Defining the widget size

The size of an IMAGE widget can be controlled in grid-based or stack-based layout, according to several attributes such as SIZEPOLICY, AUTOSCALE and STRETCH.

For more details about image sizing, see Controlling the image layout on page 783.

LABEL item type

Defines a simple text area to display a read-only value.

LABEL item basics

The LABEL form item defines a read-only text area.

Defining a LABEL

A LABEL form item can be defined as a form field image or as a static label. Use a form field label when the text changes often during program execution (for example, to display text from the database). Use a static label if the text remains the same during program execution.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and Label style attributes on page 828.

Form field LABEL item

Use a form field label item to display values that change often during program execution, for example if the text is stored in the database.

The label text is defined by the value of the field.

The value can be changed from the program by using the DISPLAY BY NAME / DISPLAY TO instruction, or just by changing the value of the program variable bound to the label field when using the UNBUFFERED mode in an interactive instruction.

When defining the LABEL item in the form, use a field name to identify the element in programs:

```
-- Grid-based layout (ATTRIBUTES item definition)
LABEL f001 = cars.description;

-- Stack-based layout (STACK item)
LABEL cars.description;
```

Static LABEL item

Use a static label item to display a text that does not change during program execution.

This kind of item is not affected by instructions such as `CLEAR FORM` or the `DISPLAY TO` instruction.

```
-- Grid-based layout (ATTRIBUTES item definition)
LABEL lab1: label1, TEXT="Name:";

-- Stack-based layout (STACK item)
LABEL label1, TEXT="Name:";
```

Consider using localized strings to ease application internationalization:

```
LABEL ...
   TEXT = %"label.customer.name";
```

Static labels display only character text values, and therefore do not follow any justification rule as form field labels.

Multi-line text in LABELs

In order to display label text on several lines, the text must contain `\n` line-feed characters:

```
LABEL lab1: label1,
      TEXT="First line.\nSecond line.";
```

Where to use a LABEL

A `LABEL` form item can be defined in two different ways:

1. With an item tag and a LABEL item definition on page 942 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a LABEL stack item on page 921 in a `STACK` container.

PAGE item type

Defines the content of a folder page.

PAGE item basics

A `PAGE` form item type groups other form elements together, to define a folder page of a parent `FOLDER` form item.

Defining an PAGE

A `PAGE` form item can only be a child of a `FOLDER` form item.

By default, `PAGE` form items are used to group elements for decoration only.

> **Note:** Use the `TABINDEX` attribute of form fields inside the folder page, to define which field gets the focus when a folder page is selected.

The `TEXT` attributes defines the label of the folder page. Consider using localized strings for this attribute.

The `IMAGE` attribute can be used to specify which image to use as an icon.

Detecting folder page selection

If needed, you can use the `ACTION` attribute to bind an action to a folder page. When the page is selected, the program gets the corresponding action event.

> **Note:** This feature should be used with care: It exists to implement different singular dialog statements (such as `INPUT`) in each folder page. You should use a `DIALOG` statement to control all folder pages simultaneously.

Bring a folder page to the top

To bring a folder page to the top, use the `NEXT FIELD` program instruction to give the focus to one of the active fields of the page, or use the `ui.Form.ensureFieldVisible()` method if the fields are disabled/unused, or use the `ui.Form.ensureElementVisible()` method if the page does not contain focusable form items.

For more details, see Giving the focus to a form element on page 1272.

Where to use a PAGE

A `PAGE` form item can be defined in two different ways:

1. As a `PAGE` container in a `LAYOUT` tree, within a grid-based layout.
2. As a `PAGE` stack item, inside a `STACK` container, within a stack-based layout.

PROGRESSBAR item type

Defines a progress indicator field.

PROGRESSBAR item basics

The `PROGRESSBAR` form item defines a field that shows a progress indicator.

> **Note:** Use a `SMALLINT` or `INTEGER` variable with a `PROGRESSBAR` form item. Larger types like `BIGINT` or `DECIMAL` are not supported.

Defining a PROGRESSBAR

The `VALUEMIN` and `VALUEMAX` attributes define respectively the lower and upper integer limit of the progress information. Any value outside this range will not be displayed. Default values are `VALUEMIN=0` and `VALUEMAX=100`.

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818 and ProgressBar style attributes on page 830.

Displaying PROGRESSBAR values

The position of the progress bar indicator is defined by the value of the corresponding form field. The value can be changed from the program by using the `DISPLAY TO` instruction, to set the value of the field, or by changing the program variable bound to the field when using the `UNBUFFERRED` dialog mode.

Progress information is typically displayed during non-interactive program code. To show changes to the end user in this context, you need to use the `ui.Interface.refresh()` method to force a refresh. To provide the best feedback to the user, consider calling the `refresh()` method regularly but not to often, otherwise you will overload the network traffic and bring down the front-end component.

For example, if you have to process 1000 rows, define `VALUEMIN=0` and `VALUEMAX=1000` in the `PROGRESSBAR` item, and perform a refresh every 50 rows:

```
FOR row=1 TO 1000
    ...
    IF (row MOD 50) == 0 THEN
       LET myprogbar = row
       CALL ui.Interface.refresh()
    END IF
END FOR
```

Where to use a PROGRESSBAR

A `PROGRESSBAR` form item can be defined in two different ways:

1. With an item tag and a PROGRESSBAR item definition on page 942 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a PROGRESSBAR stack item on page 923 in a `STACK` container.

RADIOGROUP item type

Defines a mutual exclusive set of options field.

RADIOGROUP item basics

The `RADIOGROUP` form item defines a field that provides several options that the user can make a selection from. Checking one radio button unchecks any previously checked button within the same group.

Defining a RADIOGROUP

A `RADIOGROUP` defines a set of radio buttons where each button is associated with a value defined in the `ITEMS` attribute.

The text associated with each item value will be used as the label of the corresponding radio button, for example: `ITEMS=((1,"Beginner"), (2,"Normal"), (3,"Expert"))` will create three radio buttons with the texts `Beginner`, `Normal` and `Expert`, respectively.

```
RADIOGROUP ...
   ITEMS=((1,"Beginner"),(2,"Normal"),(3,"Expert"));
```

Consider using localized strings when defining key/value pairs in the radiogroup items:

```
COMBOBOX ...
   ITEMS=((1,%"skills.beginner"),
          (2,%"skills.normal"),
          (3,%"skills.expert"));
```

If the `ITEMS` attribute is not specified, the form compiler automatically fills the list of items with the values of the `INCLUDE` attribute, when specified. However, the item list will not automatically be populated with include range values (i.e. values defined using the `TO` keyword). The `INCLUDE` attribute can be specified directly in the form or indirectly in the schema files.

During an `INPUT`, a `RADIOGROUP` field value can only be one of the values specified in the `ITEMS` attribute. During an `CONSTRUCT`, a `RADIOGROUP` field allows to uncheck all items (even if the field is `NOT NULL`), to let the user clear the search condition.

If one of the items is explicitly defined with `NULL` and the `NOT NULL` attribute is omitted: In `INPUT`, selecting the corresponding radio button sets the field value to null. In `CONSTRUCT`, selecting the radio button corresponding to null will be equivalent to the = query operator, which will generate a "*`colname`* `is null`" SQL condition.

Use the `ORIENTATION` attribute to define if the radio group must be displayed vertically or horizontally:

```
COMBOBOX ...
   ITEMS=(...),
   ORIENTATION = HORIZONTAL;
```

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818 and RadioGroup style attributes on page 831.

Detecting RADIOGROUP item selection

To inform the dialog when a value change, define an `ON CHANGE` block for the `RADIOGROUP` field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
RADIOGROUP rg1 = user.user_skill,
   ITEMS = ... ;

-- Program file:
ON CHANGE user_skill
   -- An new item was selected in the radiogroup
```

For more details, see Reacting to field value changes on page 1267.

Where to use a RADIOGROUP

A `RADIOGROUP` form item can be defined in two different ways:

1. With an item tag and a RADIOGROUP item definition on page 943 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a RADIOGROUP stack item on page 923 in a `STACK` container.

SCROLLGRID item type

Defines a scrollable grid view widget.

SCROLLGRID item basics

A `SCROLLGRID` form item type defines a grid to show a scrolling list of data records in a set of positioned form fields.

Defining an SCROLLGRID

The `SCROLLGRID` form item declares a formatted text block defining the dimensions and the position of the logical elements of a screen for a multi-record presentation.

A `SCROLLGRID` is similar to the `GRID`, except that you can only specify form fields, that repeat on several "row-templates", in order to design a multiple-record view that appears with a vertical scrollbar.

> **Note:** When using a `SCROLLGRID` container, you cannot set the `GRIDCHILDRENINPARENT` attribute. This attribute makes sense only for a scrollgrid defined with a layout tag contained in a `GRID` area.

The same layout rules apply as in a `GRID` container.

By default, a `SCROLLGRID` container is not resizable in height. Use the `WANTFIXEDPAGESIZE=NO` attribute to allow the scrollgrid to stretch vertically.

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818.

Where to use a SCROLLGRID

Within a grid-based layout, a `SCROLLGRID` form item can be defined in two different ways:

1. As a `SCROLLGRID` container in a `LAYOUT` tree.
2. As a `<SCROLLGRID >` layout tag with a `SCROLLGRID` item definition in the `ATTRIBUTES` section.

SCROLLGRID view programming

A `SCROLLGRID` is similar to a `TABLE` form item in terms of list programming. For more details about list view programming, see Table views on page 1345.

SLIDER item type

Defines a slider form item.

SLIDER item basics

The `SLIDER` form item defines a field where the user can set a value in a given range, such as a typical audio volume control widget where you can grab the slider handle to change the value.

Use a `SMALLINT` or `INTEGER` variable with a `SLIDER` form item, larger types like `BIGINT` or `DECIMAL` are not supported.

Defining a SLIDER

A `SLIDER` field lets the user move a handle along a horizontal or vertical groove and translates the handle's position into a value within the legal range.

The `VALUEMIN` and `VALUEMAX` attributes define respectively the lower and upper integer limit of the slider information. Any value outside this range will not be displayed. The step between two marks is defined by the `STEP` attribute. If `VALUEMIN` and/or `VALUEMAX` are not specified, they default respectively to 0 (zero) and 5.

The `ORIENTATION` attribute defines whether the `SLIDER` is displayed vertically or horizontally.

This item type is not designed for `CONSTRUCT`, as the user can only select one value.

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818.

Detecting SLIDER item selection

To inform the dialog when a value changes, define an `ON CHANGE` block for the `SLIDER` field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
SLIDER s1 = options.opts_volume,
   VALUEMIN=0, VALUEMAX=100;

-- Program file:
ON CHANGE opts_volume
   -- An value changed in the slider
```

For more details, see Reacting to field value changes on page 1267.

Where to use a SLIDER

A `SLIDER` form item can be defined in two different ways:

1. With an item tag and a SLIDER item definition on page 945 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a SLIDER stack item on page 924 in a `STACK` container.

SPINEDIT item type

Defines a spin box widget to enter integer values.

SPINEDIT item basics

The `SPINEDIT` form item defines a field where the users can increase or decrease the number value by a specific increment by clicking an up or down arrow button, or by typing the value directly into the text edit box.

Use a `SMALLINT` or `INTEGER` variable with a `SLIDER` form item. Larger types like `BIGINT` or `DECIMAL` are not supported.

Defining a SPINEDIT

The increment between two values is defined by the STEP attribute:

```
SPINEDIT ...
   STEP = 5;
```

The VALUEMIN and VALUEMAX attributes define respectively the lower and upper integer limit of the spin-edit range. There is no default minimum or maximum value for the SPINEDIT widget.

This widget is not designed for CONSTRUCT, as you can only enter an integer value.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818.

Detecting SPINEDIT modification

To inform the dialog when a value changes, define an ON CHANGE block for the SPINEDIT field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
SLIDER s1 = options.opts_rate,
   VALUEMIN=0, VALUEMAX=100;

-- Program file:
ON CHANGE opts_rate
   -- The value of the spinedit has changed
```

For more details, see Reacting to field value changes on page 1267.

Where to use a SPINEDIT

A SPINEDIT form item can be defined in two different ways:

1. With an item tag and a SPINEDIT item definition on page 945 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a SPINEDIT stack item on page 924 in a STACK container.

TABLE item type

Defines a list view widget.

TABLE item basics

A TABLE form item type defines a listview to show a scrolling list of data records in a set of columns.

Defining an TABLE

The TABLE form item defines list view widget to show a set of data records, bound to a screen array.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and Table style attributes on page 831.

Where to use a TABLE

A TABLE form item can be defined in three different ways:

1. As a TABLE container in a LAYOUT tree, within a grid-based layout.
2. As a <TABLE > layout tag with a TABLE item definition in the ATTRIBUTES section, within a grid-based layout.
3. As a TABLE stack item, inside a STACK container, within a stack-based layout.

TABLE view programming

For more details about table view programming, see Table views on page 1345.

TEXTEDIT item type
Defines a multi-line edit field.

TEXTEDIT item basics

The TEXTEDIT form item defines a text input field with multiple lines. This type of element is typically used to handle large text values such as comments or addresses that would not fit in a single-line edit field.

Use a VARCHAR(N) or STRING variable to hold the data for a TEXTEDIT form item.

Defining a TEXTEDIT

Use the SCROLLBARS attribute to define vertical and/or horizontal scrollbars for the TEXTEDIT form field. By default, this attribute is set to VERTICAL for TEXTEDIT fields.

The STRETCH attribute can be used to force the TEXTEDIT field to stretch when the parent container is re-sized. Values can be NONE, X, Y or BOTH. By default, this attribute is set to NONE for TEXTEDIT fields.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818 and TextEdit style attributes on page 834.

TAB and RETURN

By default, when the focus is in a TEXTEDIT field, the Tab key moves to the next field, while the Return key adds a newline (ASCII 10) character in the text.

To control the user input when the Tab and Return keys are pressed, specify the WANTTABS and WANTNORETURNS attributes.

With WANTTABS, the Tab key is consumed by the TEXTEDIT field, and a Tab character (ASCII 9) is added to the text. The user can still jump out of the field with the Shift-Tab combination.

With WANTNORETURNS, the Return key is not consumed by the TEXTEDIT field, and the action corresponding to the Return key is triggered. The user can still enter a newline character with Shift-Return or Ctrl-Return.

Where to use a TEXTEDIT

A TEXTEDIT form item can be defined in two different ways:

1. With an item tag and a TEXTEDIT item definition on page 947 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a TEXTEDIT stack item on page 925 in a STACK container.

Defining the widget size

In a grid-based layout, the layout behavior of the TEXTEDIT widget can be controlled with the STRETCH and SCROLLBARS attributes.

In a stack-based layout, the TEXTEDIT widget always adapts to the field value to avoid scrollbars. You can control the minimum height of the TEXTEDIT widget by using the HEIGHT attribute. If the field content is null and the HEIGHT attribute is not defined, the minimum size defaults to one line.

Field input length

By default, the input length in an TEXTEDIT fields is defined by the program variable. There is no need to define the SCROLL attribute, except if you explicitly specify SCROLLBARS=NONE (in a grid-based layout).

For more details about the SCROLL attribute, see Field input length on page 1260.

Rich Text HTML support

Some front-ends can also support different text formattings, according to a style attribute. You can for example display and input HTML content in a TEXTEDIT with the Genero Desktop Client. When this feature is enabled, the TEXTEDIT support rich text editing. Depending on the front-end, different formatting options are available (bold, font size, and so on) and can be controlled using either an integrated toolbox or via local actions.

Note:

- Each front-end uses its own technology to provide HTML support in TEXTEDIT fields. The HTML representation may vary between front-ends. As a result, the same HTML content may display in a different way on another front-end.
- When using rich text, the FGL_DIALOG_SETCURSOR() and FGL_DIALOG_SETSELECTION() functions must be called carefully. Because of the rich text format, having a corresponding cursor position / selection between displayed text and HTML representation may be difficult, especially in the case of hidden parts.

TIMEEDIT item type

Defines a line-edit with a clock widget to pick a time.

TIMEEDIT item basics

The TIMEEDIT form item defines a field that allows the user to edit 24H time values, or time duration (intervals), with a specific clock widget for time input.

To store TIMEEDIT field values, consider using the appropriate DATETIME HOUR TO MINUTE or DATETIME HOUR TO SECOND data type according to the target front-end.

Important: The display and input precision (with or without seconds) of the TIMEEDIT widget depends from the front-end. On some platforms, native time editors do not handle the seconds. Further, some front-ends (especially on mobile devices) deny data types different from DATETIME HOUR TO {MINUTE|SECOND}. If the front-end does not support the data type used for the TIMEEDIT field, the runtime system will raise an error and stop the program. Consider testing your application with all types of front-ends.

On some front ends, TIMEEDIT fields can also be used to handle INTERVAL values of the class HOUR TO {MINUTE|SECOND}, in order to input a time duration. Note however that in most cases the time interval pickers are limited to 24H hours and allow only positive values. As result, not all values allowed in an INTERVAL HOUR TO MINUTE variable (such as -86 hours 23 minutes) can be displayed by such widgets.

Defining a TIMEEDIT

No specific attribute is needed to define the rendering and behavior of a TIMEEDIT field. Common data validation attributes such NOT NULL, REQUIRED, DEFAULT are allowed.

Note: The time display format is automatically taken from the front-end platform settings. For example, time values can display in the 0-12 hour clock format (with AM/PM indicators), or in the 0-24 hour clock format.

The native widget used for TIMEEDIT fields usually allows only exact time value input, and therefore cannot be used with a CONSTRUCT instruction, where it must be possible to enter search filters like ">=11:00".

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818.

Detecting TIMEEDIT modification

To inform the dialog when a date is picked from the clock widget, define an `ON CHANGE` block for the `TIMEEDIT` field. The program can then react immediately to user changes in the field:

```
-- Form file (grid layout)
TIMEEDIT de1 = order.ord_shiptime,
   NOT NULL;

-- Program file:
ON CHANGE ord_shiptime
   -- An new time value was picked from the clock widget
```

For more details, see Reacting to field value changes on page 1267.

Where to use a TIMEEDIT

A `TIMEEDIT` form item can be defined in two different ways:

1. With an item tag and a TIMEEDIT item definition on page 947 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a TIMEEDIT stack item on page 926 in a `STACK` container.

Field input length

The input length in a `TIMEEDIT` fields is defined by the (`DATETIME`) program variable. In a grid-based layout, define an item tag with enough positions, to be able to display all time value digits (5 positions for HH:MM, 7 positions for HH:MM:SS). For more details, see Field input length on page 1260.

TREE item type

Defines a tree view widget.

TREE item basics

A `TREE` form item type defines a treeview to show a structured tree of data records with an optional a set of columns.

Defining an TREE

The `TREE` form item defines tree view widget to show a structured set of data records, bound to a screen array.

Tree view definitions are very similar to regular `TABLE` elements; before reading further about tree views, you should be familiar with table elements.

The first column in the `TREE` must be the field defining the text of the tree-view nodes.

The screen array definition must have exactly the same number of columns as the `TREE` form item.

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818 and Table style attributes on page 831.

Where to use a TREE

In a grid-based layout, a `TREE` form item can be defined in two different ways:

1. As a TREE container in a `LAYOUT` tree.
2. As a `<TREE >` layout tag with a TREE item definition in the `ATTRIBUTES` section.

TREE view programming

For more details about tree view programming, see Tree views on page 1384.

WEBCOMPONENT item type
Defines a specialized form item that holds an external component.

WEBCOMPONENT item basics

The `WEBCOMPONENT` form item defines a form field that will hold an external component, implemented with a front-end plug-in mechanism.

This topic describes the `WEBCOMPONENT` item type in form definition files, a complete section is dedicated to web component programming.

Defining a WEBCOMPONENT

The `COMPONENTTYPE` attribute identifies gICAPI external objects to be used for the field. The `PROPERTIES` attribute is typically used to define attributes that are specific to a given gICAPI-based web component. For example, a chart component might have properties to define x-axis and y-axis labels. For more details, see Using a gICAPI web component on page 1422.

If the `COMPONENTTYPE` attribute is not used, the web component will be a URL-based web component. For more details, see Using a URL-based web component on page 1419.

Some front-ends support different presentation and behavior options, which can be controlled by a `STYLE` attribute. For more details, see Common style attributes on page 818.

Where to use a WEBCOMPONENT

A `WEBCOMPONENT` form item can be defined in two different ways:

1. With an item tag and a WEBCOMPONENT item definition on page 949 in a grid-layout container (`GRID`, `SCROLLGRID` and `TABLE`).
2. As a WEBCOMPONENT stack item on page 927 in a `STACK` container.

Defining the widget size

The size of a `WEBCOMPONENT` widget can be controlled in grid-based or stack-based layout, according to several attributes such as `SIZEPOLICY` and `STRETCH`.

For more details about image sizing, see Controlling the web component layout on page 1418.

External form inclusion
Form inclusion allows to reuse the same form part in different forms.

In some cases, application forms can become very complex, or can have a common layout part that repeats across forms. In such case, some parts of the form can be defined in a external .per file, that will be included in the final forms by using the `FORM` clause inside the `LAYOUT` section.

Further, the external form parts can be controlled by a declarative dialog instruction that can be attached to any procedural dialog instruction, with the `SUBDIALOG` clause of `DIALOG`.

```
LAYOUT
  VBOX
    GRID g1
    {
       Customer information
       Name:  [f001                ]
       ...
    }
    END
    FORM "orders"
```

```
  END
END
```

Boolean expressions in forms
Some form item definitions can include boolean expressions with a form file specific syntax.

Syntax

```
[(] bool-expr {AND|OR} bool-expr [)] [...]
```

where *bool-expr* is:

```
[NOT]
{ field-tag
    {  =  expression
    |  <> expression
    |  != expression
    |  <= expression
    |  >= expression
    |  <  expression
    |  >  expression
    |  IS [NOT] NULL
    |  [NOT] BETWEEN expression AND expression
    |  [NOT] MATCHES "string"
    |  [NOT] LIKE "string"
    }
}
```

1. *field-tag* is the name of the current field tag in form line with the attribute definition.
2. *expression* can be the a character string, numeric or date-time literal.

Usage

Some form specification file attributes such as `COLOR WHERE` require a boolean expression. These boolean expressions are different from the language boolean expressions, and have a limited syntax which is specific to the form files.

When a *field-tag* is used in the boolean expression, the runtime system replaces *field-tag* at runtime with the current value in the screen field and evaluates the expression.

Example

```
EDIT f001 = item.price,
     COLOR=RED
          WHERE f001 >= 100 AND f001 < 1000;
```

Form file structure
A form specification file is defined by a set of sections.

The sections of a form specification file must appear in the following order:

1. SCHEMA section on page 902
2. ACTION DEFAULTS section on page 903
3. TOPMENU section on page 903
4. TOOLBAR section on page 905
5. TABLES section on page 931
6. LAYOUT section on page 907
7. ATTRIBUTES section on page 932

8. INSTRUCTIONS section on page 950

Each section must begin with the keyword for which it is named, only the LAYOUT section is mandatory.

SCHEMA section
Defines the database schema file to be used to compile the form.

Each form specification file can begin with a SCHEMA section identifying the database schema (if any) on which the form is based. This can be any database schema that is defined with a database schema file. Form field data types can be automatically extracted from the schema file if you specify the table and column name in the form field definition (see ATTRIBUTES section).

Syntax 1

```
SCHEMA { database[@dbserver] | string | FORMONLY }
```

1. *database* is the name of the database schema to be used for the form compilation.
2. *dbserver* identifies the Informix® database server (INFORMIXSERVER).
3. *string* can be a string literal containing the database name.

Syntax 2: (supported for backward compatibility)

```
DATABASE {  database[@dbserver] | string | FORMONLY } [ WITHOUT NULL INPUT ]
```

The DATABASE syntax is supported for compatibility with Informix® 4gl; using SCHEMA is recommended.

1. *database* is the name of the database schema to be used for the form compilation.
2. *dbserver* identifies the Informix® database server (INFORMIXSERVER)
3. *string* can be a string literal containing the database name.

Usage

The SCHEMA (or DATABASE) defines the database schema to be used to resolve data types for database column-based fields.

> **Note:** The DATABASE instruction is supported for backward compatibility, we recommend using SCHEMA instead.

The SCHEMA section must appear in the sequence described in form file structure.

The SCHEMA section is optional; if you do not specify it, database schema specification defaults to SCHEMA FORMONLY.

You can create a form that is not related to any database schema by using the FORMONLY keyword after SCHEMA/DATABASE. When using this option, you must omit the TABLES section and define field data types explicitly in the ATTRIBUTES section.

The *database* and *dbserver* specifications are supported (but ignored) for backward compatibility with Informix® form specifications.

When using a specific database schema, the field data types are taken from the schema file **during compilation**. Make sure that the database schema file of the development database corresponds to the production database; otherwise the form fields defined in the compiled version of your forms will not match the table structures of the production database.

The use of the WITHOUT NULL INPUT option in the DATABASE syntax is supported for backward compatibility, but is ignored.

Example

```
SCHEMA stores
```

```
LAYOUT
  ...
```

ACTION DEFAULTS section

The `ACTION DEFAULTS` section defines local action view default attributes for the form elements.

Syntax

```
ACTION DEFAULTS
   ACTION action-identifier ( action-attribute [,...] )
   [...]
END
```

1. *action-identifier* defines the name of the action.
2. *action-attribute* defines an attribute for the action.

Attributes

`ACCELERATOR`, `ACCELERATOR2`, `ACCELERATOR3`, `ACCELERATOR4`, `DEFAUTVIEW`, `COMMENT`, `CONTEXTMENU`, `IMAGE`, `TEXT`, `VALIDATE`.

Usage

The `ACTION DEFAULTS` section centralizes action view attributes (text, comment, image, accelerators) at the form level.

The `ACTION DEFAULTS` section must appear in the sequence described in form file structure.

The `ACTION DEFAULTS` section is optional.

The section holds a list of `ACTION` elements and specify attributes for each action. The action is identified by the name following the `ACTION` keyword, and attributes are specified in a list between parenthesis.

The attributes defined in this section are applied to form action views like buttons, toolbar buttons, or topmenu options, if the individual action views do not explicitly define their own attributes.

Action attributes can be defined at different levels, see action configuration for more details.

Example

```
ACTION DEFAULTS
  ACTION accept ( COMMENT="Commit order record changes",
                  CONTEXTMENU=NO )
  ACTION cancel ( TEXT="Stop", IMAGE="stop",
                  ACCELERATOR=SHIFT-F2, VALIDATE=NO )
  ACTION print ( COMMENT="Print order information",
                 ACCELERATOR=CONTROL-P,
                 ACCELERATOR2=F5 )
  ACTION zoom1 ( COMMENT="Open items list", VALIDATE=NO )
  ACTION zoom2 ( COMMENT="Open customers list", VALIDATE=NO )
END
```

TOPMENU section

The `TOPMENU` section defines a pull-down menu with options that are bound to actions.

Syntax

```
TOPMENU [menu-identifier] ( menu-attribute [,...] )
  group
  [...]
```

```
END
```

where *group* is:

```
GROUP group-identifier ( group-attribute [,...] )
  { command
  | group
  |  separator
  } [...]
END
```

where *command* is:

```
COMMAND command-identifier ( command-attribute [,...] )
```

and *separator* is:

```
SEPARATOR [separator-identifier] ( separator-attribute [,...] )
```

1. *menu-identifier* defines the name of the top menu (optional).
2. *group-identifier* defines the name of the group.
3. *command-identifier* defines the name of the action to bind to.
4. *separator-identifier* defines the name of the separator (optional).
5. *menu-attribute* can be: `STYLE`, `TAG`.
6. *group-attribute* is one of: `STYLE`, `TEXT`, `IMAGE`, `COMMENT`, `TAG`, `HIDDEN`.
7. *command-attribute* is one of: `STYLE`, `TEXT`, `IMAGE`, `COMMENT`, `TAG`, `HIDDEN`, `ACCELERATOR`.
8. *separator-attribute* is one of: `STYLE`, `TAG`, `HIDDEN`.

Attributes

`ACCELERATOR`, `COMMENT`, `HIDDEN`, `IMAGE`, `STYLE`, `TEXT`, `TAG`.

Usage

The `TOPMENU` section is used to define a pull-down menu in a form.

The `TOPMENU` section must appear in the sequence described in form file structure.

The `TOPMENU` section is optional.

In a `TOPMENU` section, you build a tree of `GROUP` elements to design the pull-down menu. A `GROUP` can contain `COMMAND`, `SEPARATOR` or `GROUP` children. A `COMMAND` defines a pull-down menu option that triggers an action when it is selected. In the topmenu specification, *command-identifier* defines which action a menu option is bound to. For example, if you define a topmenu option as "`COMMAND zoom`", it can be controlled by an "`ON ACTION zoom`" clause in an interactive instruction.

The topmenu commands are enabled according to the actions defined by the current interactive instruction. For example, you can define a topmenu option with the action name "cancel" to bind the pull-down item to this predefined dialog action.

An accelerator name can be defined for a topmenu command; this accelerator name will be used for display in the command item. You must define he same accelerator in the action defaults section for the action name of the topmenu command.

`TOPMENU` elements can get a `STYLE` attribute in order to use a specific rendering/decoration following presentation style definitions.

Example

```
TOPMENU tm ( STYLE="mystyle" )
```

```
  GROUP form (TEXT="Form")
    COMMAND help (TEXT="Help", IMAGE="quest")
    COMMAND quit (TEXT="Quit")
  END
  GROUP edit (TEXT="Edit")
    COMMAND accept (TEXT="Validate", IMAGE="ok",
 TAG="acceptMenu")
    COMMAND cancel (TEXT="Cancel", IMAGE="cancel")
    SEPARATOR
    COMMAND editcut   -- Gets its decoration from action
 defaults
    COMMAND editcopy  -- Gets its decoration from action
 defaults
    COMMAND editpaste -- Gets its decoration from action
 defaults
  END
  GROUP records (TEXT="Records")
    COMMAND append (TEXT="Add", IMAGE="plus")
    COMMAND delete (TEXT="Remove", IMAGE="minus")
    COMMAND update (TEXT="Modify", IMAGE="accept")
    SEPARATOR (TAG="lastSeparator")
    COMMAND search (TEXT="Search", IMAGE="find")
  END
END
```

TOOLBAR section

The TOOLBAR section defines a toolbar with buttons that are bound to actions.

Syntax

```
TOOLBAR [toolbar-identifier] [ ( toolbar-attribute [,...] ) ]
  { ITEM item-identifier  [ ( item-attribute [,...] ) ]
  | SEPARATOR [separator-identifier] [ (separator-attribute [,...] ) ]
  } [...]
END
```

1. *toolbar-identifier* defines the name of the toolbar (optional).
2. *item-identifier* defines the name of the action to bind to.
3. *separator-identifier* defines the name of the separator (optional).
4. *toolbar-attribute* is one of: STYLE, TAG, BUTTONTEXTHIDDEN.
5. *item-attribute* is one of: STYLE, TAG, TEXT, IMAGE, COMMENT, HIDDEN.
6. *separator-attribute* is one of: STYLE, TAG, HIDDEN.

Attributes

BUTTONTEXTHIDDEN, COMMENT, HIDDEN, IMAGE, STYLE, TEXT, TAG.

Usage

The TOOLBAR section defines a toolbar in a form.

The TOOLBAR section must appear in the sequence described in form file structure.

The TOOLBAR section is optional.

A TOOLBAR section defines a set of ITEM elements that can be grouped by using a SEPARATOR element. Each ITEM defines a toolbar button associated to an action by name. The SEPARATOR keyword specifies a vertical line.

The toolbar buttons are enabled according to the actions defined by the current interactive instruction. For example, you can define a toolbar button with the action name "cancel" to bind the toolbar item to this predefined dialog action.

Toolbar button labels are visible by default. The TOOLBAR supports the BUTTONTEXTHIDDEN attribute to hide the labels of buttons.

TOOLBAR elements can get a STYLE attribute in order to use a specific rendering/decoration following presentation style definitions.

Example

```
TOOLBAR tb ( STYLE="mystyle" )
  ITEM accept ( TEXT="Ok", IMAGE="ok" )
  ITEM cancel ( TEXT="Cancel", IMAGE="cancel" )
  SEPARATOR
  ITEM editcut   -- Gets its decoration from action defaults
  ITEM editcopy  -- Gets its decoration from action defaults
  ITEM editpaste -- Gets its decoration from action defaults
  SEPARATOR ( TAG="lastSeparator")
  ITEM append ( TEXT="Append", IMAGE="add" )
  ITEM update ( TEXT="Update", IMAGE="modify" )
  ITEM delete ( TEXT="Delete", IMAGE="del" )
  ITEM search ( TEXT="Search", IMAGE="find" )
END
```

SCREEN section

The SCREEN section defines the form layout for TUI mode forms.

Syntax

```
SCREEN [ SIZE lines [ BY chars ] ] [ TITLE "title" ]
{
  { text | [ item-tag [ | item-tag ] [...] ] }
  [...]
}
[END]
```

1. *lines* is the number of characters the form can display vertically. The default is 24.
2. *chars* is the number of characters the form can display horizontally. The default is the maximum number of characters in any line of the screen definition.
3. *title* is the title for the top window.
4. *item-tag* and *text* define form elements in the layout.

Usage

The SCREEN section must be used to design TUI mode screens. For a GUI mode application, use a LAYOUT or STACKED LAYOUT section instead.

The SCREEN section must appear in the sequence described in form file structure.

This section is mandatory, unless you use a LAYOUT section.

The END keyword is optional.

Inside the SCREEN section, you can define the position of text labels and form fields in the area delimited by the {} curly braces.

Horizontal lines can be specified with a set of dash characters.

Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters will be replaced by 8 blanks by fglform.

Example

```
SCREEN
{
  CustId : [f001   ] Name: [f002               ]
  Address: [f003                               ]
           [f003                               ]
  ---------------------------------------------
}
END
```

LAYOUT section

The LAYOUT section defines the graphical alignment of the form by using a tree of layout containers.

Syntax

```
LAYOUT [ ( layout-attribute [,...] ) ]
  root-container
     child-container
     [...]
  END
[END]
```

1. *layout-attribute* is an attribute for the whole form.
2. *root-container* is the first container that holds *child-containers*.

Attributes

IMAGE, MINHEIGHT, MINWIDTH, SPACING, STYLE, TEXT, TAG, VERSION, WINDOWSTYLE.

Can hold

FORM, VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, STACK, TABLE, TREE.

Usage

The LAYOUT section is used to define a a tree of layout containers, it can mix grid-based layout containers (GRID), with stack-based layout containers (STACK).

The LAYOUT section must appear in the sequence described in form file structure.

This section is mandatory, unless you use a SCREEN section.

Indentation is supported in the LAYOUT section.

The END keyword is optional.

The layout tree of the form is defined by associating layout containers. Different kinds of layout containers are provided, each of them having a specific role. Some containers such as VBOX, HBOX and FOLDER can hold children containers, while others such as GRID and TABLE define a screen area. Containers using a screen area define a formatted region containing static text labels, item tags and layout tags. External form files can be included in the current layout with the FORM clause.

```
LAYOUT (VERSION="12", STYLE="regular")
  VBOX
    GRID grid1
      grid-area
    END
```

```
    GROUP group1
      HBOX
        GRID grid2
          grid-area
        END
        TABLE table1
          table-area
        END
      END
    END
  END
END
```

The definition would result in a layout tree that looks like this:

```
-- VBOX
   |
   +-- GRID grid1
   |
   +-- GROUP group1
       |
       +-- HBOX
           |
           +-- GRID grid2
           |
           +-- TABLE table1
```

The layout section can also contain a simple `GRID` container (equivalent to a V3 SCREEN definition):

```
LAYOUT
  GRID
    grid-area
  END
END
```

Description of LAYOUT attributes

The `VERSION` attribute can be used to specify a version for the form. This allows you to indicate that the form content has changed. Typically used to avoid having the front-end reload the saved window settings.

The `MINHEIGHT`, `MINWIDTH` attributes can be used to specify a minimum width and height for the form. You typically use these attributes to force the form to get a bigger size as the default when it is first rendered. If the front-end stores window sizes, these attributes will only be significant the first time the form is opened, or each time the `VERSION` attribute is changed.

The `IMAGE` attribute can be used to define the icon of the window that will display the form. This attribute will automatically be applied to the parent window node when a form is loaded.

The `TEXT` attribute can be used to define the title of the window that will display the form. This attribute will automatically be applied to the parent window node when a form is loaded.

The `SPACING` attribute can be used to give a hint to the front-end to define the gad between form elements.

The `STYLE` attribute defines the presentation style for form elements, you can for example define a font property for all form elements.

With the `WINDOWSTYLE` attribute, you can define the window type and decoration. This attribute will automatically be applied to the parent window when a form is loaded. For backward compatibility, the `STYLE` attribute is used as the default `WINDOWSTYLE` if this attribute is not used.

FORM clause
Reuse the definition of a form in the current form.

Syntax

```
FORM "form-file"
```

1. *form-file* is the form to be included (without .per extension).

Attributes

None.

Usage

The FORM clause includes an external form at the current layout position, enforcing form re-usability, or to solve form complexity when using a DIALOG instruction. For example to define a common form header for several application forms.

Wherever a layout container can be specified, the layout of an external form can be merged into the layout of the current form, with the FORM clause.

The .per source of the included form must be readable. If the compiled version (.42f) does not exist, or is older as the .per source, fglform will automatically compile the included form.

The form compiler searches for the external form relative to the path of the current compiled form.
For example, with `fglform dir1/dir2/main.per`, when the main form includes an external form with `FORM "../otherdir/subform"`, fglform will include the form file located in `dir1/otherdir/subform.per`.

The form compiler performs an up to date test of the compiled form. Error -6842 will the thrown if the up to date test fails.

If the external form contains a TOOLBAR or a TOPMENU section, error -6841 will be thrown.

The external form must not define a SCREEN RECORD or use a TABLE already been defined in the current form, otherwise error -2024 will be thrown. Consider using the table alias syntax to avoid duplicate table names in merged forms.

The external form can define its own ACTION DEFAULTS section. The action defaults of the external file will be merged into the action defaults of the current form.

The TABINDEX attributes of the elements of the result form will be adjusted. As the result tabbing (OPTIONS FIELD ORDER FORM in programs) keeps the visual order of the layout.

Example

```
LAYOUT
  FOLDER
  PAGE page1 (TEXT = "Customer")
    FORM "customer"
  END
  PAGE page2 (TEXT = "Orders")
    FORM "orders"
  END
  END
END
```

HBOX container
Packs child layout elements horizontally.

Syntax

```
HBOX [identifier]  [ ( attribute [,...] ) ]
  layout-container
  [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *layout-container* is another child container.

Attributes

COMMENT, FONTPITCH, HIDDEN, STYLE, SPLITTER, TAG.

Can hold

VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, STACK, TABLE, TREE.

Usage

The HBOX container automatically packs the contained elements horizontally from left to right.

Contained elements are packed in the order in which they appear in the LAYOUT section of the form file.

No decoration is added when you use a HBOX container.

By combining VBOX and HBOX containers, you can define any alignment you choose.

Example

```
HBOX
  GROUP ( TEXT = "Customer" )
  {
   ...
  }
  END
  TABLE
  {
   ...
  }
  END
END
```

VBOX container
Packs child layout elements vertically.

Syntax

```
VBOX [identifier]  [ ( attribute [,...] ) ]
  layout-container
  [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.

3. *layout-container* is another child container.

Attributes

COMMENT, FONTPITCH, HIDDEN, STYLE, SPLITTER, TAG.

Can hold

VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, STACK, TABLE, TREE.

Usage

The VBOX container automatically packs the contained elements vertically from top to bottom.

Contained elements are packed in the order in which they appear in the LAYOUT section of the form file.

No decoration is added when you use a VBOX container.

By combining VBOX and HBOX containers, you can define any alignment you choose.

Example

```
VBOX
  GROUP ( TEXT = "Customer" )
  {
   ...
  }
  END
  TABLE
  {
   ...
  }
  END
END
```

GROUP container
Defines a layout area to group other layout elements together, in a grid-based layout.

Syntax

```
GROUP [identifier] [ ( attribute [,...] ) ]
  layout-container
  [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *layout-container* is another child container.

Attributes

COMMENT, FONTPITCH, STYLE, TAG, HIDDEN, TEXT.

Can hold

VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE, TREE.

Usage

In a LAYOUT tree definition, use a GROUP container to hold other containers such as a VBOX with children, or a GRID container.

For more details about this item type, see GROUP item type on page 887.

Example

```
GROUP ( TEXT = "Customer" )
  VBOX
    GRID
    {
     ...
    }
    END
    TABLE
    {
     ...
    }
    END
  END
END
```

FOLDER container
Defines the parent container for folder pages, in a grid-based layout.

Syntax

```
FOLDER [identifier] [ ( attribute [,...] ) ]
   folder-page
   [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *folder-page* defines a folder page that contains other form elements.

Attributes

COMMENT , FONTPITCH , STYLE , TAG , HIDDEN.

Can hold

PAGE

Usage

A FOLDER container including PAGE elements defines a folder tab widget.

Define each folder page with a PAGE container inside the FOLDER container.

For more details about this item type, see FOLDER item type on page 886.

PAGE container
Defines the content of a folder page, in a grid-based layout.

Syntax

```
PAGE [identifier] [ ( attribute [,...] ) ]
```

```
   layout-container
   [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *layout-container* is another child container.

Attributes

ACTION, COMMENT, HIDDEN, IMAGE, STYLE, TAG, TEXT.

Can hold

VBOX, HBOX, GROUP, FOLDER, GRID, SCROLLGRID, TABLE, TREE.

Usage

In a LAYOUT tree definition, use a PAGE container to define a folder page that holds other containers such as a VBOX with children, or a GRID container.

A PAGE container always belongs to a parent FOLDER container.

For more details about this item type, see PAGE item type on page 891.

Example

```
FOLDER
  PAGE p1 ( TEXT="Global info" )
    GRID
    {
     ...
    }
    END
  END
  PAGE p2 ( IMAGE="list" )
    TABLE
    {
     ...
    }
    END
  END
END
```

GRID container
Defines a layout area based on a grid of cells.

Syntax

```
GRID [identifier] [ ( attribute [,...] ) ]
{
  { text
  | item-tag
  | hbox-tag
  | layout-tag
  | horizontal-line }
  [...]
}
END
```

1. *text* is literal text that will appear in the form as a static label.
2. *item-tag* defines the position and length of a form item.
3. *hbox-tag* defines the position and length of several form items inside an horizontal box.
4. *layout-tag* defines the position and length of a layout tag.
5. *horizontal-line* is a set of dash characters defining a horizontal line.

Attributes

COMMENT, FONTPITCH, HIDDEN, STYLE, TAG.

Usage

The GRID container declares a formatted text block, defining the dimensions and the positions of children form items.

> **Note:** Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters will be replaced by 8 blanks by fglform.

For more details about this item type, see GRID item type on page 887.

Example

```
GRID
{
<GROUP g1                                      >
 Id:   [f1] Name: [f2                         ]
 Addr: [f3                                    ]
<                                              >
}
END
```

STACK container

The STACK container holds stack items defining a logical alignment of form items.

> **Important:** This feature is experimental, the syntax/name and semantics/behavior may change in a future version.
>
> **Important:** STACK layout was introduced for mobile application programming (GMA/GMI). This type of layout is not supported by GWC-JS and GDC.

Syntax

```
STACK
  { scalable-item
  | container-list
  }
END
```

where *container-list* is:

```
grouping-item
   leaf-item
   [...]
END
[...]
```

1. *scalable-item* is a leaf element of the stacked layout, for widgets with a scalable width and height.
2. *grouping-item* is a stacked layout grouping element that holds a list of *leaf-items*.
3. *leaf-item* is a leaf element of the stacked layout, for widgets with a fixed size (non-scalable).

Can hold

Scalable stack items: IMAGE, TEXTEDIT, WEBCOMPONENT.

or:

Grouping stack items: FOLDER, GROUP, TABLE.

Usage

The STACK container is used to define a stack-based layout.

> **Note:** Unlike grid-based containers (GRID) where element definition is splitted in the LAYOUT and ATTRIBUTES sections, the items in a STACK container define both the position and attributes.

The STACK container typically defines a list of elements (such as GROUP, FOLDER, TABLE), grouping leaf stack items (such as form fields) together:

```
LAYOUT
 STACK
  GROUP custinfo (TEXT="Customer info")
    EDIT customer.cust_num, TITLE="Num:", NOENTRY;
    EDIT customer.cust_name, TITLE="Name:", SCROLL;
    ...
  END
  TABLE cust_orders (STYLE="compact_list", DOUBLECLICK=select)
    LABEL orders.ord_num, TITLE="Num";
    LABEL orders.ord_ship, TITLE="Ship date";
    LABEL orders.ord_value, TITLE="Value";
    ...
  END
  ...
 END
END
```

A stack container can also define a single scalable stack item, such as an IMAGE, TEXTEDIT or WEBCOMPONENT:

```
STACK
  IMAGE FORMONLY.picture;
END
```

BUTTON stack item

Defines a push-button that can trigger an action, in a stack-based layout.

Syntax

```
BUTTON item-name [ , attribute-list ] ;
```

1. *item-name* defines the form item name and the action name.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COMMENT, DISCLOSUREINDICATOR, FONTPITCH, HIDDEN, IMAGE, SAMPLE, SIZEPOLICY, STYLE, TABINDEX, TAG, TEXT.

Usage

Define the rendering and behavior of a button stack item, with a BUTTON element inside a STACK container.

For more details about this item type, see BUTTON item type on page 878.

Example

```
BUTTON print, TEXT="Print Report", IMAGE="printer";
```

BUTTONEDIT stack item

Defines a line-edit with a push-button that can trigger an action, in a stack-based layout.

Syntax

```
BUTTONEDIT field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

ACTION, AUTONEXT, CENTURY, COLOR, COMPLETER, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, HIDDEN, FORMAT, IMAGE, INCLUDE, INVISIBLE, JUSTIFY, KEY, KEYBOARDHINT, NOT NULL, NOTEDITABLE, NOENTRY, PICTURE, PROGRAM, REVERSE, SAMPLE, SCROLL, STYLE, REQUIRED, TAG, TITLE, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.

Usage

Define the rendering and behavior of a buttonedit stack item, with a BUTTONEDIT element inside a STACK container.

For more details about this item type, see BUTTONEDIT item type on page 879.

Example

```
BUTTONEDIT customer.state,
    REQUIRED, IMAGE="smiley", ACTION=zoom;
```

CHECKBOX stack item

Defines a boolean or three-state checkbox field, in a stack-based layout.

Syntax

```
CHECKBOX field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TEXT, TITLE, VALIDATE LIKE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALUECHECKED, VALUEUNCHECKED.

Usage

Define the rendering and behavior of a checkbox stack item, with a CHECKBOX element inside a STACK container.

For more details about this item type, see CHECKBOX item type on page 880.

Example

```
CHECKBOX customer.active,
   REQUIRED, TEXT="Active",
   VALUECHECKED="Y", VALUEUNCHECKED="N";
```

COMBOBOX stack item

Defines a line-edit with a drop-down list of values, in a stack-based layout.

Syntax

```
COMBOBOX field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, KEY, INCLUDE, INITIALIZER, ITEMS, JUSTIFY, NOT NULL, NOENTRY, QUERYEDITABLE, REQUIRED, SAMPLE, SCROLL, SIZEPOLICY, STYLE, UPSHIFT, TAG, TABINDEX, UNSORTABLE , UNSIZABLE , UNHIDABLE , UNMOVABLE, TITLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a combobox stack item, with a COMBOBOX element inside a STACK container.

For more details about this item type, see COMBOBOX item type on page 881.

Example

```
COMBOBOX customer.city,
   ITEMS=((1,"Paris"),
          (2,"Madrid"),
          (3,"London"));
COMBOBOX customer.sector,
   REQUIRED,
   ITEMS=("SA","SB","SC");
COMBOBOX customer.state,
   NOT NULL,
   INITIALIZER=myinit;
```

DATEEDIT stack item

Defines a line-edit with a calendar widget to pick a date, in a stack-based layout.

Syntax

```
DATEEDIT field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, FORMAT, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a date edit stack item, with a DATEEDIT element inside a STACK container.

For more details about this item type, see DATEEDIT item type on page 883.

Example

```
DATEEDIT order.shipdate;
```

DATETIMEEDIT stack item

Defines a a line-edit with a calendar widget to pick a datetime, in a stack-based layout.

Syntax

```
DATETIMEEDIT field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, IMAGECOLUMN, JUSTIFY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a date-time edit stack item, with a DATETIMEEDIT element inside a STACK container.

For more details about this item type, see DATETIMEEDIT item type on page 884.

Example

```
DATETIMEEDIT package.modts;
```

EDIT stack item

Defines an element to enter a single-line text, in a stack-based layout.

Syntax

```
EDIT [identifier] [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, PROGRAM, REQUIRED, SAMPLE, SCROLLBARS, STYLE, STRETCH, TAG, TABINDEX, TITLE, UPSHIFT, VALIDATE LIKE, WANTTABS, WANTNORETURNS.

Usage

Define the rendering and behavior of an edit stack item, with an EDIT element inside a STACK container.

For more details about this item type, see EDIT item type on page 886.

Example

```
EDIT customer.cust_name, NOT NULL;
```

FOLDER stack item

Defines a stack area to hold a set of folder pages, in a stack-based layout.

Syntax

```
FOLDER [identifier] [ ( attribute-list  ) ]
  folder-page
  [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute-list* defines the aspect and behavior of the form item.
3. *folder-page* is a page element in the folder definition.

Attributes

COMMENT , FONTPITCH , STYLE , TAG , HIDDEN.

Can hold

PAGE.

Usage

Use a FOLDER stack layout element to define a set of folder pages with a folder tab widget.

Define each folder page with a PAGE stack item inside the FOLDER container.

For more details about this item type, see FOLDER item type on page 886.

Example

```
FOLDER folder1 ( STYLE="common" )
   PAGE page1 ( TEXT="Order details" )
      ...
   END
   PAGE page2 ( TEXT="Order items" )
      ...
   END
   ...
END
```

GROUP stack item

Defines a stack area to group other layout elements together, in a stack-based layout.

Syntax

```
GROUP [identifier] [ ( attribute-list ) ]
  stack-item
  [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute-list* defines the aspect and behavior of the form item.
3. *stack-item* is child element in the stack container.

Attributes

COMMENT, FONTPITCH, HIDDEN, STYLE, TAG, TEXT.

Can hold

BUTTON, BUTTONEDIT, CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, EDIT, IMAGE, LABEL, PROGRESSBAR, PHANTOM, SLIDER, SPINEDIT,TEXTEDIT, TIMEEDIT, RADIOGROUP, WEBCOMPONENT.

Usage

Use a GROUP stack layout element to group other stack items together.

For more details about this item type, see GROUP item type on page 887.

Example

```
GROUP group1 (TEXT="Customer info")
   EDIT ...
   BUTTONEDIT ...
   ...
END
```

IMAGE stack item

Defines an element to display an image resource, in a stack-based layout.

Syntax 1: Defining a *form field image*

```
IMAGE field-name [ , attribute-list ] ;
```

Syntax 2: Defining a *static image*

```
IMAGE : item-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *item-name* identifies the form item for a static image.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

ACTION, AUTOSCALE, COMMENT, HEIGHT, HIDDEN, STYLE, STRETCH, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, WIDTH.

Static image only: IMAGE.

Image field only: JUSTIFY, SIZEPOLICY, SAMPLE.

Usage

Define the rendering and behavior of an image stack item, with a IMAGE element inside a STACK container.

> **Note:** The IMAGE stack item can be used inside a stack container like a group, or as root element of the STACK container: When used directly under the STACK container, the IMAGE stack item must be the only element in the container. It will be rendered a scalable form item that can stretch to fit the front-end screen size.

For more details about this item type, see IMAGE item type on page 888.

Example

```
IMAGE cars.picture, COMMENT="Picture of the car";
```

LABEL stack item

Defines a simple text area to display a read-only value, in a stack-based layout.

Syntax 1: Defining a *form field label*

```
LABEL field-name [ , attribute-list ] ;
```

Syntax 2: Defining a *static label*

```
LABEL : item-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *item-name* identifies the form element (name attribute in .42f) of a static label.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, IMAGECOLUMN, JUSTIFY, REVERSE, SIZEPOLICY, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Form field label only: FORMAT, SAMPLE.

Static label only: TEXT.

Usage

Define the rendering and behavior of an label stack item, with a LABEL element inside a STACK container.

For more details about this item type, see LABEL item type on page 890.

Example

```
LABEL vehicle.description, STYLE="normal";
```

PAGE stack item

Defines the content of a folder page stack item.

Syntax

```
PAGE [identifier] [ ( attribute-list ) ]
  { scalable-item
```

```
  [ grouping-item
          leaf-item
          [...]
      END
  ]
END
```

1. *identifier* defines the name of the element.
2. *attribute-list* defines the aspect and behavior of the form item.
3. *scalable-item* is a stacked layout items that can grow and shrink.
4. *grouping-item* is a stacked layout grouping element that holds a list of *stack-items*.
5. *leaf-item* is a leaf element of the stacked layout, for widgets with a fixed size (non-scalable).

Attributes

ACTION, COMMENT, HIDDEN, IMAGE, STYLE, TAG, TEXT.

Can hold

GROUP, IMAGE, TABLE, TEXTEDIT, WEBCOMPONENT.

Usage

Use a PAGE stack layout element to group other stack items together.

A PAGE stack item always belongs to a parent FOLDER stack item.

For more details about this item type, see PAGE item type on page 891.

Example

```
FOLDER folder1 ( STYLE="common" )
   PAGE page1 ( TEXT="Customer info" )
      GROUP
          EDIT ...
          EDIT ...
          EDIT ...
      END
   END
   PAGE page2 ( TEXT="Picture" )
      IMAGE FORMONLY.cust_pic;
   END
   PAGE page3 ( TEXT="Comments" )
      TEXTEDIT customer.cust_desc;
   END
   ...
END
```

PHANTOM stack item

Defines a form field in a stack-based container, that must not be displayed to the end user.

Syntax

```
PHANTOM [field-name] ;
```

1. *field-name* identifies the name of the screen record field.

Usage

Define a PHANTOM leaf element in a stack container, to declare a form field to be used by a dialog, without beeing displayed to the user.

For more details, see Phantom fields on page 861.

Example

```
PHANTOM customer.cust_name;
```

PROGRESSBAR stack item

Defines a progress indicator field, in a stack-based layout.

Syntax

```
PROGRESSBAR field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, JUSTIFY, VALUEMIN, VALUEMAX, SAMPLE, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Usage

Define the rendering and behavior of an progress bar stack item, with an PROGRESSBAR element inside a STACK container.

For more details about this item type, see PROGRESSBAR item type on page 892.

Example

```
PROGRESSBAR workstate.position,
  VALUEMIN=-100, VALUEMAX=+100;
```

RADIOGROUP stack item

Defines a mutual exclusive set of options field, in a stack-based layout.

Syntax

```
RADIOGROUP field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, ITEMS, JUSTIFY, KEY, NOT NULL, NOENTRY, ORIENTATION, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a radio group stack item, with an `EDIT` element inside a `STACK` container.

For more details about this item type, see RADIOGROUP item type on page 893.

Example

```
RADIOGROUP player.level,
  ITEMS=((1,"Beginner"),
         (2,"Normal"),
         (3,"Expert"));
```

SLIDER stack item

Defines a slider element, in a stack-based layout.

Syntax

```
SLIDER field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

`COLOR`, `COLOR WHERE`, `COMMENT`, `DEFAULT`, `FONTPITCH`, `HIDDEN`, `INCLUDE`, `JUSTIFY`, `ORIENTATION`, `SAMPLE`, `STEP`, `STYLE`, `TABINDEX`, `TAG`, `TITLE`, `UNSORTABLE`, `UNSIZABLE`, `UNHIDABLE`, `UNMOVABLE`, `VALIDATE LIKE`, `VALUEMIN`, `VALUEMAX`.

Usage

Define the rendering and behavior of a slider stack item, with an `SLIDER` element inside a `STACK` container.

For more details about this item type, see SLIDER item type on page 895.

Example

```
SLIDER workstate.duration,
  VALUEMIN=0, VALUEMAX=5,
  STEP=1;
```

SPINEDIT stack item

Defines a spin box widget to enter integer values, in a stack-based layout.

Syntax

```
SPINEDIT field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

`AUTONEXT`, `COLOR`, `COLOR WHERE`, `COMMENT`, `DEFAULT`, `FONTPITCH`, `HIDDEN`, `IMAGECOLUMN`, `INCLUDE`, `JUSTIFY`, `NOT NULL`, `NOENTRY`, `REQUIRED`, `SAMPLE`, `STEP`, `STYLE`, `TABINDEX`, `TAG`, `TITLE`, `UNSORTABLE`, `UNSIZABLE`, `UNHIDABLE`, `UNMOVABLE`, `VALIDATE LIKE`, `VALUEMIN`, `VALUEMAX`.

Usage

Define the rendering and behavior of a spin edit stack item, with an SPINEDIT element inside a STACK container.

For more details about this item type, see SPINEDIT item type on page 895.

Example

```
SPINEDIT command.nbitems, STEP=5;
```

TABLE stack item

Defines a re-sizable table designed to display a list of records, in a stack-based layout.

Syntax

```
TABLE identifier [ ( attribute-list ) ]
  stack-item
  [...]
END
```

1. *identifier* defines the name of the element.
2. *attribute-list* defines the aspect and behavior of the form item.
3. *stack-item* is child element in the stack container defining a column in the table.

Attributes

AGGREGATETEXT, COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT.

Can hold

BUTTONEDIT, CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, EDIT, IMAGE, LABEL, PROGRESSBAR, PHANTOM, SLIDER, SPINEDIT, TIMEEDIT, RADIOGROUP.

Usage

The TABLE stack layout element defines defines a list view element, in a stack-based layout.

To create a table view in a stacked layout, define the following elements in the form file:

1. The layout of the list, with a TABLE stack item.
2. The columns definitions as stack items inside the TABLE item.

Note: The TABLE item must get an identifier, that will be used as screen-array in list dialogs.

For more details about table view programming, see Table views on page 1345

Example

```
TABLE custlist (STYLE="regular")
   EDIT ...
   BUTTONEDIT ...
   ...
END
```

TEXTEDIT stack item

Defines an multi-line edit field, in a stack-based layout.

Syntax

```
TEXTEDIT [identifier] [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, PROGRAM, REQUIRED, SAMPLE, SCROLLBARS, STYLE, STRETCH, TAG, TABINDEX, TITLE, UPSHIFT, VALIDATE LIKE, WANTTABS, WANTNORETURNS.

Usage

Define the rendering and behavior of a text edit stack item, with a TEXTEDIT element inside a STACK container.

> **Note:** The TEXTEDIT stack item can be used inside a stack container like a group, or as root element of the STACK container: When used directly under the STACK container, the TEXTEDIT stack item must be the only element in the container. It will be rendered a scalable form item that can stretch to fit the front-end screen size.

For more details about this item type, see TEXTEDIT item type on page 897.

Example

```
TEXTEDIT customer.cust_address, HEIGHT=3, REQUIRED;
```

TIMEEDIT stack item

Defines a line-edit with a clock widget to pick a time, in a stack-based layout.

Syntax

```
TIMEEDIT field-name [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a time edit stack item, with a TIMEEDIT element inside a STACK container.

For more details about this item type, see TIMEEDIT item type on page 898.

Example

```
TIMEEDIT package.arrtime;
```

WEBCOMPONENT stack item

Defines a generic form field that can receive an external widget, in a stack-based layout.

Syntax

```
WEBCOMPONENT [identifier] [ , attribute-list ] ;
```

1. *field-name* identifies the name of the screen record field.
2. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMPONENTTYPE, COMMENT, DEFAULT, FONTPITCH, HEIGHT, HIDDEN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, PROPERTIES, REQUIRED, SCROLLBARS, SIZEPOLICY, STYLE, STRETCH, TAG, TABINDEX, TITLE, VALIDATE LIKE, WIDTH.

Usage

Define the rendering and behavior of a web component stack item, with a WEBCOMPONENT element inside a STACK container.

> **Note:** The WEBCOMPONENT stack item can be used inside a stack container like a group, or as root element of the STACK container: When used directly under the STACK container, the WEBCOMPONENT stack item must be the only element in the container. It will be rendered a scalable form item that can stretch to fit the front-end screen size.

For more details about this item type, see WEBCOMPONENT item type on page 900.

Example

```
-- URL-based web component (recommended):
WEBCOMPONENT FORMONLY.mymap;

-- gICAPI web component:
WEBCOMPONENT FORMONLY.mycal,
     COMPONENTTYPE="Calendar", -- lookup "Calendar.html"
     STYLE="regular";
```

SCROLLGRID container

Defines a scrollable grid view widget, in a grid-based layout.

Syntax

```
SCROLLGRID [identifier] [ ( attribute [,...] ) ]
{
  row-template
  [...]
}
END
```

where *row-template* is a text block containing:

```
 item-tag
```

```
[...]
```

1. *item-tag* defines the position and length of a form item. This item tag must define a form field.

Attributes

COMMENT, FONTPITCH, STYLE, TAG, HIDDEN, WANTFIXEDPAGESIZE.

Usage

The SCROLLGRID container declares a formatted text block defining the dimensions and the position of the logical elements of a screen for a multi-record presentation.

> **Important:** This feature is not supported on mobile platforms.

Scrollgrids are by default non-resizable; The number of visible rows is defined by the number of repeated form items inside the SCROLLGRID area. To implement a resizable scrollgrid, define a single scrollgrid row in the form layout, and use the WANTFIXEDPAGESIZE=NO attribute. Resizable scrollgrids is the recommended way to implement scrollgrids.

> **Note:** Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters will be replaced by 8 blanks by fglform.

For more details about this item type, see SCROLLGRID item type on page 894.

Example 1: Resizable scrollgrid (using WANTFIXEDPAGESIZE=NO):

```
SCROLLGRID (WANTFIXEDPAGESIZE=NO)
{
 [f001   ]   [f002                          ]
 [f003                                      ]
}
END
```

Example 2: Scrollgrid with fixed page size, using four rows:

```
SCROLLGRID
{
 [f001   ]   [f002                          ]
 [f003                                      ]

 [f001   ]   [f002                          ]
 [f003                                      ]

 [f001   ]   [f002                          ]
 [f003                                      ]

 [f001   ]   [f002                          ]
 [f003                                      ]

}
END
```

TABLE container
Defines a re-sizable table designed to display a list of records.

Syntax

```
TABLE [identifier] [ ( attribute [,...] ) ]
{
```

```
 title [...]
[col-name   [|...]     ]
[...]
[aggr-name  [|...]     ]
}
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *title* is the text to be displayed as column title.
4. *col-name* is an identifier that references a form field.
5. *aggr-name* is an identifier that references an aggregate Field.

Attributes

AGGREGATETEXT, COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT.

Usage:

The TABLE container defines a list view element in a grid-based layout.

To create a table view in a grid layout, define the following elements in the form file:

1. The layout of the list, with a TABLE container in the LAYOUT section.
2. The column data types and field properties, in the ATTRIBUTES section.
3. The field list definition to group form fields together with a screen array, in the INSTRUCTIONS section.

For more details about this item type, see TABLE item type on page 896.

Example

```
SCHEMA videolab
LAYOUT ( TEXT="Customer list" )
TABLE ( TAG="normal" )
{
 [c1     |c2                          |c3        |c4 ]
 [c1     |c2                          |c3        |c4 ]
 [c1     |c2                          |c3        |c4 ]
 [c1     |c2                          |c3        |c4 ]
}
END
END
TABLES
  customer
END
ATTRIBUTES
  EDIT c1 = customer.cust_num, TITLE="Num";
  EDIT c2 = customer.cust_name, TITLE="Customer name";
  EDIT c3 = customer.cust_cdate, TITLE="Date";
  CHECKBOX c4 = customer.cust_status, TITLE="Status";
END
INSTRUCTIONS
  SCREEN RECORD custlist( cust_num, cust_name, cust_cdate,
 cust_status )
END
```

TREE container
The TREE container defines the presentation of a list of ordered records in a tree-view widget.

Syntax

```
TREE [identifier] [ ( attribute [,...] ) ]
{
 title [...]
 [name_column   [|identifier   [|...]] ]
[...]
}
END
```

1. *identifier* defines the name of the element.
2. *attribute* is an attribute for the element.
3. *title* is the text to be displayed as column title.
4. *name_column* is a mandatory column referencing a form item defining the node text.
5. *identifier* references a form item.

Attributes

COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT, PARENTIDCOLUMN, IDCOLUMN, EXPANDEDCOLUMN, ISNODECOLUMN, IMAGEEXPANDED, IMAGECOLLAPSED, IMAGELEAF.

Usage

To create a tree view in a grid-based layout, you must define the following elements in the form file:

1. The layout of the tree-view, with a TREE container in the LAYOUT section.
2. The column data types and field properties, in the ATTRIBUTES section.
3. The field list definition to group form fields together with a screen array, in the INSTRUCTIONS section.

For more details about this item type, see TREE item type on page 899.

Example

```
LAYOUT
GRID
{
<Tree t1                        >
 Name                    Index
[c1                      |c2    ]
[c1                      |c2    ]
[c1                      |c2    ]
[c1                      |c2    ]
}
END
END

ATTRIBUTES
LABEL c1 = FORMONLY.name;
LABEL c2 = FORMONLY.idx;
PHANTOM FORMONLY.pid;
PHANTOM FORMONLY.id;
TREE t1: tree1
    PARENTIDCOLUMN = pid,
    IDCOLUMN = id;
END
```

```
INSTRUCTIONS
SCREEN RECORD sr_tree(name, pid, id, idx);
END
```

TABLES section
Defines the list of database tables referenced by form field definitions.

Syntax

```
TABLES
[ alias = [database[@dbserver]:][owner.] ] table [,...]
[END]
```

1. *alias* represents an alias name for the given table.
2. *table* is the name of the database table.
3. *database* is the name of the database of the table (see warnings).
4. *dbserver* identifies the Informix® database server (INFORMIXSERVER)
5. *owner* is the name of the table owner (see warnings).

Usage

The TABLES section lists every database table or view referenced the form specification file. This section is mandatory when form fields reference database columns defined in the database schema file.

The TABLE section must appear in the sequence described in form file structure.

The END keyword is optional.

The SCHEMA section must also exist to define the database schema.

Field identifiers in programs or in other sections of the form specification file can reference screen fields as *column*, *alias.column*, or *table.column*.

The same *alias* must also appear in screen interaction statements of programs that reference screen fields linked to the columns of a table that has an *alias*.

If a table requires the name of an *owner* or of a *database* as a qualifier, the TABLES section must also declare an alias for the table. The *alias* can be the same identifier as *table*.

For backward compatibility with the Informix® form specification, the comma separator is optional and the *database*, *dbserver* and *owner* specifications are ignored.

Example

```
SCHEMA stores
LAYOUT
GRID
{
  ...
}
END
TABLES
 customer, orders
END
ATTRIBUTES
...
END
```

ATTRIBUTES section

The ATTRIBUTES section describes properties of grid-based layout elements used in the form.

Syntax

```
ATTRIBUTES
 { form-field-definition
 | phantom-field-definition
 | form-item-definition }
[...]
[END]
```

where *form-field-definition* is:

```
item-type item-tag = field-name [ , attribute-list ] ;
```

where *phantom-field-definition* is:

```
PHANTOM field-name ;
```

where *form-item-definition* is:

```
item-type item-tag: item-name [ , attribute-list ] ;
```

1. *item-type* defines the type of the Form Item.
2. *item-tag* is the name of the screen element used in the LAYOUT section.
3. *field-name* defines the name of the screen record field.
4. *item-name* identifies the form item that is not a form field containing data.
5. *attribute-list* defines the aspect and behavior of the form item.

where *attribute-list* is:

```
attribute [,...]
```

1. The attribute list is a comma-separated list of attributes.

where *attribute* is:

```
attribute-name [ = { value | value-list } ]
```

1. *attribute* identifies the attribute of the form item.

where *value-list* is:

```
( { value | sub-value-list } [,...])
```

1. *value* is a string, date or numeric literal, or predefined constant like TODAY.
2. *sub-value-list* is a set of values separated by comma, to support subset definitions as in "(1,(21,22),(31,32,33))".

Usage

The ATTRIBUTES section is required to define the attributes for the form items used in grid-based containers of the LAYOUT section.

The ATTRIBUTES section must appear in the sequence described in form file structure.

The END keyword is optional.

Every item-tag used in the LAYOUT section must get an item definition in the ATTRIBUTES section.

A form item definition is associated by name to an item tag or layout tag defined in the grid-based container.

In order to define a form field, the form item definition must use the equal sign notation to associate a screen record field with the form item. If the form item is not associated with a screen record field (for example, a push button), you must use the colon notation.

To match the complete structure of a database table record, additional fields can be defined as phantom fields, when no corresponding item tag is used in the layout.

Form item definitions can optionally include an *attribute-list* to specify the appearance and behavior of the item. For example, you can define acceptable input values, on-screen comments, and default values for fields.

When no screen record is defined in the `INSTRUCTION` section, a default screen record is built for each set of form items declared with the same table name.

The order in which you list the form items determines the order of fields in the default screen records that the form compiler creates for each table.

To define form items as form fields, you are not required to specify *table* unless the name *column* is not unique within the form specification. However, it is recommended that you always specify *table.column* rather than the unqualified *column* name. As you can refer to field names collectively through a screen record built upon all the fields linked to the same table, your forms might be easier to work with if you specify *table* for each field.

When used in a table, some widgets are rendered only when the user enters in the field. For example `RadioGroup`, `CheckBox`, `ComboBox`, `ProgressBar`.

Example

```
SCHEMA game
LAYOUT
GRID
{
  ...
}
END
TABLES
 player
END
ATTRIBUTES
 EDIT f001 = player.name, REQUIRED,
               COMMENT="Enter player's name";
 EDIT f002 = player.ident, NOENTRY;
 COMBOBOX f003 = player.level, NOT NULL,
      ITEMS=((1,"Beginner"), (2,"Normal"),(3,"Expert"));
 CHECKBOX f004 = FORMONLY.winner,
      VALUECHECKED=1, VALUEUNCHECKED=0,
      TEXT="Winner";
 BUTTON b1: print, TEXT="Print Report";
 GROUP g1: print, TEXT="Description";
END
```

AGGREGATE item definition
Defines screen-record fields that hold computed values to be displayed as footer cells in a `TABLE` container.

Syntax

```
AGGREGATE item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AGGREGATETEXT, AGGREGATETYPE.

Usage

Aggregate fields used as must be declared with an AGGREGATE element in the ATTRIBUTES section.

Important: This feature is not supported on mobile platforms.

For more details see Aggregate fields on page 863.

Example

```
AGGREGATE total = FORMONLY.o_total,
                  AGGREGATETEXT = "Total:",
                  AGGREGATETYPE = SUM;
```

PHANTOM item definition
Defines a form field in a grid-based container, that must not be displayed to the end user.

Syntax

```
PHANTOM [field-name] ;
```

1. *field-name* identifies the name of the screen record field.

Usage

Define a phantom form field (that will be used by a dialog, but not displayed in the form layout), with a PHANTOM element in the ATTRIBUTES section.

For more details, see Phantom fields on page 861.

Example

```
PHANTOM customer.cust_name;
```

BUTTON item definition
Defines a push-button that can trigger an action, in a grid-based layout.

Syntax

```
BUTTON item-tag: item-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *item-name* defines the form item name and the action name.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COMMENT, DISCLOSUREINDICATOR, FONTPITCH, HIDDEN, IMAGE, SAMPLE, SIZEPOLICY, STYLE, TABINDEX, TAG, TEXT.

Usage

Define the rendering and behavior of a button item tag, with a `BUTTON` element in the `ATTRIBUTES` section.

For more details about this item type, see BUTTON item type on page 878.

Example

```
LAYOUT
GRID
{
[btn1         ]
 ...

}
END
END

ATTRIBUTES
BUTTON btn1: print, TEXT="Print Report", IMAGE="printer";
...
```

BUTTONEDIT item definition
Defines a line-edit with a push-button that can trigger an action, in a grid-based layout.

Syntax

```
BUTTONEDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

ACTION, AUTONEXT, CENTURY, COLOR, COMPLETER, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, FONTPITCH, HIDDEN, FORMAT, IMAGE, INCLUDE, INVISIBLE, JUSTIFY, KEY, KEYBOARDHINT, NOT NULL, NOTEDITABLE, NOENTRY, PICTURE, PROGRAM, REVERSE, SAMPLE, SCROLL, STYLE, REQUIRED, TAG, TITLE, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.

Usage

Define the rendering and behavior of a buttonedit item tag, with a `BUTTONEDIT` element in the `ATTRIBUTES` section.

For more details about this item type, see BUTTONEDIT item type on page 879.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END
```

```
ATTRIBUTES
BUTTONEDIT f1 = customer.state,
   REQUIRED, IMAGE="smiley", ACTION=zoom;
...
```

CANVAS item definition
Defines an area in which you can draw shapes, in a grid-based layout.

Syntax

```
CANVAS item-tag: item-name [  , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COMMENT, HIDDEN, TAG.

Usage

Define the rendering and behavior of a canvas drawing area item tag, with a CANVAS element in the ATTRIBUTES section.

Note: The CANVAS feature is deprecated, consider using a WEBCOMPONENT with SVG graphics.

Example

```
LAYOUT
GRID
{
[cvs1             ]
[                 ]
[                 ]
 ...

}
END
END

ATTRIBUTES
CANVAS cvs1: canvas1;
...
```

CHECKBOX item definition
Defines a boolean or three-state checkbox field, in a grid-based layout.

Syntax

```
CHECKBOX item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TEXT, TITLE, VALIDATE LIKE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALUECHECKED, VALUEUNCHECKED.

Usage

Define the rendering and behavior of a checkbox item tag, with a CHECKBOX element in the ATTRIBUTES section.

For more details about this item type, see CHECKBOX item type on page 880.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
CHECKBOX f1 = customer.active,
   REQUIRED, TEXT="Active",
   VALUECHECKED="Y", VALUEUNCHECKED="N";
...
```

COMBOBOX item definition
Defines a COMBOBOX item in a grid-based layout, in a grid-based layout.

Syntax

```
COMBOBOX item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, KEY, INCLUDE, INITIALIZER, ITEMS, JUSTIFY, NOT NULL, NOENTRY, QUERYEDITABLE, REQUIRED, SAMPLE, SCROLL, SIZEPOLICY, STYLE, UPSHIFT, TAG, TABINDEX, UNSORTABLE , UNSIZABLE , UNHIDABLE , UNMOVABLE, TITLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a combobox item tag, with a COMBOBOX element in the ATTRIBUTES section.

For more details about this item type, see COMBOBOX item type on page 881.

Example

```
LAYOUT
```

```
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
COMBOBOX f1 = customer.city,
   ITEMS=((1,"Paris"),
          (2,"Madrid"),
          (3,"London"));
...
```

DATEEDIT item definition
Defines a line-edit with a calendar widget to pick a date, in a grid-based layout.

Syntax

```
DATEEDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, CENTURY, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, FORMAT, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a date edit item tag, with a DATEEDIT element in the ATTRIBUTES section.

For more details about this item type, see DATEEDIT item type on page 883.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
DATEEDIT f1 = order.shipdate;
...
```

DATETIMEEDIT item definition
Defines a line-edit with a calendar widget to pick a datetime, in a grid-based layout.

Syntax

```
DATETIMEEDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, IMAGECOLUMN, JUSTIFY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a date edit item tag, with a DATETIMEEDIT element in the ATTRIBUTES section.

For more details about this item type, see DATETIMEEDIT item type on page 884.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
DATETIMEEDIT f1 = package.modts;
...
```

EDIT item definition
Defines a simple line-edit field, in a grid-based layout.

Syntax

```
EDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, CENTURY, COLOR, COMPLETER, COLOR WHERE, COMMENT, DEFAULT, DISPLAY LIKE, DOWNSHIFT, HIDDEN, FONTPITCH, FORMAT, IMAGECOLUMN, INCLUDE, INVISIBLE, JUSTIFY, KEYBOARDHINT, KEY, NOT NULL, NOENTRY, PICTURE, PROGRAM, REQUIRED, REVERSE, SAMPLE, STYLE, SCROLL, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, VERIFY.

Usage

Define the rendering and behavior of an edit item tag, with an `EDIT` element in the `ATTRIBUTES` section.

For more details about this item type, see EDIT item type on page 886.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
EDIT f1 = customer.cust_state,
   REQUIRED,
   COMMENT = %"customer.cust_state.comment",
   INCLUDE=(0,1,2);
...
```

GROUP item definition
Defines a groupbox layout tag, in a grid-based layout.

Syntax

```
GROUP layout-tag: item-name [  , attribute-list ] ;
```

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

`COMMENT`, `FONTPITCH`, `GRIDCHILDRENINPARENT`, `HIDDEN`, `STYLE`, `TAG`, `TEXT`.

Usage

Define the rendering and behavior of a group layout tag, with an `GROUP` element in the `ATTRIBUTES` section.

For more details about this item type, see GROUP item type on page 887.

Example

```
LAYOUT
GRID
{
<GROUP g1             >
 Num: [f001         ]
 ...

}
END
END
```

```
ATTRIBUTES
GROUP g1: group1,
   TEXT="Description",
   GRIDCHILDRENINPARENT;
...
```

IMAGE item definition
Defines an area that can display an image resource, in a grid-based layout.

Syntax 1: Defining a *form field image*

```
IMAGE item-tag = field-name [ , attribute-list ] ;
```

Syntax 2: Defining a *static image*

```
IMAGE item-tag: item-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *item-name* identifies the form item for a static image.
4. *attribute-list* defines the aspect and behavior of the form item.

Attributes

ACTION, AUTOSCALE, COMMENT, HEIGHT, HIDDEN, STYLE, STRETCH, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, WIDTH.

Image field only: JUSTIFY, SIZEPOLICY, SAMPLE.

Static image only: IMAGE.

Usage

Define the rendering and behavior of an image item tag, with an IMAGE element in the ATTRIBUTES section.

For more details about this item type, see IMAGE item type on page 888.

Example

```
LAYOUT
GRID
{
[f1                    ]
[                      ]
[                      ]
[                      ]
 ...

}
END
END

ATTRIBUTES
IMAGE f1 = cars.picture,
      SIZEPOLICY=FIXED, AUTOSCALE,
      COMMENT="Picture of the car";
...
```

LABEL item definition
Defines a simple text area to display a read-only value, in a grid-based layout.

Syntax 1: Defining a *form field label*

```
LABEL item-tag = field-name [ , attribute-list ] ;
```

Syntax 2: Defining a *static label*

```
LABEL item-tag: item-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *item-name* identifies the form element (name attribute in .42f) of a static label.
4. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, IMAGECOLUMN, JUSTIFY, REVERSE, SIZEPOLICY, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Form field label only: FORMAT, SAMPLE.

Static label only: TEXT.

Usage

Define the rendering and behavior of an label item tag, with an LABEL element in the ATTRIBUTES section.

For more details about this item type, see LABEL item type on page 890.

Example

```
LAYOUT
GRID
{
[l1  :f1         ]
 ...

}
END
END

ATTRIBUTES
LABEL l1: label1, TEXT="Desc:"; -- This is a static label
LABEL f1 = vehicle.description; -- This is a form field label
...
```

PROGRESSBAR item definition
Defines a progress indicator field, in a grid-based layout.

Syntax

```
PROGRESSBAR item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.

3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, FONTPITCH, HIDDEN, JUSTIFY, VALUEMIN, VALUEMAX, SAMPLE, STYLE, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE.

Usage

Define the rendering and behavior of an progress bar item tag, with an PROGRESSBAR element in the ATTRIBUTES section.

For more details about this item type, see PROGRESSBAR item type on page 892.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
PROGRESSBAR f1 = workstate.position,
  VALUEMIN=-100, VALUEMAX=+100;
...
```

RADIOGROUP item definition
Defines a mutual exclusive set of options field, in a grid-based layout.

Syntax

```
RADIOGROUP item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, ITEMS, JUSTIFY, KEY, NOT NULL, NOENTRY, ORIENTATION, REQUIRED, SAMPLE, SIZEPOLICY, STYLE, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a radio group item tag, with a RADIOGROUP element in the ATTRIBUTES section.

For more details about this item type, see RADIOGROUP item type on page 893.

Example

```
LAYOUT
GRID
```

```
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
RADIOGROUP f1 = player.level,
  ITEMS=((1,"Beginner"),
         (2,"Normal"),
         (3,"Expert"));
...
```

SCROLLGRID item definition
Defines a scrollgrid layout tag, in a grid-based layout.

Syntax

```
SCROLLGRID layout-tag: item-name [  , attribute-list ] ;
```

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COMMENT, FONTPITCH, GRIDCHILDRENINPARENT, HIDDEN, STYLE, TAG.

Usage

The SCROLLGRID form item type to specify the attributes of a scrollgrid container defined with a layout tag.

Important: This feature is not supported on mobile platforms.

For more details about this item type, see SCROLLGRID item type on page 894.

Example

```
LAYOUT
GRID
{
<SCROLLGRID sg1             >
 [f001         ]
 ...

}
END
END

ATTRIBUTES
SCROLLGRID sg1: scrollgrid1,
  GRIDCHILDRENINPARENT;
```

SLIDER item definition
Defines a slider element, in a grid-based layout.

Syntax

```
SLIDER item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, ORIENTATION, SAMPLE, STEP, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, VALUEMIN, VALUEMAX.

Usage

Define the rendering and behavior of a slider item tag, with an SLIDER element in the ATTRIBUTES section.

For more details about this item type, see SLIDER item type on page 895.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
SLIDER f1 = workstate.duration,
  VALUEMIN=0, VALUEMAX=50,
  STEP=1;
...
```

SPINEDIT item definition
Defines a spin box widget to enter integer values, in a grid-based layout.

Syntax

```
SPINEDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STEP, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, VALUEMIN, VALUEMAX.

Usage

Define the rendering and behavior of a spin edit item tag, with an SPINEDIT element in the ATTRIBUTES section.

For more details about this item type, see SPINEDIT item type on page 895.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
SPINEDIT f1 = command.nbitems, STEP=5;
...
```

TABLE item definition
Defines attributes for a table layout tag, in a grid-based layout.

Syntax

```
TABLE layout-tag: item-name [  , attribute-list ] ;
```

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AGGREGATETEXT, COMMENT, DOUBLECLICK, FONTPITCH, HEIGHT, HIDDEN, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH.

Usage

Define a TABLE element in the ATTRIBUTES section, to configure a table layouted with a <TABLE > layout tag.

For more details about this item type, see TABLE item type on page 896.

Example

```
LAYOUT
GRID
{
<TABLE t1                    >
[c1  |c2        |c3          ]
[c1  |c2        |c3          ]
 ...

}
END
END
```

```
ATTRIBUTES
TABLE t1: table1, UNSORTABLECOLUMNS;
...
```

TEXTEDIT item definition
Defines a multi-line edit field, in a grid-based layout.

Syntax

```
TEXTEDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMMENT, DEFAULT, DOWNSHIFT, FONTPITCH, HIDDEN, INCLUDE, JUSTIFY, KEY, NOT NULL, NOENTRY, PROGRAM, REQUIRED, SAMPLE, SCROLLBARS, STYLE, STRETCH, TAG, TITLE, TABINDEX, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, UPSHIFT, VALIDATE LIKE, WANTTABS, WANTNORETURNS.

Usage

Define the rendering and behavior of a text edit item tag, with an TEXTEDIT element in the ATTRIBUTES section.

For more details about this item type, see TEXTEDIT item type on page 897.

Example

```
LAYOUT
GRID
{
[f1                    ]
[                      ]
[                      ]
[                      ]
 ...

}
END
END

ATTRIBUTES
TEXTEDIT f1 = customer.address,
   WANTTABS, SCROLLBARS=BOTH;
...
```

TIMEEDIT item definition
Defines a line-edit with a clock widget to pick a time, in a grid-based layout.

Syntax

```
TIMEEDIT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.

2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

AUTONEXT, COLOR, COLOR WHERE, COMMENT, DEFAULT, FONTPITCH, HIDDEN, IMAGECOLUMN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, REQUIRED, SAMPLE, STYLE, TABINDEX, TAG, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE.

Usage

Define the rendering and behavior of a text edit item tag, with an TEXTEDIT element in the ATTRIBUTES section.

For more details about this item type, see TIMEEDIT item type on page 898.

Example

```
LAYOUT
GRID
{
[f1         ]
 ...

}
END
END

ATTRIBUTES
TIMEEDIT f1 = package.arrtime;
...
```

TREE item definition
Defines attributes for a tree layout tag, in a grid-based layout.

Syntax

```
TREE layout-tag: item-name [  , attribute-list ] ;
```

1. *layout-tag* is an identifier that defines the name of the layout tag.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COMMENT, DOUBLECLICK, HIDDEN, FONTPITCH, STYLE, TAG, UNHIDABLECOLUMNS, UNMOVABLECOLUMNS, UNSIZABLECOLUMNS, UNSORTABLECOLUMNS, WANTFIXEDPAGESIZE, WIDTH, HEIGHT, PARENTIDCOLUMN, IDCOLUMN, EXPANDEDCOLUMN, ISNODECOLUMN, IMAGEEXPANDED, IMAGECOLLAPSED, IMAGELEAF.

Usage

The TREE form item type can be used to specify the attributes of a tree container defined with a layout tag.

For more details about this item type, see TREE item type on page 899.

WEBCOMPONENT item definition
Defines a generic form field that can receive an external widget, in a grid-based layout.

Syntax

```
WEBCOMPONENT item-tag = field-name [ , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *field-name* identifies the name of the screen record field.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

COLOR, COLOR WHERE, COMPONENTTYPE, COMMENT, DEFAULT, FONTPITCH, HEIGHT, HIDDEN, INCLUDE, JUSTIFY, NOT NULL, NOENTRY, PROPERTIES, REQUIRED, SCROLLBARS, SIZEPOLICY, STYLE, STRETCH, TAG, TABINDEX, TITLE, UNSORTABLE, UNSIZABLE, UNHIDABLE, UNMOVABLE, VALIDATE LIKE, WIDTH.

Usage

Define the rendering and behavior of a web component item tag, with an WEBCOMPONENT element in the ATTRIBUTES section.

For more details about this item type, see WEBCOMPONENT item type on page 900.

Example

```
LAYOUT
GRID
{
[f1             ]
[               ]
[               ]
 ...
[f2             ]
[               ]
[               ]
 ...

}
END
END

ATTRIBUTES

-- URL-based web component (recommended):
WEBCOMPONENT f1 = FORMONLY.mymap,
     STRETCH=BOTH;

-- gICAPI web component:
WEBCOMPONENT f2 = FORMONLY.mycal,
     COMPONENTTYPE="Calendar", -- lookup "Calendar.html"
     STRETCH=BOTH, STYLE="regular";
```

INSTRUCTIONS section

The INSTRUCTIONS section is used to define screen arrays, non-default screen records and global form properties.

Syntax

```
INSTRUCTIONS
{ screen-record-definition [;...] }
[ DELIMITERS AB [;] ]
[ DEFAULT SAMPLE = "string" ]
[END]
```

1. *screen-record-definition* is the definition of a screen record or screen array.
2. *A* and *B* define the opening and closing field delimiters for character based terminals.

Usage

The INSTRUCTIONS section must appear in the sequence described in form file structure.

The INSTRUCTIONS section is optional in a form definition.

The END keyword is optional.

This section is mainly used to define screen records, to group fields using tables, tree views, scrollgrids or traditional static field arrays.

Screen records (or screen arrays)

A *screen record* is a named group of form fields.

See Screen records on page 866 for more details.

Field delimiters

Use the DELIMITER keyword to specify the characters to be displayed as field delimiters on the screen.

This option is especially used for TUI mode applications.

Default sample

The DEFAULT SAMPLE directive defines the default sample text for all fields.

```
DEFAULT SAMPLE = "MMM"
```

See SAMPLE attribute on page 981 for more details.

Example

```
SCHEMA stores
LAYOUT
GRID
{
  ...
}
END
TABLES
 stock, items
END
ATTRIBUTES
...
END
INSTRUCTIONS
  SCREEN RECORD s_items[10]
```

```
    ( stock.*,
      items.quantity,
      FORMONLY.total_price )
  DELIMITERS "[]"
END
```

KEYS section
The KEYS section can be used to define default key labels for the current form.

Syntax

```
KEYS
key-name = [%]"label"
[...]
[END]
```

1. *key-name* is the name of a key (like `F10`, `Control-z`).
2. *label* is the text to be displayed in the button corresponding to the key.

Usage

The KEYS section can be used to define default key labels at the form level.

The KEYS section must appear in the sequence described in form file structure.

The KEYS section is optional in a form definition.

The END keyword is optional.

> **Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

Example

```
KEYS
  F10 = "City list"
  F11 = "State list"
  F15 = "Validate"
END
```

Form item attributes
The form item attributes reference.

- ACCELERATOR attribute on page 953
- ACCELERATOR2 attribute on page 953
- ACCELERATOR3 attribute on page 954
- ACCELERATOR4 attribute on page 954
- ACTION attribute on page 954
- AGGREGATETEXT attribute on page 955
- AGGREGATETYPE attribute on page 955
- AUTONEXT attribute on page 956
- AUTOSCALE attribute on page 956
- BUTTONTEXTHIDDEN attribute on page 956
- CENTURY attribute on page 956
- CLASS attribute on page 957
- COLOR attribute on page 957
- COLOR WHERE Attribute on page 958

- COMMENT attribute on page 959
- COMPONENTTYPE attribute on page 960
- CONFIG Attribute on page 958
- CONTEXTMENU attribute on page 958
- DEFAULT attribute on page 960
- DEFAULTVIEW attribute on page 961
- DISPLAY LIKE attribute on page 961
- DISCLOSUREINDICATOR attribute on page 962
- DOUBLECLICK attribute on page 962
- DOWNSHIFT attribute on page 962
- EXPANDEDCOLUMN attribute on page 963
- FONTPITCH attribute on page 963
- FORMAT attribute on page 963
- GRIDCHILDRENINPARENT attribute on page 964
- HEIGHT attribute on page 965
- HIDDEN attribute on page 965
- IDCOLUMN attribute on page 966
- IMAGE attribute on page 967
- IMAGECOLLAPSED attribute on page 968
- IMAGECOLUMN attribute on page 967
- IMAGEEXPANDED attribute on page 968
- IMAGELEAF attribute on page 969
- INCLUDE attribute on page 969
- INITIALIZER attribute on page 970
- INVISIBLE attribute on page 970
- ISNODECOLUMN attribute on page 971
- ITEMS attribute on page 971
- JUSTIFY attribute on page 972
- KEY attribute on page 973
- MINHEIGHT attribute on page 975
- MINWIDTH attribute on page 975
- NOENTRY attribute on page 975
- NOT NULL attribute on page 976
- NOTEDITABLE attribute on page 976
- OPTIONS attribute on page 977
- ORIENTATION attribute on page 977
- PARENTIDCOLUMN attribute on page 977
- PICTURE attribute on page 977
- PROGRAM attribute on page 978
- PROPERTIES attribute on page 979
- QUERYEDITABLE attribute on page 979
- REQUIRED attribute on page 980
- REVERSE attribute on page 981
- SAMPLE attribute on page 981
- SCROLL attribute on page 982
- SCROLLBARS attribute on page 982
- SIZEPOLICY attribute on page 982
- SPACING attribute on page 984
- SPLITTER attribute on page 985
- STEP attribute on page 985

- STRETCH attribute on page 985
- STYLE attribute on page 986
- TABINDEX attribute on page 986
- TAG attribute on page 987
- TEXT attribute on page 987
- TITLE attribute on page 988
- UNHIDABLE attribute on page 989
- UNHIDABLECOLUMNS attribute on page 990
- UNMOVABLE attribute on page 990
- UNMOVABLECOLUMNS attribute on page 990
- UNSIZABLE attribute on page 989
- UNSIZABLECOLUMNS attribute on page 989
- UNSORTABLE attribute on page 988
- UNSORTABLECOLUMNS attribute on page 988
- UPSHIFT attribute on page 991
- VALIDATE attribute on page 991
- VALIDATE LIKE attribute on page 991
- VALUECHECKED attribute on page 993
- VALUEMAX attribute on page 992
- VALUEMIN attribute on page 992
- VALUEUNCHECKED attribute on page 993
- VERIFY attribute on page 993
- VERSION attribute on page 994
- WANTFIXEDPAGESIZE attribute on page 994
- WANTNORETURNS attribute on page 995
- WANTTABS attribute on page 995
- WIDGET attribute on page 995
- WIDTH attribute on page 999
- WINDOWSTYLE attribute on page 999
- WORDWRAP Attribute on page 1000

ACCELERATOR attribute
The `ACCELERATOR` is an action attribute defining the primary accelerator key for an action.

Syntax

```
ACCELERATOR = key
```

1. *key* defines the accelerator key.

Usage

This attribute is an action attribute that can be specified in form `ACTION DEFAULTS`, for more details, see ACCELERATOR action attribute on page 1323.

ACCELERATOR2 attribute
The `ACCELERATOR2` is an action attribute defining the secondary accelerator key for an action.

Syntax

```
ACCELERATOR2 = key
```

1. *key* defines the accelerator key.

Usage

This attribute is an action attribute that can be specified in form `ACTION DEFAULTS`, for more details, see ACCELERATOR2 action attribute on page 1324.

ACCELERATOR3 attribute
The `ACCELERATOR3` is an action attribute defining the third accelerator key for an action.

Syntax

```
ACCELERATOR3 = key
```

1. *key* defines the accelerator key.

Usage

This attribute is an action attribute that can be specified in form `ACTION DEFAULTS`, for more details, see ACCELERATOR3 action attribute on page 1325.

ACCELERATOR4 attribute
The `ACCELERATOR4` is an action attribute defining the fourth accelerator key for an action.

Syntax

```
ACCELERATOR4 = key
```

1. *key* defines the accelerator key.

Usage

This attribute is an action attribute that can be specified in form `ACTION DEFAULTS`, for more details, see ACCELERATOR4 action attribute on page 1325.

ACTION attribute
The `ACTION` attribute defines the action associated to the form item.

Syntax

```
ACTION = action-name
```

1. *action-name* is an identifier that defines the name of the action to be sent.

Usage

The `ACTION` attribute defines the name of the action to be sent to the program when the user activates the form item.

This attribute can for example be used in a `BUTTONEDIT` field to identify the corresponding action handle to be executed in the program when the button is pressed.

The action name can be prefixed with a sub-dialog identifier and/or field name, to define a qualified action view (see action handler binding rules for more details).

Example

```
BUTTONEDIT f001 = customer.state, ACTION = print;
```

AGGREGATETEXT attribute

The AGGREGATETEXT attribute defines a label to be displayed for aggregate fields.

Syntax

```
AGGREGATETEXT = [%]"string"
```

1. *string* defines the label to be associated with the aggregate cell, with the % prefix it is a localized string.

Usage

The AGGREGATETEXT attribute can be specified at the AGGREGATE field level, or globally at the TABLE level, to define a label for the whole summary line. When defining the AGGREGATETEXT attribute at the aggregate field level, the text will be anchored to the value cell. If the AGGREGATETEXT attribute is specified at the TABLE level, the label will appear on the left in the summary line. When an aggregate text is defined at both levels, the global aggregate text of the table will be ignored.

Example

```
AGGREGATE tot = FORMONLY.total, AGGREGATETEXT="Total:";
```

AGGREGATETYPE attribute

The AGGREGATETYPE attribute defines how the aggregate field value is computed.

Syntax

```
AGGREGATETYPE = { PROGRAM | SUM | AVG | MIN | MAX | COUNT }
```

Usage

PROGRAM specifies that the aggregate value will be computed and displayed by the program code.

An aggregate type different from PROGRAM specifies that the aggregate value is computed automatically:

- SUM computes the total of all values of the corresponding numeric column.
- AVG computes the average of all values of the corresponding numeric column.
- MIN displays the minimum value of the corresponding numeric column.
- MAX displays the maximum value of the corresponding numeric column.
- COUNT computes the number of rows.

The SUM and AVG aggregate types apply to data types that can be used as operand for an addition, such as INTEGER, DECIMAL, INTERVAL.

The MIN and MAX aggregate types apply to data types that can be compared, such as INTEGER, DECIMAL, INTERVAL, CHAR, DATETIME.

Example

```
AGGREGATE tot = FORMONLY.total, AGGREGATETYPE=PROGRAM;
```

AUTOSCALE attribute
The AUTOSCALE attribute causes the form element contents to automatically scale to the size given to the item.

Syntax

```
AUTOSCALE
```

Usage

For images, this attribute forces the image to be stretched to fit in the area reserved for the image.

AUTONEXT attribute
The AUTONEXT attribute forces the cursor to automatically leave the current field when full.

Syntax

```
AUTONEXT
```

Usage

With AUTONEXT, when the user types a character that completely fills the current field, the focus goes automatically to the next field in the input order.

If data values entered in the field do not meet the requirements of other field attributes like INCLUDE or PICTURE, the cursor does not automatically move to the next field. It remains in the current field, and an error message displays.

AUTONEXT is particularly useful with character fields in which the input data is of a standard length, such as numeric postal codes. It is also useful if a character field has a length of 1, as only one keystroke is required to enter data and move to the next field.

BUTTONTEXTHIDDEN attribute
The BUTTONTEXTHIDDEN attribute indicates that the button labels for an element should not be displayed.

Syntax

```
BUTTONTEXTHIDDEN
```

Usage

Use BUTTONTEXTHIDDEN in a TOOLBAR definition to hide the labels of toolbar buttons.

CENTURY attribute
The CENTURY attribute defines expansion of the year in a DATE or DATETIME field.

Syntax

```
CENTURY = { "R" | "C" |"F" | "P" }
```

Usage

The CENTURY attribute specifies how to expand abbreviated one- and two-digit *year* specifications in a DATE and DATETIME field.

Century expansion is based on this attribute and on the current year defined by the system clock.

The CENTURY attribute can specify any of four algorithms to expand abbreviated years into four-digit year values that end with the same digits (or digit) that the user has entered.

CENTURY supports the same settings as the DBCENTURY environment variable, but with a scope that is restricted to a single field.

If the CENTURY and DBCENTURY settings are different, CENTURY takes precedence.

Unlike DBCENTURY, the CENTURY attribute is not case sensitive. However, we recommend that you use uppercase letters in the attribute.

CLASS attribute
The CLASS attribute defines the behavior of a field defined with the WIDGET attribute.

Syntax

```
CLASS = "identifier"
```

1. *identifier* is a predefined keyword defining the class of the field.

Usage

The CLASS attribute can only be used with the WIDGET attribute. It is ignored if WIDGET is not used.

> **Important:** This attribute is deprecated, use new form item types instead.

Table 258: Supported field classes

Class	Description
KEY	Field is used to trigger a keystroke instead of being a normal input field. Only supported with WIDGET="BMP"|"CHECK"|"RADIO"
PASSWORD	Field input is masked by replacing normal character echo by stars.

COLOR attribute
The COLOR attribute defines the foreground color of the text displayed by a form element.

Syntax

```
COLOR = color-name
```

1. *color-name* can be: BLACK, BLUE, CYAN, GREEN, MAGENTA, RED,WHITE, and YELLOW.

Usage

The COLOR attribute defines the logical color of a value displayed in a field.

For backward compatibility, *color-name* can be combined with an intensity keyword: REVERSE, LEFT, BLINK, and UNDERLINE.

Example

```
EDIT f001 = customer.name, COLOR = RED;
```

COLOR WHERE Attribute

The `COLOR WHERE` attribute defines a condition to set the foreground color dynamically.

Syntax

```
COLOR = color-name [...] WHERE bool-expr
```

1. *color-name* can be `BLACK`, `BLUE`, `CYAN`, `GREEN`, `MAGENTA`, `RED`, `WHITE`, or `YELLOW`.
2. *color-name* can also be an intensity keyword: `REVERSE`, `LEFT`, `BLINK`, and `UNDERLINE`.
3. *bool-expr* defines a boolean expression with a restricted syntax.

Usage

The `COLOR WHERE` attribute defines the logical color of the text of a field when the value satisfies the conditional expression.

The condition in `COLOR WHERE` can only reference the field for which the attribute is set.

The boolean expression is automatically evaluated at runtime to check when the color attribute must be set.

Example

```
EDIT f001 = item.price, COLOR=RED WHERE f001 > 100;
```

CONFIG Attribute

The `CONFIG` attribute defines the behavior and decoration of a field defined with the `WIDGET` attribute.

Syntax

```
CONFIG = "parameter [...]"
```

1. *parameter* is the value of a configuration parameter.

Usage

The `CONFIG` attribute can only be used with the `WIDGET` attribute. It is ignored if `WIDGET` is not used.

Configuration parameters are separated by blanks.

If a configuration parameter holds blank characters, you must use `{}` curly braces to delimit the parameter value.

Important: This attribute is deprecated, use new form item types instead.

CONTEXTMENU attribute

The `CONTEXTMENU` attribute defines whether a context menu option must be displayed for an action.

Syntax

```
CONTEXTMENU = [ AUTO | YES | NO ]
```

Usage

This attribute is an action attribute that can be specified in form `ACTION DEFAULTS`, for more details, see CONTEXTMENU action attribute on page 1326.

COMMENT attribute

The COMMENT attribute defines hint for the user about the form element.

Syntax

```
COMMENT = [%]"string"
```

1. *string* is the text to display, with the % prefix it is a localized string.

Usage

The most common use of the COMMENT attribute is to give information or instructions to the user.

The COMMENT attribute can be used for different sort of form elements:

- Form field definitions, to show a message when the field gets the focus.
- Action views, to give a hint to the user about the action.

With form fields, this attribute is particularly appropriate when the field accepts only a limited set of values. The screen location where the message is displayed depends on external configuration. It can be displayed in the comment line, or in the status bar when using a graphical user interface. If the OPEN WINDOW statement specifies COMMENT LINE OFF, any output to the comment area is hidden even if the window displays a form that includes fields that include the COMMENT attribute.

This attribute is also an action attribute that can be defined in the ACTION DEFAULTS section of a form or directly in an action view (BUTTON), see COMMENT action attribute on page 1325 for more details.

Example

```
-- As action default
ACTION DEFAULTS
  ACTION print (COMMENT="Print current order information")
END

-- In a form field definition
EDIT f1 = customer.name, COMMENT = "The customer name";

-- In a form buttom
BUTTON b1: print, COMMENT = "Print customer details";
```

COMPLETER attribute

The COMPLETER attribute enables autocompletion for the edit field.

Syntax

```
COMPLETER
```

Usage:

Form fields with COMPLETER attribute provide suggestions while the end-user types text into the field, it can be used in text edit fields such as EDIT and BUTTONEDIT item types.

Normally, the ON CHANGE trigger is fired for text edit fields when leaving the field and if the content was modified. Form fields defined with the COMPLETER attribute will trigger the ON CHANGE control block when the end user modifies the content of the field.

See Enabling autocompletion on page 1274 for more details.

Example

```
EDIT f1 = FORMONLY.custname, COMPLETER;
```

COMPONENTTYPE attribute

The COMPONENTTYPE attribute defines a name identifying the external widget for WEBCOMPONENT fields.

Syntax

```
COMPONENTTYPE = "name"
```

1. *name* defines the HTML file defining the web component.

Usage

The COMPONENTTYPE attribute is used to define the type of a WEBCOMPONENT form item for gICAPI web components.

When this attribute is specified, it defines the name of the HTML file that will be loaded by the front-end. If this attribute is not defined, the web component will be specified by an URL set dynamically by program in the field value. Consider using URL-based web components instead of gICAPI web components.

Example

```
WEBCOMPONENT f001 = FORMONLY.mycal, COMPONENTTYPE="Calendar";
```

DEFAULT attribute

The DEFAULT attribute assigns a default value to a field during data entry.

Syntax

```
DEFAULT = value
```

1. *value* can be any literal expression supported by the form compiler, as long as it matches the form field type.
2. *value* can be TODAY to specify the current system date as default.
3. *value* can be CURRENT to specify the current system datetime as default.

Usage

The literal constant specified after as default value must match the form field type. For example, when defining a numeric field, use a numeric decimal constant, for character string fields, use a double-quoted character literal.

The effect of the DEFAULT attribute depends on the WITHOUT DEFAULTS configuration option of the dialog using the form:

With the INPUT statement, form default values have are ignored when using the WITHOUT DEFAULTS option. With this option, the runtime system displays the values in the program variables to the screen. Otherwise, the form default values will be displayed when the dialog starts.

With the INPUT ARRAY statement, the form default values are always used for new rows inserted by the user. With INPUT ARRAY, the WITHOUT DEFAULTS option indicates if the existing program array elements have to be used.

Defaults values can also be specified in the database schema file, for form fields defined with database column reference.

If the field is FORMONLY, you must also specify a data type when you assign the DEFAULT attribute to a field.

If both the DEFAULT attribute and the REQUIRED attribute are assigned to the same field, the REQUIRED attribute is ignored.

If you do not use the WITHOUT NULL INPUT option in the DATABASE section of a form, all fields default to null values unless you have specified a DEFAULT attribute.

Note that DATETIME and INTERVAL literals are not supported in the DEFAULT attribute.

Example

```
EDIT f001 = order.orderdate, DEFAULT = TODAY;
EDIT f012 = FORMONLY.discount TYPE DECIMAL(5,2), DEFAULT=0.10;
```

DEFAULTVIEW attribute

The DEFAULTVIEW attribute defines if a default view (a button) must be displayed for a given action.

Syntax

```
DEFAULTVIEW = [ AUTO | YES | NO ]
```

Usage

This attribute is an action attribute that can be specified in form ACTION DEFAULTS, for more details, see DEFAULTVIEW action attribute on page 1327.

DISPLAY LIKE attribute

The DISPLAY LIKE attribute applies column attributes defined in the database schema files (.att) to a field.

Syntax

```
DISPLAY LIKE [table.]column
```

1. *table* is the optional table name to qualify the column.
2. *column* is the name of the column to be used to retrieve display attributes.

Usage

Specifying this attribute is equivalent to listing all the attributes that are assigned to *table.column* in the database schema file with the .att extension.

Display attributes are automatically taken from the schema file if the field is linked to *table.column* in the field name specification.

The DISPLAY LIKE attribute is evaluated at compile time, not at runtime. If the database schema file changes, recompile all forms using this attribute. Even if all of the fields in the form are FORMONLY, this attribute requires the form compiler to access the database schema file that contains the description of *table*.

Example

```
EDIT f001 = FORMONLY.fullname, DISPLAY LIKE customer.custname;
```

DISCLOSUREINDICATOR attribute

The DISCLOSUREINDICATOR attribute adds a drill-down decoration to the form item.

Syntax

```
DISCLOSUREINDICATOR
```

Usage

The DISCLOSUREINDICATOR attribute is used on BUTTON form items to add a graphical hint, to indicate that a click on the button will drill down in the application windows, typically to show a detail view of the information displayed in the current window.

This is a simple decoration attribute, with no other functional purpose.

For example, on iOS devices, the buttons defined with this attribute will show a typical > icon on the right.

Example

```
BUTTON b_details : details,
   TEXT="Show details",
   DISCLOSUREINDICATOR;
```

DOUBLECLICK attribute

The DOUBLECLICK attribute defines the action for double-clicks or tap on TABLE/TREE rows.

Syntax

```
DOUBLECLICK = action-name
```

1. *action-name* defines the name of the action to be invoked.

Usage

> **Note:** The double-click/tap action can also be defined as DISPLAY ARRAY dialog attribute. For more details, see Defining the action for a row choice on page 1360.

The DOUBLECLICK attribute is typically used in a TABLE or TREE container, to define the action to be sent when the user double-clicks on a row on a front-end using a mouse device. On mobile front-ends, this attribute corresponds to the action of tapping on the row with the finger.

By default, when the TABLE is driven by a DISPLAY ARRAY, a double-click invokes the "accept" action.

With an INPUT ARRAY, double-click selects the whole text if the current widget is editable. If DOUBLECLICK is defined when using an INPUT ARRAY, the action can only be sent when the user double-clicks on a non-editable widget like a LABEL. It is not recommended to define this attribute when an INPUT ARRAY dialog is used.

DOWNSHIFT attribute

The DOWNSHIFT attribute forces character input to lowercase letters.

Syntax

```
DOWNSHIFT
```

Usage

Assign the DOWNSHIFT attribute to a character field to automatically convert uppercase letters entered by the user to lowercase letters.

Because uppercase and lowercase letters have different values, storing character strings in one or the other format can simplify sorting and querying a database.

The results of conversions between uppercase and lowercase letters are based on the locale settings.

EXPANDEDCOLUMN attribute
The EXPANDEDCOLUMN attribute specifies the form field that indicates whether a tree node is expanded.

Syntax

```
EXPANDEDCOLUMN = column-name
```

1. *column-name* is the name of the form field holding the flag indicating whether a tree node is expanded (opened.)

Usage

This attribute is used in the definition of a TREE container.

You must specify form field column names, not item tag identifiers.

This attribute is optional.

FONTPITCH attribute
The FONTPITCH attribute defines the character font type as fixed or variable when the default font is used.

Syntax

```
FONTPITCH = {FIXED|VARIABLE}
```

Usage

By default, most front ends use variable width character fonts, but some fields might need to use a fixed font.

> **Tip:** Use a STYLE defining a fixed font instead of this attribute.

FORMAT attribute
The FORMAT attribute defines the data formatting for numeric and date time fields, for input and display.

Syntax

```
FORMAT = "format"
```

1. *format* is a string of characters that specifies a data format.

Usage

The FORMAT attribute can be set to define a input and display format for numeric and date fields.

When this attribute is not used, environment variables define the default format:

- For MONEY and numeric fields such as DECIMAL fields, a format can be specified with the DBFORMAT (or DBMONEY) environment variables.
- For DATE fields, the default format is defined by the DBDATE environment variable.

The data format is used when converting the input buffer to the program variable, and when displaying program variable data to form fields. For example, when defining a `FORMAT="yyyy-mm-dd"` for a form field bound to a program variable defined as a `DATE`, the user can input a date as `2013-12-24`, and the date value will be displayed in the same manner.

Do not confuse the `FORMAT` and `PICTURE` attributes: The `PICTURE` attribute is used to define an input mask for character string fields, such as vehicle registration numbers. Do not mix `PICTURE` and `FORMAT` attributes in field defintions.

If the format string is smaller than the field width, you get a compile-time warning, but the form is usable.

The format string can be any valid string expression using formatting characters as described in Formatting numeric values on page 217 and Formatting DATE values on page 220.

Example

```
EDIT f001 = order.thedate, FORMAT = "mm/dd/yyyy";
```

GRIDCHILDRENINPARENT attribute

The `GRIDCHILDRENINPARENT` attribute is used for a container to align its children to the parent container.

Syntax

```
GRIDCHILDRENINPARENT
```

Usage

By default, in a grid-based layout, child elements of a container are aligned locally inside the container layout cells. With the `GRIDCHILDRENINPARENT` attribute, you can force children to be aligned in the vertical or horizontal direction, according to the layout cells in the parent container of the container to which you assign this attribute.

Important: This feature is not supported on mobile platforms.

Note: The `GRIDCHILDRENINPARENT` attribute applies only to `GROUP` and `SCROLLGRID` containers used inside a parent `GRID` container.

When the group or scrollgrid containers are placed vertically over each other, the alignment applies on parent grid columns, and when the containers are placed side by side horizontally, the alignment applies on parent grid rows.

Example

With the next form definition, the elements in the four group boxes will align vertically and horizontally to the parent grid cells:

```
LAYOUT
GRID
{
<G ga       ><G gb       >
 Some text
[a          ]  b[b     ]
<           ><           >
<G gc       ><G gd       >
[c          ]  d[d     ]
<           ><           >
}
END
END
ATTRIBUTES
```

```
GROUP ga: GRIDCHILDRENINPARENT;
GROUP gb: GRIDCHILDRENINPARENT;
GROUP gc: GRIDCHILDRENINPARENT;
GROUP gd: GRIDCHILDRENINPARENT;
EDIT a = FORMONLY.f_a;
EDIT b = FORMONLY.f_b;
EDIT c = FORMONLY.f_c;
EDIT d = FORMONLY.f_d;
END
```

HIDDEN attribute

The `HIDDEN` attribute indicates that the element should not be displayed.

Syntax

```
HIDDEN [ = USER ]
```

1. `HIDDEN` sets the underlying item attribute to 1.
2. `HIDDEN=USER` sets the underlying item attribute to 2.

Usage

By default, all form elements are visible. Specify the `HIDDEN` attribute to hide a form element, such as a form field or a groupbox.

The runtime system detects hidden form fields: If you write an `INPUT` statement using a hidden field, the field is ignored (as if it was declared as `NOENTRY`).

If the `HIDDEN` keyword is specified alone, the underlying item attribute is set to 1. The value 1 indicates that the element is definitvely hidden to the end user, which cannot show the element, for example with the context menu of `TABLE` headers. In this hidden mode, the `UNHIDABLE` attribute is ignored by the front end.

With `HIDDEN=USER`, the underlying item attribute is set to 2. The value 2 indicates that the element is hidden by default, but the end user can show/hide the element as needed. For example, the user can change a hidden column back to visible. Form elements like table columns that are hidden by the user might be automatically re-shown (hidden=0) by the front-end if the program dialog gives the focus to that field for input. In such case the program dialog takes precedence over the hidden attribute.

When you set a hidden attribute for a form field, the model node gets the hidden attribute, not the view node.

Form fields hidden with `HIDDEN=USER` (value 2) might be shown anyway, if the field is needed by a dialog for input.

Programs may also change the visibility of form elements dynamically with the `ui.Form.setElementHidden()` or `ui.Form.setFieldHidden()` methods.

Example

```
EDIT f001 = FORMONLY.field1, HIDDEN;
EDIT col1 = FORMONLY.column1, HIDDEN=USER;
```

HEIGHT attribute

The `HEIGHT` attribute defines an explicit height for a form element.

Syntax

```
HEIGHT = integer [CHARACTERS|LINES|POINTS|PIXELS]
```

1. *integer* defines the height of the element.

Usage

By default, the height of an element is defined by the size of the form item tag in a grid-based layout, or by the type of the form item in a stack-based layout. Use the `HEIGHT` attribute to define a specific height for a form item.

> **Note:** As a general rule, consider not specifying a unit, to default to relative characters/lines/columns, instead of specifying exact pixels or points. This is especially important for mobile devices, where the screen resolution can significantly vary according to the smartphone or tablet model.

In a grid-based layout and stack-based layout, if you don't specify a size unit, it defaults to `CHARACTERS`, which defines a height based on the characters size in the current font.

Grid-based layout

For sizable items like `IMAGE`, the default height is defined by the number of lines of the form item tag in the layout, as a vertical character height. Overwrite this default by specifying the `HEIGHT` attribute.

For `TABLE`/`TREE` containers, the default height is defined by the number of lines used in the table layout. Overwrite the default by specifying the `HEIGHT = x LINES` attribute.

```
IMAGE img1: image1, WIDTH = 20, HEIGHT = 12;
```

Stack-based layout

For `TABLE` containers, the height of a list is defined by the actual number of rows, this cannot be changed.

For `IMAGE` items, by default the image is rendered full size, which means that the actual size of the image is used. Overwrite the default by specifying the `HEIGHT` attribute:

```
IMAGE image1, HEIGHT = 12, ...;
```

By default, `WEBCOMPONENT` items adapt their size to the content. To force a give size, use the `HEIGHT` attribute:

```
WEBCOMPONENT FORMONLY.chart, HEIGHT = 10, ...;
```

A `TEXTEDIT` item always adapts its size to the text value. By using the `HEIGHT` attribute, you can define a minimum height, when the value of the field is empty:

```
TEXTEDIT FORMONLY.comment, HEIGHT = 5, ...;
```

IDCOLUMN attribute

The `IDCOLUMN` attribute specifies the form field that contains the identifier of a tree node.

Syntax

```
IDCOLUMN = column-name
```

1. *column-name* is a form field name.

Usage

This attribute is used in the definition of a `TREE` container, to define the name of the form field containing the identifier of a node in a tree view

You must specify form field column names, not item tag identifiers.

This attribute is mandatory.

IMAGE attribute
The IMAGE attribute defines the image resource to be displayed for the form item.

Syntax

```
IMAGE = "resource"
```

1. *resource* defines the file name, path or URL to the image source.

Usage:

The IMAGE attribute is used to define the image resource to be displayed form items such a BUTTON, BUTTONEDIT, a TOOLBAR button or a static IMAGE item.

For more details about image resource specification, see Providing the image resource on page 784.

This attribute is also an action attribute that can be defined in the ACTION DEFAULTS section of a form or directly in an action view (BUTTON), see IMAGE action attribute on page 1328 for more details.

Example

```
-- As action default
ACTION DEFAULTS
  ACTION print (IMAGE="printer")
END

-- In a form buttonedit or button
BUTTONEDIT f001 = FORMONLY.field01, IMAGE = "zoom";
BUTTON b01: open_file, IMAGE = "buttons/fileopen";
BUTTON b02: accept, IMAGE = "http://myserver/images/accept.png";

-- In a static image form item
IMAGE: img1, IMAGE = "mylogo.png"
```

IMAGECOLUMN attribute
The IMAGECOLUMN attribute defines the form field containing the image for the current field.

Syntax

```
IMAGECOLUMN = column-name
```

1. *column-name* is a form field name.

Usage

The IMAGECOLUMN attribute allows displaying an image on the left of the value of this column value. The image can be different for each row.

A typical usage is the TREE container: IMAGECOLUMN will allow to display a row-specific image left of the tree node text. You defined only one image column for a tree node decoration.

When used in the definition of a TABLE column, the image and the column will be displayed in the same table cell. There can be several TABLE columns using an IMAGECOLUMN.

For TREE containers, the images defined by the IMAGECOLLAPSED, IMAGEEXPANDED and IMAGELEAF attributes take precedence over the images defined by the IMAGECOLUMN cell.

This attribute references form field that contains the name of an image. This form field must be defined as a PHANTOM form field, that will be part of the screen record definition in the INSTRUCTIONS section.

For more details about image resource specification in the `PHANTOM` column, see Providing the image resource on page 784.

Example

```
...
ATTRIBUTES
PHANTOM FORMONLY.icon;
EDIT FORMONLY.file_name, IMAGECOLUMN=icon;
...
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.icon, FORMONLY.file_name, ...);
...
```

IMAGECOLLAPSED attribute

The `IMAGECOLLAPSED` attribute sets the global icon to be used when a tree node is collapsed.

Syntax

```
IMAGECOLLAPSED = "image-name"
```

1. *image-name* is an image resource.

Usage

This attribute is used in the definition of a `TREE` container, to define the icon to be used for nodes that are collapsed.

It overwrites the program array image defined by `IMAGECOLUMN`, if both are used.

This attribute is optional.

For more details about image resource specification, see Providing the image resource on page 784.

IMAGEEXPANDED attribute

The `IMAGEEXPANDED` attribute sets the global icon to be used when a tree node is expanded.

Syntax

```
IMAGEEXPANDED = "image-name"
```

1. *image-name* is an image resource.

Usage

This attribute is used in the definition of a `TREE` container, to define the icon to be used for nodes that are expanded.

It overwrites the program array image defined by `IMAGECOLUMN`, if both are used.

This attribute is optional.

For more details about image resource specification, see Providing the image resource on page 784.

IMAGELEAF attribute

The IMAGELEAF attribute defines the global icon for leaf nodes of a TREE container.

Syntax

```
IMAGELEAF = "image-name"
```

1. *image-name* is an image resource.

Usage

This attribute is used in the definition of a TREE container, to specify the name of the icon that must be used for leaf nodes.

It overwrites the program array image defined by IMAGECOLUMN, if both are used.

This attribute is optional.

For more details about image resource specification, see Providing the image resource on page 784.

INCLUDE attribute

The INCLUDE attribute defines a list of possible values for a field.

Syntax

```
INCLUDE = ( { NULL | literal [ TO literal] } [,...] )
```

1. *literal* can be any literal expression supported by the form compiler.

Usage

The INCLUDE attribute specifies acceptable values for a field and causes the runtime system to check the data before accepting an input value.

If the field is FORMONLY, you must also specify a data type when you assign the INCLUDE attribute to a field.

Include the NULL keyword in the value list to specify that it is acceptable for the user to leave the field without entering any value.

Use the TO keyword to specify an inclusive range of acceptable values. When specifying a range of values, the lower value must appear first. The field value is accepted if it is greater or equal to the first literal, and lower or equal to the second literal.

```
INCLUDE = (1 TO 999)
  is equivalent to:
( field_value >= 1 AND field_value <= 999 )
```

Special consideration must be taken for character string fields:

```
INCLUDE = ("AAA" TO "ZZZ")
  is equivalent to:
( field_value >= "AAA" AND field_value <= "ZZZ" )
  ABC is accepted
  A!! is not accepted
  Zaa is not accepted
```

When combining several ranges and single values, the value entered by the user is verified for each element of the INCLUDE attribute:

```
INCLUDE = (1 TO 999, -1, NULL)
```

```
  is equivalent to:
( field_value >= 1 AND field_value <= 999 )
OR
( field_value == -1 )
OR
( field_value IS NULL )
```

Example

```
EDIT f001 = compute.rate, INCLUDE = ( 1 TO 100, 200, NULL);
EDIT f002 = customer.state, INCLUDE = ( "AL" TO "GA", "IA" TO
 "WY" );
EDIT f003 = FORMONLY.valid TYPE CHAR, INCLUDE = ("Y","N");
```

INITIALIZER attribute

The `INITIALIZER` attribute allows you to specify an initialization function that will be automatically called by the runtime system to set up the form item.

Syntax

```
INITIALIZER = function
```

1. *function* is an identifier defining the program function to be called.

Usage

The initialization function must exist in the program using the form file and must be defined with a `ui.ComboBox` parameter.

The initialization function name is converted to lowercase by fglform.

> **Tip:** Consider defining the initialization function name in lowercase letters. The language syntax allows case-insensitive functions names, but to avoid mistakes, it is recommended to use a common naming convention with lowercase letters.

INVISIBLE attribute

The `INVISIBLE` attribute prevents user-entered data from being echoed on the screen during an interactive statement.

Syntax

```
INVISIBLE
```

Usage

The `INVISIBLE` attribute can be used for `EDIT` and `BUTTONEDIT` fields.

Characters that the user enters in a field with the `INVISIBLE` attribute are not displayed during data entry. Depending on the front end type, the typed characters are displayed using the blank, star, underscore or dot characters.

The `INVISIBLE` attribute has no effect when display data directly to a field with `DISPLAY TO` or `DISPLAY BY NAME`.

ISNODECOLUMN attribute
The `ISNODECOLUMN` attribute specifies the form field that indicates whether a tree node has children.

Syntax

```
ISNODECOLUMN = column-name
```

1. *column-name* is a form field name.

Usage

This attribute is used in the definition of a `TREE` container, to specify the name of the form field indicating whether a tree node has children.

Even if the program node does not contain child nodes for this tree node, this attribute may be used, to implement dynamic filling of tree views.

You must specify form field column names, not item tag identifiers.

This attribute is optional.

ITEMS attribute
The `ITEMS` attribute defines a list of possible values that can be used by the form item.

Syntax

```
ITEMS = { single-value-list | double-value-list }
```

where *single-value-list* is:

```
( value [,...])
```

where *double-value-list* is:

```
( ( value, label-value) [,...])
```

1. *single-value-list* is a comma-separated list of single values.
2. *double-value-list* is a comma-separated list of (a, b) values pairs within parentheses.
3. *value* is a numeric or string literal, or one of the following keywords: `NULL`, `TRUE`, `FALSE`.
4. *label-value* is a numeric literal, a string literal, or a localized string.

Usage

The list must be delimited by parentheses, and each element of the list can be a simple literal value or a pair of literal values delimited by parentheses.

This attribute is not used by the runtime system to validate the field, you must use the `INCLUDE` attribute to force the possible values.

This example defines a list of simple values:

```
ITEMS = ("Paris", "London", "New York")
```

This example defines a list of pairs:

```
ITEMS = ((1,"Paris"),(2,"London"),(3,"New York"))
```

This attribute can be used, for example, to define the list of a COMBOBOX form item:

```
COMBOBOX cb01 = FORMONLY.combobox01,
        ITEMS = ((1,"Paris"), (2,"London"),(3,"New York"));
```

In this example, the first value of a pair (1,2,3) defines the data values of the form field and the second value of a pair ("Paris", "London", "New York") defines the value to be displayed in the selection list.

When used in a RADIOGROUP form item, this attribute defines the list of radio buttons:

```
RADIOGROUP rg01 = FORMONLY.radiogroup01,
        ITEMS = ((1,"Paris"), (2,"London"),(3,"New York"));
```

In this case, the first value of a pair (1,2,3) defines the data values of the form field and the second value of a pair ("Paris", "London", "New York") defines the value to be displayed as the radio button label.

You can specify item labels with localized strings, but this is only possible when you specify a key and a label:

```
ITEMS = ((1,%"item1"),(2,%"item2"),(3,%"item3"))
```

It is allowed to define a NULL value for an item (An empty string is equivalent to NULL):

```
ITEMS = ((NULL,"Enter bug status"),(1,"Open"),(2,"Resolved"))
```

In this case, the behavior of the field depends from the item type used.

JUSTIFY attribute
The JUSTIFY attribute defines the justification of the content of a field and the alignment of table column headers.

Syntax

```
JUSTIFY = { LEFT | CENTER | RIGHT }
```

Usage

With the JUSTIFY attribute, you specify the justification of the content of a field as LEFT, CENTER or RIGHT when the field is in display state.

This attribute is ignored for input (i.e. when the field has the focus); only the default data justification rule applies when a field is in input state. The default data justification depends on the dialog type, the field data type and the FORMAT attribute. For example, a numeric field value is right aligned, while a string field is left aligned. The type of dialog also defines the default justification: In a CONSTRUCT, all input fields are left aligned, for search criteria input.

The JUSTIFY attribute can be used with all form item types: Additionally to the field content/data alignment, JUSTIFY defines the alignment of table column headers indirectly (i.e. table column header follows the alignment of field data). However, column header alignment in tables may not be enabled by default; Check the `headerAlignment` presentation style attribute for the Table class.

With mobile front-ends, tables are rendered as list views with a maximum of two visible columns. By default, the main and the comment columns are displayed vertically in each row (i.e. main is on top of the comment). Use JUSTIFY=RIGHT for the second column, in order to display columns side by side. Note that numeric fields are by default right justified and thus do not need that attribute to be set.

You can also specify the text alignment of static form labels with the JUSTIFY attribute.

Example

```
LABEL t01: TEXT="Hello!", JUSTIFY=RIGHT;
EDIT f01 = order.value, JUSTIFY=CENTER;
```

KEY attribute

The KEY attribute is used to define the labels of keys when the field is made current.

Syntax

```
KEY keyname = [%]"label"
```

1. *keyname* is the name of a key (like F10, "Control-z").
2. *label* is the text to be displayed in the button corresponding to the key.

Usage

Use the KEY attribute to define a label for the accelerator key corresponding to an action when the focus is in the field.

The *keyname* must be specified in quotes if you want to use Control / Shift / Alt key modifiers.

See the KEYS section to define key labels for the whole form.

> **Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

Example

```
EDIT f001 = customer.city, KEY F10 = "City list";
EDIT f002 = customer.state, KEY "Control-z" = "Open Zoom";
```

KEYBOARDHINT attribute

The KEYBOARDHINT attribute gives an indication on the kind of data the form field contains, to let the front-end adapt the keyboard accordingly.

Syntax

```
KEYBOARDHINT = { DEFAULT | EMAIL | NUMBER | PHONE }
```

Usage

The KEYBOARDHINT attribute can be used to give a hint to the front-end, regarding the kind of data the form field will contain. According to this hint, the front-end will open the virtual keyboard adapted to the data type, especially useful when designing application forms for mobile platforms.

Valid values for KEYBOARDHINT are:

- DEFAULT: No hint, the only hint is the data type of the program variable bound to the form field.
- EMAIL: The field is used to enter an e-mail address.
- NUMBER: The field is used to enter a numeric value.
- PHONE: The field is used to enter a phone number.

For example, when defining a numeric field with the attribute KEYBOARDHINT=NUMBER, the iOS device will display a numeric keyboard when entering data into that field.

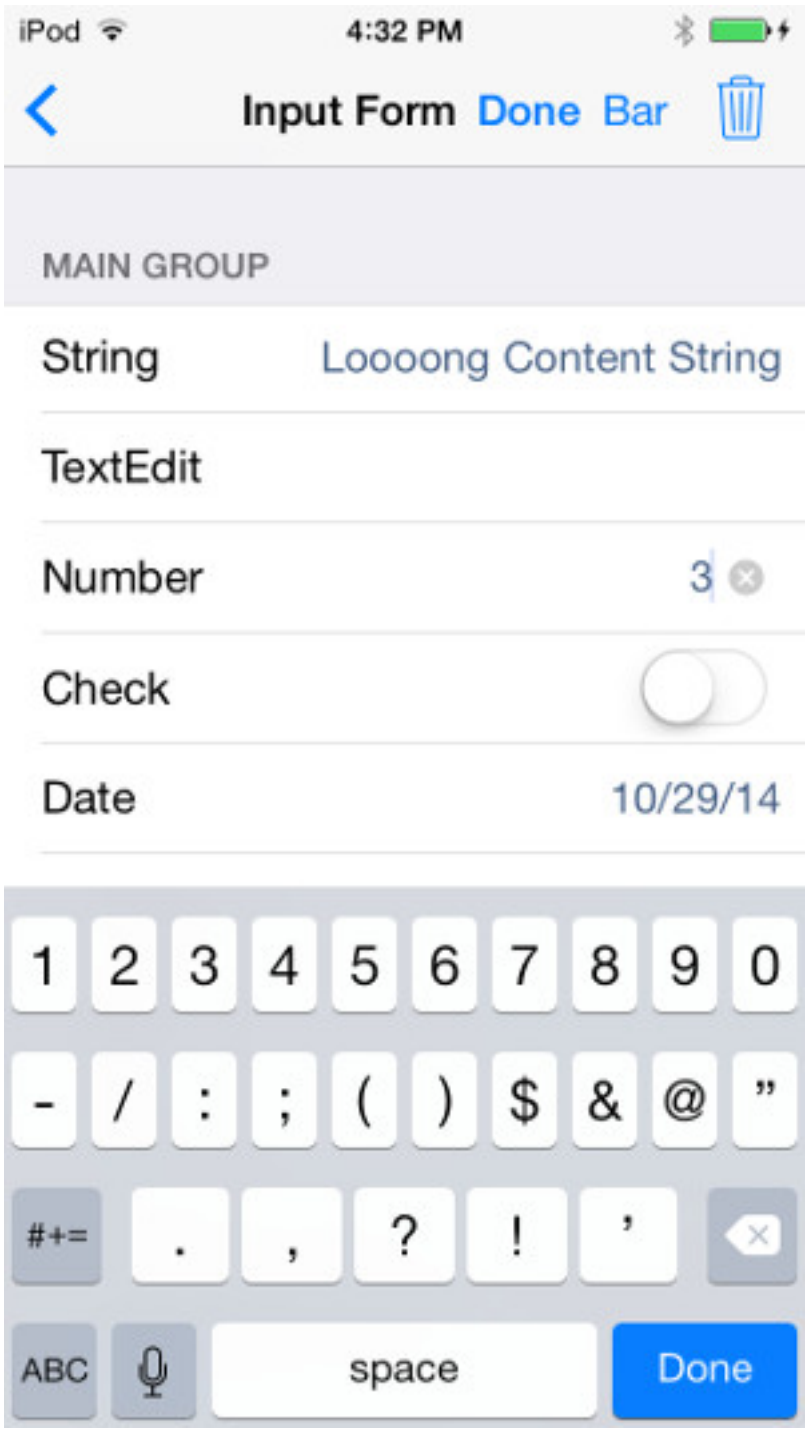


Figure 47: Mobile application using a numeric keyboard

Example

```
EDIT f23 = customer.cust_phone, KEYBOARDHINT=PHONE;
```

MINHEIGHT attribute
The MINHEIGHT attribute defines the minimum height of a form.

Syntax

```
MINHEIGHT = integer
```

1. *integer* defines the minimum height of the element, as a number of grid cells.

Usage

The MINHEIGHT attribute is used to define a minimum height for the form/window. It must be specified in the attributes of the LAYOUT section.

The unit defaults to a number of grid cells. This is the equivalent of the CHARACTERS in the HEIGHT attribute specification.

Example

```
LAYOUT ( MINWIDTH=60, MINHEIGHT=50 )
GRID
  ...
```

MINWIDTH attribute
The MINWIDTH attribute defines the minimum width of a form.

Syntax

```
MINWIDTH = integer
```

1. *integer* defines the minimum width of the element, as a number of grid cells.

Usage

The MINWIDTH attribute is used to define a minimum width for the form/window. It must be specified in the attributes of the LAYOUT section.

The unit defaults to a number of grid cells. This is the equivalent of the CHARACTERS in the WIDTH attribute specification.

Example

```
LAYOUT ( MINWIDTH=60, MINHEIGHT=50 )
GRID
  ...
```

NOENTRY attribute
The NOENTRY attribute prevents data entry in the field during an input dialog.

Syntax

```
NOENTRY
```

Usage

Use the NOENTRY attribute to bypass field input during an INPUT or INPUT ARRAY statement.

A NOENTRY field is like a disabled field, it cannot get the focus.

When compiling a form with a field referencing a SERIAL/BIGSERIAL column in the database schema, the NOENTRY attribute is automatically set. However, the attribute will not be set if the field is defined with a TYPE LIKE syntax.

When using a WITHOUT DEFAULTS dialog option, the content of the corresponding program variable is displayed in the field.

The NOENTRY attribute does not prevent data entry into a field during a CONSTRUCT statement.

Example

```
EDIT f001 = order.totamount, NOENTRY;
```

NOT NULL attribute

The NOT NULL attribute sets that the field does not accept NULL values.

Syntax

```
NOT NULL
```

Usage

The NOT NULL attribute requires that the field contains a non-null value. It can be specified explicitly in the form field definition, or in the corresponding column definition in the database schema file. If not column is associated to the field, the NOT NULL attribute can also be used in the type definition of FORMONLY fields.

The NOT NULL attribute is effective only when the field name appears in the list of screen fields of an INPUT or INPUT ARRAY statement.

If a DEFAULT attribute is used for the field and the input dialog does not use the WITHOUT DEFAULTS option, the runtime system assumes that the default value satisfies the NOT NULL attribute.

Unlike the REQUIRED attribute which has no effect when the INPUT dialog uses the WITHOUT DEFAULTS option, the NOT NULL attribute is always checked when validating a dialog.

Example

```
EDIT f001 = customer.city, NOT NULL;
```

NOTEDITABLE attribute

The NOTEDITABLE attribute disables the text editor.

Syntax

```
NOTEDITABLE
```

Usage:

The NOTEDITABLE attribute can be used in BUTTONEDIT field to disable the text editor. The button of the field remains active, if there is a corresponding active action handler in the current dialog. The field can still get the focus.

Use this attribute if you want to deny text edition in BUTTONEDIT fields, when the value can only be set by the action.

OPTIONS attribute
The OPTION attribute specifies widget options for the field.

Syntax

```
OPTIONS = "option [...]"
```

1. *option* can be -nolist (to indicate that the column should appear as an independent field).

Usage

The OPTIONS attribute specifies parameters for a form item defined with the WIDGET attribute.

Important: This attribute is deprecated, use new form item types instead.

ORIENTATION attribute
The ORIENTATION attribute defines whether an element displays vertically or horizontally.

Syntax

```
ORIENTATION = { VERTICAL | HORIZONTAL }
```

Usage

The ORIENTATION attribute is typically used in the definition of a RADIOGROUP form item, to specify how radio items have to be displayed.

Example

```
RADIOGROUP f001 = customer.status, ORIENTATION=HORIZONTAL;
```

PARENTIDCOLUMN attribute
The PARENTIDCOLUMN attribute specifies the form field that contains the identifier of the parent node of a tree node.

Syntax

```
PARENTIDCOLUMN = column-name
```

1. *column-name* is a form field name.

Usage

This attribute is used in the definition of a TREE container, to define the name of the form field containing the identifier of the tree node that is the parent of the current node in a tree view

You must specify form field column names, not item tag identifiers.

This attribute is mandatory.

PICTURE attribute
The PICTURE attribute specifies a character pattern for data entry in a text field, and prevents entry of values that conflict with the specified pattern.

Syntax

```
PICTURE = "format-string"
```

1. *format-string* defines the data input pattern of the field.

Usage

format-string can be any combination of characters, where the characters "A", "#" and "X" have a special meaning.

- The character "A" specifies any letter (alpha-numeric) character at a given position.
- The character "#" specifies any digit character at a given position.
- The character "X" specifies any character at a given position.

Any character different from "A", "X" and "#" is treated as a literal. Such characters automatically appear in the field and do not have to be entered by the user.

The `PICTURE` attribute does not require data entry into the entire field. It only requires that whatever characters are entered conform to *format-string*.

When `PICTURE` specifies input formats for `DATETIME` or `INTERVAL` fields, the form compiler does not check the syntax of *format-string*, but your form will work if the syntax is correct. Any error in *format-string*, however, such as an incorrect field separator, produces a runtime error.

The typical usage for the `PICTURE` attribute is for (fixed-length) `CHAR` fields. It is not recommended to use `PICTURE` for other data types, especially numeric or date/time fields: The current value of the field must always match (i.e. be formatted according to) `PICTURE`.

Understand that the `PICTURE` attribute defines a mask for data entry. In order to format fields when data is displayed to the field, use the `FORMAT` attribute instead. `FORMAT` is typically used for numeric and date fields, while `PICTURE` is typically used for formatted character string fields requiring input control.

The `PICTURE` attribute is ignored in `CONSTRUCT` and `DISPLAY` / `DISPLAY ARRAY` instructions: It takes only effect in `INPUT` and `INPUT ARRAY` dialogs.

Example

```
EDIT f001 = carinfo.ident, PICTURE = "AA####-AA(X)";
```

PROGRAM attribute

The `PROGRAM` attribute can specify an external application program to edit `TEXT` or `BYTE` fields.

Syntax

```
PROGRAM = "editor"
```

1. *editor* is the name of the program that must be used to edit the special field data.

Usage

You can assign the `PROGRAM` attribute to a `TEXT` or `BYTE` field to call an external program to work with the `BYTE` or `TEXT` values.

This attribute works in TUI mode only.

Users can invoke the external program by pressing the exclamation point (!) key while the screen cursor is in the field.

The external program then takes over control of the screen. When the user exits from the external program, the form is redisplayed with any display attributes besides `PROGRAM` in effect.

When no `PROGRAM` attribute is used, the `DBEDIT` environment variable defines the default editor.

PROPERTIES attribute

The PROPERTIES attribute is used to define a list of widget-specific characteristics.

Syntax

```
PROPERTIES = ( { single-property | array-property | map-property} [,...] )
```

where *single-property* is:

```
identifier = property-value
```

and *array-property* is:

```
identifier = ( property-value [,...])
```

and *map-property* is:

```
identifier = ( item-identifier = property-value[,...] )
```

and *property-value* is:

```
{ numeric-value | "string-value" }
```

1. *numeric-value* is an integer or decimal literal.
2. *string-value* is a string literal delimited by single or double quotes.

Usage

The PROPERTIES attribute is typically used to define the widget-specific attributes of a WEBCOMPONENT form item.

Property names and values are not checked, to let you freely set any characteristic of an external widget. You must verify that the front-end side implementation supports the specified properties.

Example

```
WEBCOMPONENT f01 = FORMONLY.mycalendar,
   COMPONENTTYPE = calendar,
   PROPERTIES = (
               type = "gregorian",
               week_start = 2,
               days_off = ( 1, 7 ),
               dates_off = ( "????-11-25", "????-06-20" ),
               day_titles = ( t1 = "Sunday",
                              t2 = "Monday",
                              t3 = "Tuesday",
                              t4 = "Wednesday",
                              t5 = "Thursday",
                              t6 = "Friday",
                              t7 = "Saturday"
             );
```

QUERYEDITABLE attribute

The QUERYEDITABLE attribute makes a COMBOBOX field editable during a CONSTRUCT statement.

Syntax

```
QUERYEDITABLE
```

Usage

The QUERYEDITABLE attribute is effective only during a CONSTRUCT statement. This attribute is useful when the display values match the real values in the ITEMS attribute, for example when ITEMS=("Paris","London","Madrid"). Do not use this attribute when the real field values are mapped to display values, for example when ITEMS=((1,"Paris"),(2,"London"),(3,"Madrid")).

During a CONSTRUCT, a COMBOBOX is not editable by default: The end-user is forced to set one of the values of the list as defined by the ITEMS attribute, or set the 'empty' item. The QUERYEDITABLE attribute can be used to force the COMBOBOX to be editable during a CONSTRUCT instruction, in order to allow free search criterion input such as "A*". If QUERYEDITABLE is used and the ITEMS are defined with key/label combinations, the text entered by the user will be automatically searched in the list of items. If a label corresponds, the key will be used in the SQL criterion, otherwise the text entered by the user will be used. For example, if the items are defined as ((1,"Paris"),(2,"Madrid"),(3,"London")), and the user enters "Paris" in the field, the item (1,"Paris") will match and will be generate "*colname = 1*". If the user enters ">2", the text does not match any item so it will be used as is and generate the SQL "*colname > 2*". Users may enter values like "Par*", but in this case the runtime system will raise an error because this criterion does is not valid for the numeric data type of the field. To avoid end-user confusion, a COMBOBOX defined with key/label combinations should not use the QUERYEDITABLE attribute.

REQUIRED attribute
The REQUIRED attribute forces the user to modify the content of a field during an input dialog.

Syntax

```
REQUIRED
```

Usage

The REQUIRED attribute forces the user to modify the content of a field controlled by an input dialog (INPUT or INPUT ARRAY), when the INPUT dialog does not use the WITHOUT DEFAULTS option. Within INPUT ARRAY, the REQUIRED attribute always applies to new created rows.

If an INPUT dialog uses the WITHOUT DEFAULTS clause, the current value of the variable linked to the REQUIRED field is considered as a default value; the runtime system assumes that the field satisfies the REQUIRED attribute, even if the variable value is null.

In an INPUT ARRAY dialog, the REQUIRED attribute always applies to new created rows, even if WITHOUT DEFAULTS is used. In other words, when creating a new row, INPUT ARRAY behaves like INPUT without the WITHOUT DEFAULTS clause.

If REQUIRED is effective regarding the WITHOUT DEFAULTS conditions, and a DEFAULT attribute is used for the field, the runtime system assigns the default value to the field and assumes that the REQUIRED attribute is satisfied.

The REQUIRED attribute does not prevent fields to be null; If the field contains a value, and the user subsequently erases this value during the same input, the runtime system considers the REQUIRED attribute satisfied. To insist on a non-null entry, use the NOT NULL attribute.

Example

```
EDIT f001 = orders.ord_shipcmt, REQUIRED;
```

REVERSE attribute

The REVERSE attribute displays any value in the field in reverse video (dark characters in a bright field).

Syntax

```
REVERSE
```

Usage

Use the REVERSE attribute to highlight specific fields in your forms.

On graphical front-ends, the REVERSE attribute is rendered by using the field COLOR attribute as background color.

On character based terminals, the REVERSE video escape sequences must be defined in the TERMINFO or TERMCAP databases.

Example

```
EDIT f001 = customer.name, COLOR = BLUE, REVERSE;
```

SAMPLE attribute

The SAMPLE attribute defines the text to be used to compute the width of a form field.

Syntax

```
SAMPLE = "text"
```

1. *text* is the sample string that will be used to compute the width of the field.

Usage

By default, form fields are rendered by the client with a size determined by the current font and the number of characters used in the layout grid. The field width is computed so that the largest value can fit in the widget.

> **Important:** This feature is not supported on mobile platforms.

Sometimes the default computed width is too wide for the typical values displayed in the field. For example, numeric fields usually need less space as alphanumeric fields. If the values are always smaller, you can use the SAMPLE attribute to provide a hint for the front end to compute the best width for that form field.

When specifying the SAMPLE attribute, you do not have to fill the sample string up to the width of the corresponding field tag: The front-ends will be able to compute a physical width by applying a ratio to fit the best visual result. For example, for a sample of 'XY' used for a field defined with 10 characters, is equivalent to specifying a sample of 'XYXYXYXYXY'.

If the SAMPLE attribute is not used, the first 6 cells are always computed with the pixel width of the 'M' character in the current font. Next cells are computed with the pixel width of the '0' (zero) character. In other words, the default sample model is 'MMMMMM000000.....", reduced to the size of the field tag in the layout:

```
   -123456789-123456789-         default sample
   [f01 ]                        MMMM
   [f02   ]                      MMMMMM
   [f03             ]            MMMMMM0000000000
```

You can define a default sample for all fields used in the form, by specifying a `DEFAULT SAMPLE` option in the `INSTRUCTIONS` section.

Example

```
EDIT cid = customer.custid, SAMPLE="0000";
EDIT ccode = customer.ucode, SAMPLE="MMMMMM";
DATEEDIT be01 = customer.created, SAMPLE="00-00-0000";
```

SCROLL attribute

The `SCROLL` attribute can be used to enable horizontal scrolling in a character field.

Syntax

```
SCROLL
```

Usage

By default, the maximum data input length is defined by the width of the item-tag of the field. For example, if you define an `CHAR` field in the form with a length of 3 characters, users can only enter a maximum of 3 characters, even if the program variable used for input is a `CHAR(20)`.

If you want to let the user input more characters than the width of the item-tag of the field, use the `SCROLL` attribute.

The `SCROLL` attribute applies only to fields with character data input.

Use the `SCROLL` attribute only when the layout of the form does not allow to define an item tag that is large enough to hold all possible character string data that fits in the corresponding program variable. Understand that the end user can miss a part of the displayed data when the field is too small. Therefore, using the `SCROLL` attribute should be rare.

SCROLLBARS attribute

The `SCROLLBARS` attribute can be used to specify scrollbars for a form item.

Syntax

```
SCROLLBARS = { NONE | VERTICAL | HORIZONTAL | BOTH }
```

Usage

This attribute defines scrollbars for the form item, such as a `TEXTEDIT`.

Example

```
TEXTEDIT f001 = customer.fname, SCROLLBARS=BOTH;
```

SIZEPOLICY attribute

The `SIZEPOLICY` attribute is a sizing directive, according to the content of a form item.

Syntax

```
SIZEPOLICY = { INITIAL | FIXED | DYNAMIC }
```

Usage

The SIZEPOLICY attribute defines how the front-ends computes the size of some form elements, according to the content of the form field or form item.

The SIZEPOLICY applies only to leaf elements of the layout. It does not apply to containers. The attribute applies to form elements with sizable content, typically IMAGE, COMBOBOX, WEBCOMPONENT. Elements allowing user input such as EDIT, or elements where the size does not depend from the value content such as PROGRESSBAR, SLIDER do not use this attribute.

Note that the SIZEPOLICY attribute is ignored for columns used in TABLE or TREE containers: In list views, the size policy is implicitly defined by the cell (i.e., the size of the column in the form layout). The SIZEPOLICY attribute is also implicitly fixed for fields inside SCROLLGRID and GRID containers that are controlled by a list dialog, such as a DISPLAY ARRAY. With a list dialog, each row can have a different value, which would imply a different widget size for each row; this is not supported.

When the SIZEPOLICY is not specified, the default behavior depends on the type of the form item. See Table 259: Behavior of SIZEPOLICY=INITIAL, according to the form item type and front-end type on page 984.

SIZEPOLICY = FIXED

When SIZEPOLICY is FIXED, the form element's size is exactly the size defined in the layout of the form specification file.

The size of the element is computed from the width and height in the form grid and the font used on the front-end side.

The element keeps the size, even if the content is modified. However, if the STRETCH attribute is set to X, Y or BOTH, the form element can still stretch when the parent window size changes.

SIZEPOLICY = DYNAMIC

When SIZEPOLICY is DYNAMIC, the size of the element grows and shrinks according to the width of the content, during the lifetime of the application.

This can be used for COMBOBOX or RADIOGROUP fields, when the size of the widget must fit exactly to its content, which can vary during the program execution.

> **Note:** With SIZEPOLICY=DYNAMIC, some element such as BUTTON, LABEL, IMAGE and RADIOGROUP can shrink and grow all the time, while COMBOBOX elements can only grow.

SIZEPOLICY = INITIAL

When SIZEPOLICY is INITIAL, the size is computed from the initial content, the first time the element appears on the screen. Once the widget displays, its size is frozen. However, if the STRETCH attribute is set to X, Y or BOTH, the form element can still stretch when the parent window size changes.

This is typically used when the size of the element must be fixed, but is not known at design time. For example, when populating a COMBOBOX item list from a database table, the size of the COMBOBOX depends on the size of the labels in the drop-down list).

This size policy mode is also useful when the text of labels is unknown at design time because of internationalization.

With SIZEPOLICY=INITIAL, the behavior differs depending on the form element type.

Table 259: Behavior of SIZEPOLICY=INITIAL, according to the form item type and front-end type

Form item	Behavior with SIZEPOLICY=INITIAL
BUTTON	The size defined in the form is a minimum size. If the text is bigger, the size grows (width and height).
COMBOBOX	The width defined in the form is a minimum width. If one of the items in the value list is bigger, the size grows in order for the combobox to fully display the largest item.
LABEL, CHECKBOX, RADIOGROUP	These form items can shrink or grow. The size defined in the form is ignored. The fields are sized according to the element text.
IMAGE	Image form items can shrink and grow according to the picture displayed. Images can use the STRETCH attribute, so that the widget size is dependent on the parent container, overriding the SIZEPOLICY attribute. If the WIDTH and HEIGHT attributes must be used, the SIZEPOLICY attribute must be set to FIXED.
WEBCOMPONENT	The web component is scaled to the right size, after the first web page is loaded. It stays at that size, except if the STRETCH attribute is used, and the parent container size changes.

Keep in mind that after the first display, the element size will be frozen.

Example

```
COMBOBOX f001 = customer.city,
    ITEMS=((1,"Paris"),(2,"Madrid"), (3,"London")),
    SIZEPOLICY=DYNAMIC;

WEBCOMPONENT wc1 = FORMONLY.chart,
    COMPONENTTYPE="chart",
    SIZEPOLICY=FIXED,
    STRETCH=BOTH;
```

SPACING attribute

The SPACING attribute is a spacing directive to display form elements.

Syntax

```
SPACING = { NORMAL | COMPACT }
```

Usage

This attribute defines the global distance between two neighboring form elements. In NORMAL mode, the front end displays form elements consistent with the desktop spacing, which is, for example, 6 and 10 pixels on Microsoft™ Windows™ platforms.

When using the COMPACT mode, large forms that by default do not fit to the screen can be displayed with less space between elements.

By default, forms are displayed with COMPACT spacing.

Example

```
LAYOUT ( SPACING=COMPACT )
```

SPLITTER attribute

The SPLITTER attribute forces the container to use a splitter widget between each child element.

Syntax

```
SPLITTER
```

Usage

This attribute indicates that the container (typically, a VBOX or HBOX) must have a splitter between each child element held by the container. If a container is defined with a splitter and if the children are stretchable (like TABLE or TEXTEDIT), users can resize the child elements inside the container.

Example

```
VBOX ( SPLITTER )
```

STEP attribute

The STEP attribute specifies how a value is increased or decreased in one step (by a mouse click or key up/down).

Syntax

```
STEP = integer
```

1. *integer* defines a positive integer value to be added (for an increase) or subtracted (for a decrease).

Usage

This attribute is typically used with form items allowing the user to change the current integer value by a mouse click like SLIDER, SPINEDIT.

Example

```
SLIDER s01 = FORMONLY.slider, STEP=10;
```

STRETCH attribute

The STRETCH attribute specifies how a widget must resize when the parent container is resized.

Syntax

```
STRETCH = { NONE | X | Y | BOTH }
```

Usage

This attribute is typically used with form items that can be re-sized like IMAGE. TEXTEDIT, or WEBCOMPONENT fields. By default such form items have a fixed width and height, but in some cases you may want to force the widget to resize vertically, horizontally, or in both directions.

- To allow the widget to resize vertically only, use STRETCH=Y.
- To allow the widget to resize horizontally only, use STRETCH=X.
- To allow the widget to resize vertically and horizontally, use STRETCH=BOTH.

Example

```
IMAGE i01 = FORMONLY.image01, STRETCH=BOTH;
```

STYLE attribute

The STYLE attribute specifies a presentation style for a form element.

Syntax

```
STYLE = "string"
```

1. *string* is a user-defined style.

Usage

This attribute specifies a presentation style to be applied to a form element.

The presentation style can define decoration attributes such as a background color, a font type, and so on.

Note: The string used to define this attribute must be a *style-name* only, it must not contain the *element-type* that is typically used to define the style in a .4st file (as CheckBox.important for example)

Example

```
EDIT c01 = item.comment, STYLE = "important";
```

TABINDEX attribute

The TABINDEX attribute defines the tab order for a form item.

Syntax

```
TABINDEX = integer
```

1. *integer* defines the order of the item in the tab sequence.

Usage

This attribute can be used to define the order in which the form items are selected as the user "tabs" from field.

To take TABINDEX attributes into account in dialogs, the program must defined the form tabbing order with the OPTIONS FIELD ORDER FORM instruction. Alternatively, a dialog can use the FIELD ORDER FORM option as well.

The TABINDEX attribute can also be used to define which field must get the focus when a FOLDER page is selected.

By default, form items get a tab index according to the order in which they appear in the LAYOUT section.

Tip: TABINDEX can be set to zero in order to exclude the form item from the tabbing list. The item can still get the focus with the mouse.

Example

```
EDIT f001 = customer.fname, TABINDEX = 1;
EDIT f002 = customer.lname, TABINDEX = 2;
EDIT f003 = customer.comment,
```

```
      TABINDEX = 0; -- Excluded from tabbing list
```

TAG attribute

The TAG attribute can be used to identify the form item with a specific string.

Syntax

```
TAG = "tag-string"
```

1. *tag-string* is free text.

Usage

This attribute is used to identify form items with a specific string. It can be queried in the program to perform specific processing.

You are free to use this attribute as you need. For example, you can define a numeric identifier for each field in the form in order to show context help, or group fields for specific input verification.

If you need to handle multiple data, you can format the text, for example, by using a pipe separator.

Example

```
EDIT f001 = customer.fname, TAG = "name";
EDIT f002 = customer.lname, TAG = "name|optional";
```

TEXT attribute

The TEXT attribute defines the label associated with a form item.

Syntax

```
TEXT = [%]"string"
```

1. *string* defines the label to be associated with the form item, with the % prefix it is a localized string.

Usage

The TEXT attribute is used to define the label of a form item, for example for a CHECKBOX form field or a BUTTON action view.

Consider using localized strings with the %"*string-id*" syntax, if you plan to internationalize your application.

This attribute is also an action attribute that can be defined in the ACTION DEFAULTS section of a form or directly in an action view (BUTTON), see TEXT action attribute on page 1330 for more details.

Example

```
-- As form action default
ACTION DEFAULTS
  ACTION print (TEXT="Print")
END

-- As a CHECKBOX label
CHECKBOX cb01 = FORMONLY.checkbox01,
                TEXT="OK" ... ;

-- As a BUTTON label
```

```
BUTTON b1: print, TEXT="Print";
```

TITLE attribute
The TITLE attribute defines the title of a form item.

Syntax

```
TITLE = [%]"string"
```

1. *string* defines the title to be associated with the form item, with the % prefix it is a localized string.

Usage

The TITLE attribute is typically used to define the title of a form field that will be defined as a TABLE or TREE column, or form items used in a stacked layout, to define the label associated to the item.

> **Note:** Use of the TITLE attribute should be restricted to form fields that make up the columns of a table/tree container, or form items used in a stacked layout.

Consider using localized strings with the %"*string-id*" syntax, if you plan to internationalize your application.

Example

```
EDIT col4 = FORMONLY.ord_shipdate, TITLE="Ship date";
```

UNSORTABLE attribute
The UNSORTABLE attribute indicates that the element cannot be selected by the user for sorting.

Syntax

```
UNSORTABLE
```

Usage

By default, a TABLE container allows the user to sort the columns by a left-click on the column header.

Use the UNSORTABLE attribute to prevent a sort on a specific column.

Makes sense only for a field that is used for the definition of a column in a TABLE container.

Example

```
EDIT c01 = item.comment, UNSORTABLE;
```

UNSORTABLECOLUMNS attribute
The UNSORTABLECOLUMNS attribute indicates that the columns of the table cannot be selected by the user for sorting.

Syntax

```
UNSORTABLECOLUMNS
```

Usage

When using this attribute in a TABLE definition, the end user will not be allowed to sort rows.

Example

```
TABLE t1 ( UNSORTABLECOLUMNS )
```

UNSIZABLE attribute

The UNSIZABLE attribute indicates that the element cannot be resized by the user.

Syntax

```
UNSIZABLE
```

Usage

By default, a TABLE container allows the user to resize the columns by a drag-click on the column header.

Use this attribute to prevent a resize on a specific column.

Makes sense only for a field that is used for the definition of a column in a TABLE container.

Example

```
EDIT c01 = item.comment, UNSIZABLE;
```

UNSIZABLECOLUMNS attribute

The UNSIZABLECOLUMNS attribute indicates that the columns of the table cannot be resized by the user.

Syntax

```
UNSIZABLECOLUMNS
```

Usage

When using this attribute in a TABLE definition, the end user will not be allowed to resize the columns.

Example

```
TABLE t1 ( UNSIZABLECOLUMNS )
```

UNHIDABLE attribute

The UNHIDABLE attribute indicates that the element cannot be hidden or shown by the user with the context menu.

Syntax

```
UNHIDABLE
```

Usage

By default, a TABLE container allows the user to hide the columns by a right-click on the column header.

Use this attribute to prevent the user from hiding a specific column.

The end user is also denied to show columns that are hidden by default with HIDDEN=USER

Makes sense only for a field that is used for the definition of a column in a TABLE container.

Example

```
EDIT c01 = item.comment, UNHIDABLE;
```

UNHIDABLECOLUMNS attribute

The UNHIDABLECOLUMNS attribute indicates that the columns of the table cannot be hidden or shown by the user with the context menu.

Syntax

```
UNHIDABLECOLUMNS
```

Usage

When using this attribute in a TABLE definition, the end user will not be allowed to hide columns, or show columns that are hidden by default with HIDDEN=USER.

Example

```
TABLE t1 ( UNHIDABLECOLUMNS )
```

UNMOVABLE attribute

The UNMOVABLE attribute prevents the user from moving a defined column of a table.

Syntax

```
UNMOVABLE
```

Usage

By default, a TABLE container allows the user to move the columns by dragging and dropping the column header.

Use this attribute to prevent the user from changing the order of a specific column.

Typically, UNMOVABLE is used on at least two columns to prevent the user from changing the order of the input on these columns.

Example

```
EDIT c01 = item.comment, UNMOVABLE;
```

UNMOVABLECOLUMNS attribute

The UNMOVABLECOLUMNS attribute prevents the user from moving columns of a table.

Syntax

```
UNMOVABLECOLUMNS
```

Usage

When using this attribute in a TABLE definition, the end user will not be allowed to move columns around.

Example

```
TABLE t1 ( UNMOVABLECOLUMNS )
```

UPSHIFT attribute

The UPSHIFT attribute forces character input to uppercase letters.

Syntax

```
UPSHIFT
```

Usage

Assign the UPSHIFT attribute to a character field to automatically convert lowercase letters entered by the user to uppercase letters.

Because uppercase and lowercase letters have different values, storing character strings in one or the other format can simplify sorting and querying a database.

The results of conversions between uppercase and lowercase letters are based on the locale settings.

Example

```
EDIT f001 = FORMONLY.name, UPSHIFT;
```

VALIDATE attribute

The VALIDATE action attribute defines the data validation level for a given action.

Syntax

```
VALIDATE = NO
```

Usage

This attribute is an action attribute that can be specified in form ACTION DEFAULTS, for more details, see VALIDATE action attribute on page 1331.

VALIDATE LIKE attribute

The VALIDATE LIKE attribute applies column attributes defined in the .val database schema files to a field.

Syntax

```
VALIDATE LIKE [table.]column
```

Note:

1. *table* is the optional table name to qualify the column.
2. *column* is the name of the column used to search for validation rules.

Usage

Specifying the VALIDATE LIKE attribute is equivalent to writing in the field definition all the attributes that are assigned to *table*.*column* in the .val database schema file.

Note that .val attributes are taken automatically from the schema file if the field is linked to *table*.*column* in the field name specification. The VALIDATE LIKE attribute is usually specified for FORMONLY fields.

The VALIDATE LIKE attribute is evaluated at compile time, not at runtime. If the database schema file changes, you should recompile all your forms.

Even if all of the fields in the form are FORMONLY, the VALIDATE LIKE attribute requires the form compiler to access the database schema file that contains the description of *table*.

Example

```
EDIT f001 = FORMONLY.fullname, VALIDATE LIKE customer.custname;
```

VALUEMIN attribute

The VALUEMIN attribute defines a lower limit of values displayed in widgets (such as progress bars).

Syntax

```
VALUEMIN = integer
```

1. *integer* is a integer literal.

Usage

This attribute is typically used to define the lower limit in PROGRESSBAR, SPINEDIT and SLIDER fields.

This attribute is not used by the runtime system to validate the field. You must use the INCLUDE attribute to control value boundaries.

Example

```
SLIDER s01 = FORMONLY.slider01,
             VALUEMIN=0,
             VALUEMAX=500;
```

VALUEMAX attribute

The VALUEMAX attribute defines a upper limit of values displayed in widgets (such as progress bars).

Syntax

```
VALUEMAX = integer
```

1. *integer* is an integer literal.

Usage

This attribute is typically usedto define the upper limit in PROGRESSBAR, SPINEDIT and SLIDER fields.

This attribute is not used by the runtime system to validate the field. You must use the INCLUDE attribute to control value boundaries.

Example

```
SLIDER s01 = FORMONLY.slider01,
                         VALUEMIN=0,
                         VALUEMAX=500;
```

VALUECHECKED attribute

The VALUECHECKED attribute defines the value associated with a checkbox item when it is checked.

Syntax

```
VALUECHECKED = value
```

1. *value* is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.

Usage

This attribute is used in conjunction with the VALUEUNCHECKED attribute to define the values corresponding to the states of a CHECKBOX.

This attribute is not used by the runtime system to validate the field, you must use the INCLUDE attribute to control value boundaries.

Example

```
CHECKBOX cb01 = FORMONLY.checkbox01,
                TEXT="OK",
                VALUECHECKED=TRUE,
                VALUEUNCHECKED=FALSE;
```

VALUEUNCHECKED attribute

The VALUEUNCHECKED attribute defines the value associated with a checkbox item when it is not checked.

Syntax

```
VALUEUNCHECKED = value
```

1. *value* is a numeric or string literal, or one of the following keywords: NULL, TRUE, FALSE.

Usage

This attribute is used in conjunction with the VALUECHECKED attribute to define the values corresponding to the states of a CHECKBOX.

This attribute is not used by the runtime system to validate the field, you must use the INCLUDE attribute to control value boundaries.

Example

```
CHECKBOX cb01 = FORMONLY.checkbox01,
                TEXT="OK",
                VALUECHECKED="Y",
                VALUEUNCHECKED="N";
```

VERIFY attribute

The VERIFY attribute requires users to enter data in the field twice to reduce the probability of erroneous data entry.

Syntax

```
VERIFY
```

Usage

This attribute supplies an additional step in data entry to ensure the integrity of your data. After the user enters a value into a VERIFY field and presses the Return or Tab key, the runtime system erases the field and requests reentry of the value. The user must enter exactly the same data each time, character for character: 15000 is not exactly the same as 15000.00.

The VERIFY attribute takes effect in INPUT or INPUT ARRAY instructions only, it has no effect on CONSTRUCT statements.

VERSION attribute
The VERSION attribute is used to specify a user version string for an element.

Syntax

```
VERSION = { "string" | TIMESTAMP }
```

1. *string* is a user-defined version string.

Usage

This attribute specifies a version string to distinguish different versions of a form element. You can specify an explicit version string or use the TIMESTAMP keyword to force the form compiler to write a timestamp string into the 42f file.

Typical usage is to specify a version of the form to indicate if the form content has changed. This attribute is used by the front-end to distinguish different form versions and to avoid reloading window/form settings into a new version of a form.

You should use the TIMESTAMP only during development.

Example

```
LAYOUT ( TEXT="Orders", VERSION = "1.23" )
```

WANTFIXEDPAGESIZE attribute
The WANTFIXEDPAGESIZE attribute controls the vertical resizing of a list element.

Syntax

```
WANTFIXEDPAGESIZE [ = NO ]
```

Usage

The WANTFIXEDPAGESIZE attribute can be used for TABLE and SCROLLGRID containers to control the vertical resizing of the list element.

By default, a TABLE container is resizable (vertically and horizontally). To freeze the height of the table to the number of lines defined by the form file, use the attribute WANTFIXEDPAGESIZE.

By default, a SCROLLGRID container is not resizable in height. The number of visible scrollgrid rows is defined by the form file. To allow the scrollgrid to stretch vertically, use the attribute WANTFIXEDPAGESIZE=NO.

WANTNORETURNS attribute
The `WANTNORETURNS` attribute forces a text field to reject newline characters when the user presses the Return key.

Syntax

```
WANTNORETURNS
```

Usage

By default, text fields like `TEXTEDIT` insert a newline (ASCII 10) character in the text when the user presses the Return key. As the Return key is typically used to fire the *accept* action to validate the dialog, you can force the field to reject Return keys with this attribute.

The user can still enter newline characters with Shift-Return or Ctrl-Return, if these keys are not bound to actions.

WANTTABS attribute
The `WANTTABS` attribute forces a text field to insert Tab characters in the text when the user presses the Tab key.

Syntax

```
WANTTABS
```

Usage

By default, text fields like `TEXTEDIT` do not insert a Tab character in the text when the user presses the Tab key, since the Tab key is used to move to the next field. You can force the field to consume Tab keys with this attribute.

The user can still jump out of the field with Shift-Tab, if this key is not bound to an action.

WIDGET attribute
The `WIDGET` attribute specifies the type of graphical widget to be used for the field.

Syntax

```
WIDGET = "identifier"
```

1. *identifier* defines the type of widget, it can be one of the keywords listed in Table 260: Supported widgets on page 996.

Usage

The `WIDGET` attribute defines the type of widget to be used for the form field.

This attribute is used with `CONFIG`, `CLASS` and `INCLUDE` attributes, to parameter the field widget.

> **Important:** This attribute is deprecated, use new form item types instead.

- Instead of `WIDGET="IMAGE"`, you should now use a IMAGE form item.
- Instead of `WIDGET="CANVAS"`, you should now use a CANVAS form item.
- Instead of `WIDGET="CHECK"`, you should now use a CHECKBOX form item.
- Instead of `WIDGET="COMBO"`, you should now use a COMBOBOX form item.
- Instead of `WIDGET="BMP"`, you should now use a BUTTON form item.
- Instead of `WIDGET="FIELD_BMP"`, you should now use a BUTTONEDIT form item.
- Instead of `WIDGET="RADIO"`, you should now use a RADIOGROUP form item.

The *identifier* widget type is case sensitive, only uppercase letters are recognized.

When you use the `WIDGET` attribute, the form cannot be properly displayed on character based terminals, it should only be displayed on graphical front ends.

Table 260: Supported widgets

Symbol	Effect	Other attributes
`Canvas`	The field is used as a drawing area. Field must be declared as `FORMONLY` field.	None.
`BUTTON`	The field is presented as a button widget with a label.	`CONFIG`: The unique parameter defines the key to be sent when the user clicks on the button. Button text is defined in configuration files or from the program with a `DISPLAY TO` instruction. For example: `CONFIG = "Control-z"`
`BMP`	The field is presented as a button with an image.	`CONFIG`: First parameter defines the image file to be displayed, second parameter defines the key to be sent when the user clicks on the button. For example: `CONFIG = "smiley.bmp F11"` **Important:** Image files are not centralized on the machine where the program is executed; image files must be present on the Workstation. See front end specific documentation for more details.

`CHECK`	The field is presented as a checkbox widget. It can be used with the `CLASS` attribute to change the behavior of the widget.	`CONFIG`: First and second parameters define the values corresponding respectively to the state "Checked" / "Unchecked" of the check box, while the third parameter defines the label of the checkbox.
		For example:
		`CONFIG = "Y N Confirmation"`
		If the text part must include spaces, add {} curly braces around the text:
		`CONFIG = "Y N {Order validated}"`
		If the `CLASS` attribute is used with the `"KEY"` value, the first and second parameters defines the keys to be sent respectively when the checkbox is "Checked" / "Unchecked", and the third parameter defines the label of the checkbox as with normal checkbox usage.
		For example (line breaks added for document readability):

Symbol	Effect	Other attributes
		`CLASS = "KEY",` `CONFIG = "F11 F12 Confirmation"`

`COMBO`	The field is presented as a combobox widget. It can be used with the `CLASS` attribute to change the behavior of the widget.	`INCLUDE`: This attribute defines the list of acceptable values that will be displayed in the combobox list.
		For example (line breaks added for document readability):
		`INCLUDE =`
		`("Paris", "London", "Madrid")`
		Important: The `INCLUDE` attribute cannot hold value range definitions, because all items must be explicitly listed to be added to the combobox list.
		The following example is not supported:
		`INCLUDE = (1 TO 10)`

`FIELD_BMP`	The field is presented as a normal editbox, plus a button on the right.	`CONFIG`: The first parameter defines the image file to be displayed in the button; the second parameter defines the key to be sent when the user clicks on the button. For example: `CONFIG = "combo.bmp Control-z"`
`LABEL`	The field is presented as a simple label, a read-only text.	None.

`RADIO`	The field is presented as a radiogroup widget.	`CONFIG`: Parameter pairs define respectively the value and the label corresponding to one radio button.
		For example (line breaks added for document readability):
		`CONFIG =`
		`"AA First BB Second CC Third"`
		If the radio texts must include spaces, add {} curly braces around the texts (line breaks added for document readability):
		`CONFIG = "AA {First option}`
		`BB {Second option} CC {Third option}"`
		If the `CLASS` attribute is used with the value `"KEY"`, the first element of each pairs represents the key to be sent when the user selects a radio button.
		For example (line breaks added for document readability):
		`CLASS = "KEY",`
		`CONFIG = "F11 First F12 Second`

Symbol	Effect	Other attributes
		F13 Third"

Controlling old style widgets activation

The following list of widgets can be enabled or disabled from programs with a DISPLAY TO instruction:

- Text buttons (WIDGET="BUTTON")
- Image buttons (WIDGET="BMP")
- Checkboxes of class "KEY" (WIDGET="CHECK", CLASS="KEY")
- Radio buttons of class "KEY" (WIDGET="RADIO", CLASS="KEY")

If you display an exclamation mark in such fields, the button is enabled, but if you display a star (*), it is disabled:

```
DISPLAY "*" TO button1 # disables the button
DISPLAY "!" TO button1 # enables the button
```

Changing the text of WIDGET="BMP" fields

The text of button fields (WIDGET="BUTTON") can be changed from programs with the DISPLAY TO instruction:

```
DISPLAY "Click me" TO button1
 # Sets text and enables the button
```

Changing the image of WIDGET="BMP" fields:

The image of button fields (WIDGET="BMP") can be changed from programs with the DISPLAY TO instruction:

```
DISPLAY "smiley.bmp" TO button1
 # Sets image and enables the button
```

Image files are not centralized on the machine where the program is executed; image files must be present on the Workstation. See front end specific documentation for more details.

Changing the text of WIDGET="LABEL" fields:

The text of label fields (WIDGET="LABEL") can be changed from programs with the DISPLAY TO instruction:

```
DISPLAY "Firstname" TO l_firstname
 # Sets text of the label field
```

Using WIDGET="Canvas" fields:

The fields declared with the WIDGET="Canvas" attribute can be used by the program as drawing areas. Canvas fields must be defined in the LAYOUT section. A set of drawing functions are provided to fill canvas fields with graphical elements.

WIDTH attribute

The WIDTH attribute defines an explicit width of a form element.

Syntax

```
WIDTH = integer [CHARACTERS|COLUMNS|POINTS|PIXELS]
```

1. *integer* defines the width of the element.

Usage

By default, the width of an element is defined by the size of the form item tag in a grid-based layout, or by the type of the form item in a stack-based layout. Use the WIDTH attribute to define a specific width for a form item.

> **Note:** As a general rule, consider not specifying a unit, to default to relative characters/lines/columns, instead of specifying exact pixels or points. This is especially important for mobile devices, where the screen resolution can significantly vary according to the smartphone or tablet model.

Grid-based layout

For sizable items like IMAGE, the default width is defined by the number of horizontal characters used in the form item tag. Overwrite this default by specifying the WIDTH attribute.

```
IMAGE img1: image1, WIDTH = 20, HEIGHT = 12;
```

For TABLE/TREE containers, the default width is defined by the columns used in the table layout. Overwrite the default by specifying the WIDTH = *x* COLUMNS attribute. This will give a small initial width for tables with a large number of columns.

```
TABLE t1: table1, WIDTH = 5 COLUMNS;
```

In a grid-based layout, if you don't specify a size unit, it defaults to CHARACTERS, which defines a width based on the characters size in the current font.

Stack-based layout

In a stack-based layout, the WIDTH attribute cannot be used: The width of form element are automatically computed.

WINDOWSTYLE attribute

The WINDOWSTYLE attribute defines the style to be used by the parent window of a form.

Syntax

```
WINDOWSTYLE = "string"
```

1. *string* is a user-defined style name.

Usage

The WINDOWSTYLE attribute can be used to specify the style of the parent window that will hold the form. This attribute is specific to the LAYOUT element. Do not confuse this attribute with the STYLE attribute, which is used to specify the decoration style of the form elements.

When a form is loaded by the OPEN WINDOW or DISPLAY FORM instruction, the runtime system automatically assigns the WINDOWSTYLE to the STYLE attribute of the parent window element.

Example

```
LAYOUT ( STYLE="BigFont", WINDOWSTYLE="dialog" )
```

WORDWRAP Attribute

The WORDWRAP attribute enables a multiple-line editor in TUI mode.

Syntax

```
WORDWRAP [ { COMPRESS | NONCOMPRESS } ]
```

Usage

This attribute is provided for backward compatibility with character-based forms, to support word wrapping in multi-line text input.

In GUI mode, you should use a TEXTEDIT form item instead. When used, the WORDWRAP attribute is ignored, because text input and display is managed by the text editor widget. The text data is not automatically modified by the editor by adding blanks to put words on the next line.

In TUI mode, the WORDWRAP attribute has following effects:

- During input and display, the runtime system treats all segments that have that field tag as segments of a single field.
- The multi-line editor can *wrap* long character strings to the next line of a multiple-segment field for data entry, data editing, and data display.
- The COMPRESS option prevents blanks produced by the editor from being included in the program variable. COMPRESS is applied by default and can cause truncation to occur if the sum of intentional characters exceeds the field or column size. Because of editing blanks in the WORDWRAP field, the stored value might not correspond exactly to its multiple-line display.
- Specifying NONCOMPRESS after the WORDWRAP keyword causes any editor blanks to be saved when the string value is saved in a database column, in a variable, or in a file.

Using WORDWRAP fields with character-based terminals results in quite different behavior than with graphical front ends. With character-based terminals, the text input and display is modified by the multi-line editor. This editor can automatically modify the text data by adding blanks to put words to the next line, in order to make the text fit into the form field. In GUI mode, the text input and display is managed by a multi-line edit control.

The maximum number of bytes a user can enter is the width of the form-field multiplied by the height of the form-field. Blank characters may be intentional blanks or fill blanks. Intentional blanks are initially stored in the target variable where entered by the user. Fill blanks are inserted at the end of a line by the editor when a newline or a word-alignment forces a line-break. It is not possible to set the cursor at a fill blank. Intentional blanks are always displayed (even on the beginning of a line; the word-wrapping method used in reports with PRINT WORDWRAP works differently).

When entering characters with Japanese locales, special characters are prohibited from being the first or the last character on a line. If the last character is prohibited from ending a line, this character is wrapped down to the next line. If the first character is prohibited from beginning a line, the preceding character will also wrap down to the next line. This method is called kinsoku. The test for prohibited characters will be done only once for the first and the last character on each line.

Word-wrapping is disabled on the last row of a WORDWRAP field. The last word on the last row may by truncated. The WORDWRAP COMPRESS attribute instructs the editor to remove fill blanks before assigning the field-buffer to the target variable. The WORDWRAP NONCOMPRESS attribute instructs the editor to store fill blanks to the target variable. The WORDWRAP and WORDWRAP NONCOMPRESS attributes are equivalent.

The WORDWRAP attribute is not used by the CONSTRUCT instruction.

Examples

Example 1: Grid-based layout form

```
LAYOUT ( TEXT = "Customer orders" )
  VBOX
    GROUP group1 ( TEXT = "Customer" )
      GRID
      {
        <GROUP Name                                          >
         [f001                                              ]
        <                                                    >
        <GROUP Identifiers       ><GROUP Contact             >
         FCode: [f002           ]  Phone: [f004             ]
         LNumb: [f003           ]  EMail: [f005             ]
        <                         ><                         >
      }
      END
    END
    TABLE
    {
      OrdNo  Date       Ship date   Weight
     [c01   |c02       |c03        |c04        ]
     [c01   |c02       |c03        |c04        ]
     [c01   |c02       |c03        |c04        ]
     [c01   |c02       |c03        |c04        ]
    }
    END
    FOLDER
      PAGE pg1 ( TEXT = "Address" )
        GRID
        {
           Address:  [f011                              ]
           State:    [f012           ]
           Zip Code: [f013      ]
        }
        END
      END
      PAGE pg2 ( TEXT = "Comments" )
        GRID
        {
          [f021                                         ]
          [                                             ]
          [                                             ]
          [                                             ]
        }
        END
      END
    END
  END
END
```

Example 2: Stack-based layout form

```
SCHEMA stores

ACTION DEFAULTS
  ACTION import(TEXT=%"action.import")
END

TABLES
  customer
END
```

```
LAYOUT(TEXT=%"title.customer")
STACK
GROUP
  LABEL, TEXT=%"label.new_customer";
END --GROUP
GROUP
  EDIT customer.customer_num, NOENTRY, TITLE=%"label.number";
  EDIT customer.fname, TITLE=%"label.first_name";
  EDIT customer.lname, TITLE=%"label.last_name";
  EDIT customer.company, TITLE=%"label.company";
END --GROUP
GROUP(TEXT = "group.address")
  EDIT customer.address1, TITLE=%"label.address1";
  EDIT customer.address2, TITLE=%"label.address2";
  EDIT customer.city, TITLE=%"label.city";
  BUTTONEDIT customer.state, TITLE=%"label.state", UPSHIFT,
             NOTEDITABLE, ACTION = zoom;
  EDIT customer.zipcode, TITLE=%"label.zipcode";
END --GROUP
GROUP phone_edit
  EDIT customer.phone, TITLE=%"label.telephone",
       KEYBOARDHINT=PHONE;
END --GROUP
GROUP phone_dial
  BUTTON dial, TEXT=%"button.dial";
END --GROUP
END --STACK
END --LAYOUT
```

Form rendering

The section explains the layout rules to render forms on graphical front-ends.

- Form rendering basics on page 1002
- Grid-based layout on page 1004
- Stack-based layout on page 1017

Form rendering basics

Get the essentials about form rendering.

In the graphical mode (GUI mode), forms are not displayed in a fixed text-mode screen. Application windows can display complex layouts and are resizable by the end user, if the platform allows window resizing (mobile devices versus desktop platforms).

When developing with command line tools, forms are designed with `.per` form specification files, which are text files. In order to display text-based forms in graphical mode, the text-based form definitions must be converted to graphical forms, which implies specific layout rules. These rules are explained in this section.

We distinguish two type of form rendering techniques:

- Grid-based rendering, based on a grid of cells, to place and size form elements.
- Stack-based rendering, where all form elements are place over each other vertically.

Character set usage

The character set used to edit and compile `.per` form specification files is defined by the current locale.

Form elements (typically, labels) can be written with non-ASCII characters of the current codeset.

In a grid-based layout, the form element positions and sizes are determined by counting the width of characters, rather than the number of bytes identifying the characters in the current codeset. This rule can be ignored when using a single-byte character set such as ISO-8859-1 or CP-1252, where each character

has width of 1 and codepoint of 1 byte. This rule is important when using a multibyte character set such as BIG5 or UTF-8.

For example, in the UTF-8 multibyte codeset, a Chinese ideogram is encoded with three bytes, while the visual width of the character is twice the size of a Latin character. In the next example, the labels with three Chinese characters have the same width as the labels using six Latin characters. As a result, all the labels will get the same size (6 cells), and all fields will be aligned properly in a proportional font display:

```
LAYOUT
GRID
{
### [f001  ] abcdef [f002  ]
abcdef [f003  ] ### [f004  ]
}
END
END
```

In a stack-based container, the position of form elements is logical, the current locale does not impact on the form item positions as in a grid-based container:

```
LAYOUT
 STACK
   GROUP
      EDIT customer.cust_num, TITLE="###";
      EDIT customer.cust_name;
      EDIT customer.cust_address;
   END
   ...
 END
END
```

For maximum portability, it is recommended to write all form specification files in ASCII (7 bit), and use localized strings to internationalize your forms.

Adapting to viewport changes

Application forms and functions can be adapted according to the front-end viewport size or mobile device orientation.

Detecting viewport size / orientation changes

When the mobile device orientation changes, or when the current window is resized on desktop/web front-ends, the `windowresized` specific predefined action will be sent, if an `ON ACTION` handler is defined by the current dialog for this action.

> **Note:** The `windowresized` action should only be used to hide/show items on the current form using the standard user interface API (`ui.Form.setElementHidden()`) and not reload forms on the fly.

This predefined action can be used to detect viewport geomatry changes and adapt the application form to the new size:

```
  ON ACTION windowresized
     -- Code to adapt to the new viewport size
```

> **Note:** In dialogs allowing field input (`INPUT` / `INPUT ARRAY` or `CONSTRUCT`), take care of the current field input: The `windowresized` action can force the field validation. Therefore, it is not recommended to use this special action in these dialogs. The action can be safely used in `DISPLAY ARRAY` and `MENU` dialogs.

To control the action view rendering defaults and current field validation behavior when the `windowresized` action is used and fired, consider setting action defaults attribute for this action in your .4ad file as follows:

```
<ActionDefaultList>
  ...
  <ActionDefault name="windowresized" validate="no" defaultView="no"
 contextMenu="no"/>
  ...
</ActionDefaultList>
```

Querying the geometry of the viewport

Use the feInfo/windowSize front call to query the actual size of the front-end view-port (GDC current window, GBC webview, or mobile screen size):

```
DEFINE size STRING
CALL ui.Interface.frontCall("standard","feInfo",["windowSize"],[size])
IF size == "1200x1824" THEN
   ...
END IF
```

Grid-based layout

A form file can define a grid-based layout within a tree of layout items.

In a `.per` form specification file, the `LAYOUT` section defines a tree of layout containers, which hold form items such as labels and form fields.

The `GRID` container can be used to define a grid of cells that hold form items: In the layout tree, the `GRID` container acts as a leaf node, which holds the visible widgets (fields, buttons, and so on).

> **Note:** `SCROLLGRID` and `GROUP` containers defined by layout tags inside a grid without the `GRIDCHILDRENINPARENT` attribute, are similar to `GRID` containers in regards to the layout rules describe in this section.

The `.per` form specification file defines a form layout based on a character grid, each character defines a cell of the grid:

```
GRID
{
First Name [fname  ]
Last Name  [lname  ]
}
END
```

The `.per` file layout specification can be shown in a character grid.

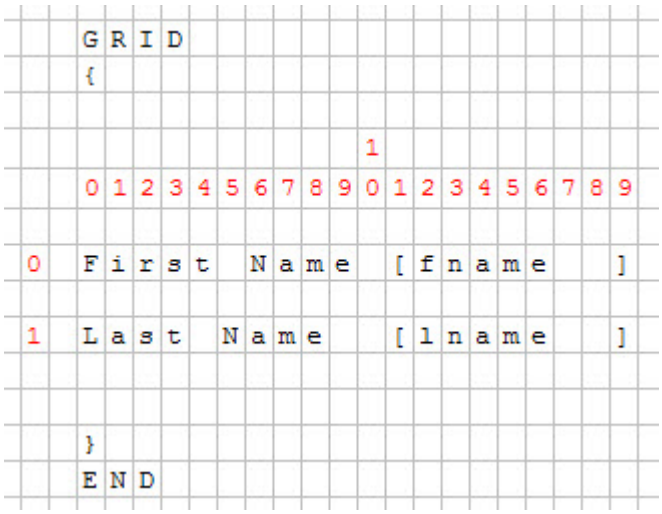


Figure 48: Character grid of a form layout

With a fixed-font based front end (such as a dumb terminal), the forms appear within a screen where each cell is identified by x and y coordinates, as in the SCREEN section of the form specification file. There is no particular layout issue, as all characters can be displayed at the same (relative) position as in the source form file.

With the graphical front-end, text-based forms must be displayed in a graphical window using fonts with a proportional size. In a proportional font, the field label "Key" has a different graphical length than the label "Num", despite having the same number of characters.

In the compiled version of the form specification file, all form items get coordinates in a virtual grid (defined by `posX` and `posY` attributes), and the number of cells the item occupies in the grid (in the `gridWidth` and `gridHeight` attributes):

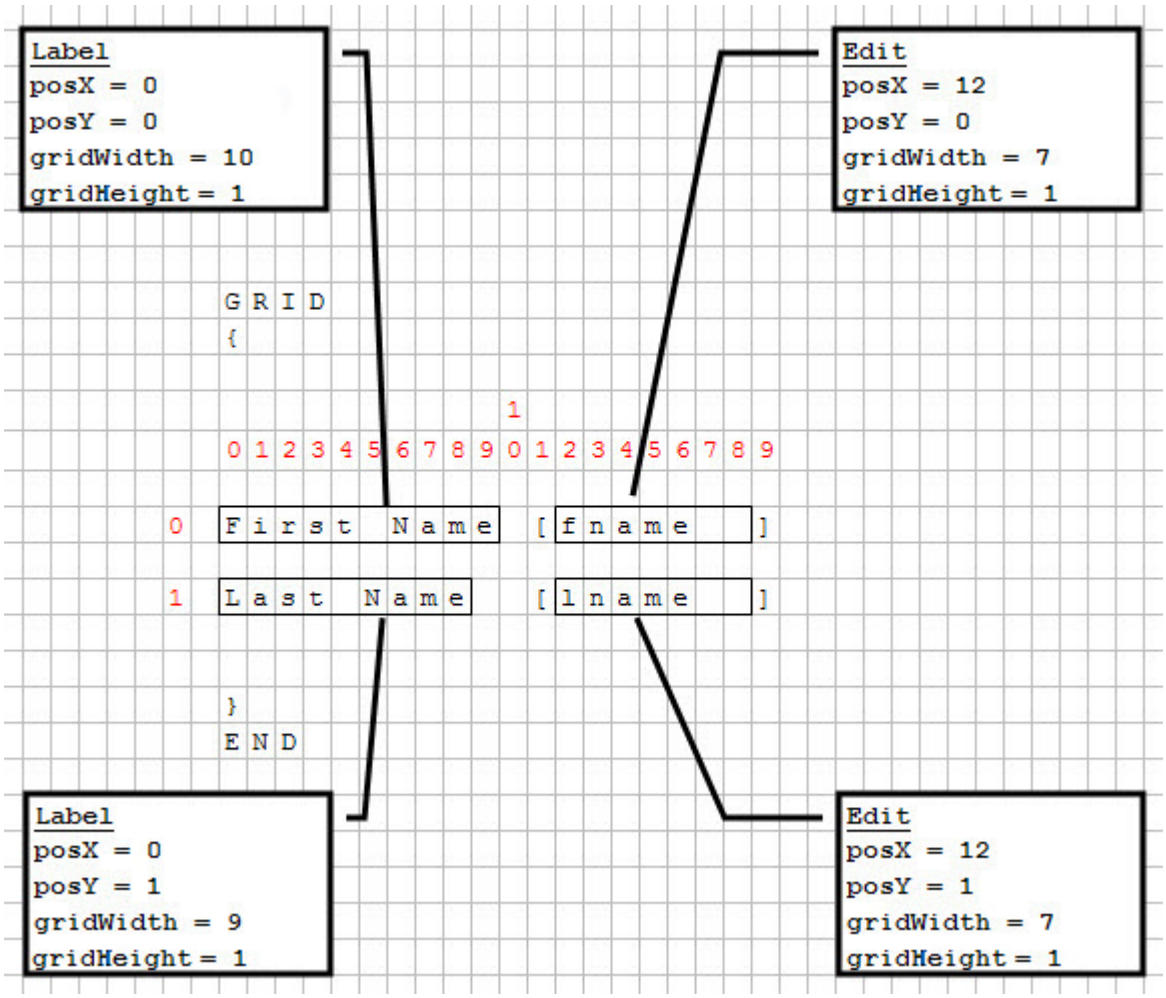


Figure 49: Grid positioning

The "First Name" and "Last Name" texts are identified as whole labels, even if the words "First" and "Name" (or "Last" and "Name") are not joined in the form definition, because the form compiler considers a single blank as a word separator within labels.

Packed and unpacked grids
When resizing a window, the content will either grow with the window or be packed in the top left position.

If elements in the window can grow, they will follow the window container and resize accordingly. Some elements can grow vertically, some can grow horizontally, and some can grow in both directions. The way resizable form items can grow is controlled by the `STRETCH` attribute. The window content is packed horizontally, vertically or in both directions, if none of the elements can grow in that direction.

The following form item types can grow horizontally:

- `TABLE` / `TREE` items
- `IMAGE` items (with `STRETCH=BOTH` or `STRETCH=X`)
- `TEXTEDIT` items (with `STRETCH=BOTH` or `STRETCH=X`)

The following form item types can grow vertically:

- `TABLE` / `TREE` items (without `WANTFIXEDPAGESIZE` attribute)
- `IMAGE` items (with `STRETCH=BOTH` or `STRETCH=Y`)
- `TEXTEDIT` items (with `STRETCH=BOTH` or `STRETCH=Y`)

In general, a `GRID` container can grow if any object inside the `GRID` can grow. The exception to this rule: If there is a single `GROUP` container (defined without the `GRIDCHILDRENINPARENT` attribute) inside a `GRID` and nothing else, the grid can grow even if the objects inside the grid cannot grow.

This exception allows better rendering of a grouped grid.

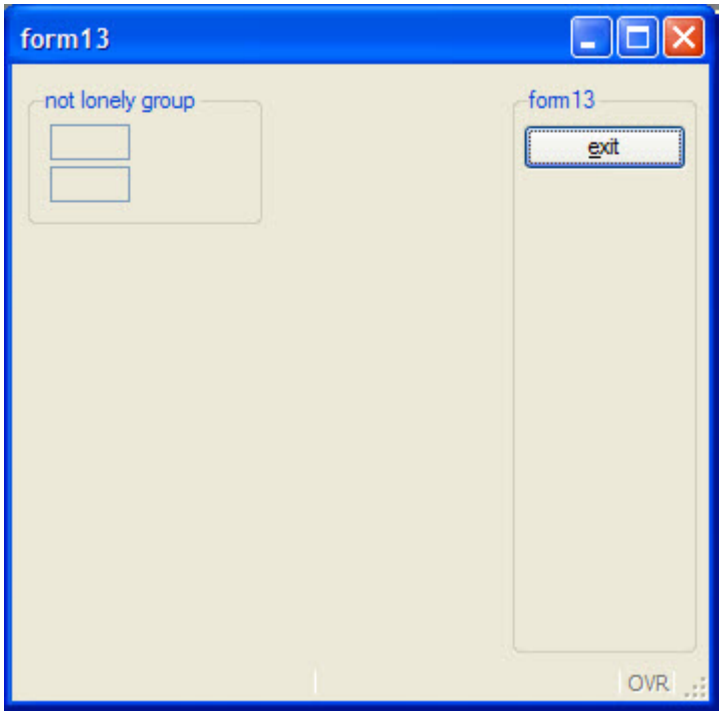


Figure 50: Packed grid

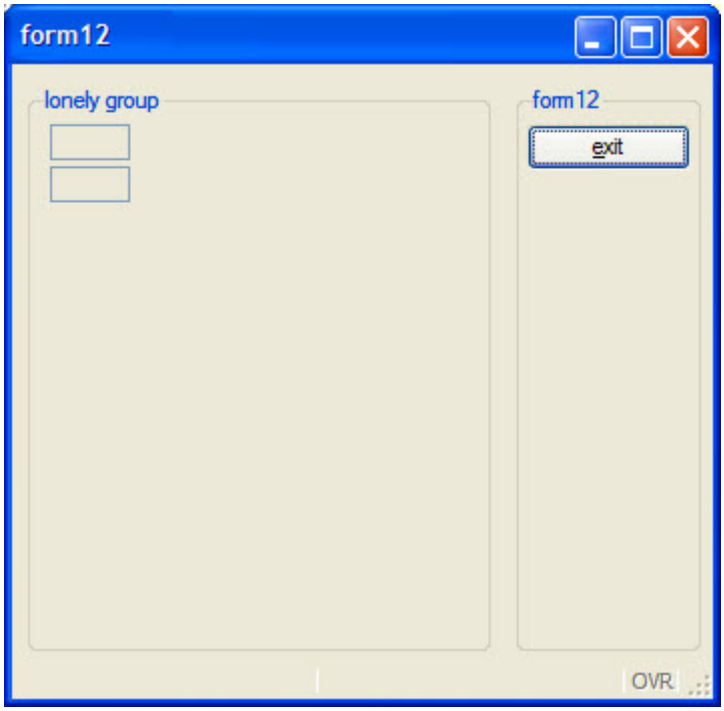


Figure 51: Unpacked grid

Automatic HBoxes and VBoxes
Horizontal and vertical boxes are added automatically when stretchable elements are used.

When using layout tags in a `GRID` container, the `fglform` compiler will automatically add hbox or vbox containers with splitters in the following conditions:

- An hbox is created when two or more stretchable elements are stacked side by side and touch each other (no space between).
- A vbox is created when two or more stretchable elements are stacked vertically and touch each other (no space between).

No hbox or vbox will be created if the elements are in a `SCROLLGRID` container.

This example defines two tables stacked vertically, generating a vbox with splitter. The ending tags for the tables are omitted.

```
<T table1         >
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
[colA  |colB      ]
<T table2         >
[colC  |colD      ]
[colC  |colD      ]
```

This example defines a layout with two stretchable `TEXTEDIT` fields placed side by side, which would generate an automatic hbox with splitter. To make both widgets touch, you need to use a pipe delimiter in between the two widgets.

```
[textedit1         |textedit2                ]
[                  |                         ]
[                  |                         ]
[                  |                         ]
```

Widget position and size in grid
Form items render as widgets in the window, at a given position and with a given size.

To render form items, grid-based rendering follows the layout rules described below:

1. The position of the widgets in the virtual grid is defined by the `posX` and `posY` AUI tree attributes.
2. The number of virtual grid cells occupied by a widget is defined by the `gridWidth` and `gridHeight` AUI tree attributes.
3. The real size (i.e. pixels) of a widget is defined by the `width` and `height` AUI tree attributes.
4. Empty lines and empty columns in the form layout definition take a size of 0 pixels, but this can be configured with `emptyColWidth` and `emptyRowHeight` style attributes (see below).
5. The size of a cell in the virtual grid depends on the real size of the widgets inside the grid.
6. A widget's minimum size is computed via its real size and the `SAMPLE` attribute.
7. The preferred size of the widget is computed according to the `SIZEPOLICY` attribute.
8. The final widget size is computed according to the minimum and preferred size, to fill the cells in the grid.
9. A small spacing is applied in non-empty cells.

The next screen-shot shows 2 labels and 2 fields placed in a grid.

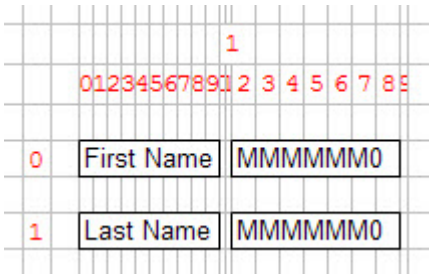


Figure 52: Two labels and two fields placed in a grid: Grid view

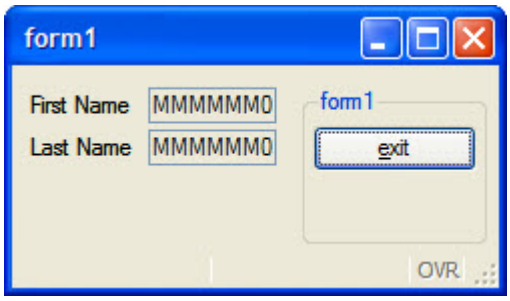


Figure 53: Two labels and two fields placed in a grid: Form view

By default, empty grid rows and empty grid columns get not size when rendering on the front-end. For example, in the above grid sample, the grid columns #10 and #11 are empty.

Form item dependencies in grids
Form items interact with each other in terms of width, according to the front-end widget size.

This example illustrates how form items are dependent of each other inside the grid.

```
GRID
{
  [a      ]
  [b      ]
}
END
```

This `.per` implies that form items a and b start at the same position and have the same size, whatever a and b are.

This rule could lead to very different results, especially when a large widget is assigned into a small number of cells.

Example:

```
LAYOUT
GRID
{
[a|b   ][f     ]
[c|d]   [e     ]
}
END
END

ATTRIBUTES
CHECKBOX a = formonly.a, TEXT="A Checkbox";
EDIT b = formonly.b;
EDIT c = formonly.c;
CHECKBOX d = formonly.d, TEXT="Another Checkbox";
EDIT e = formonly.e;
```

```
EDIT f = formonly.f;
END
```

The grid will be computed regarding characters cells in the form definition:

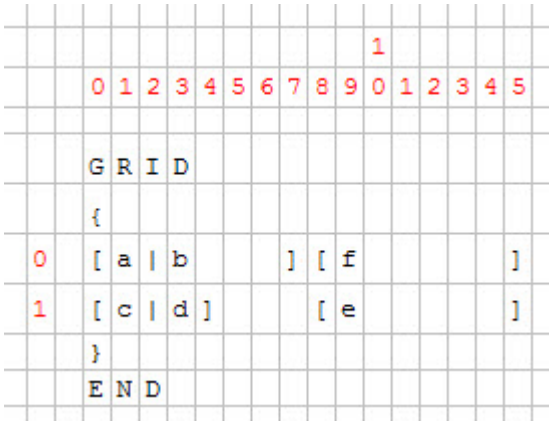


Figure 54: Grid layout

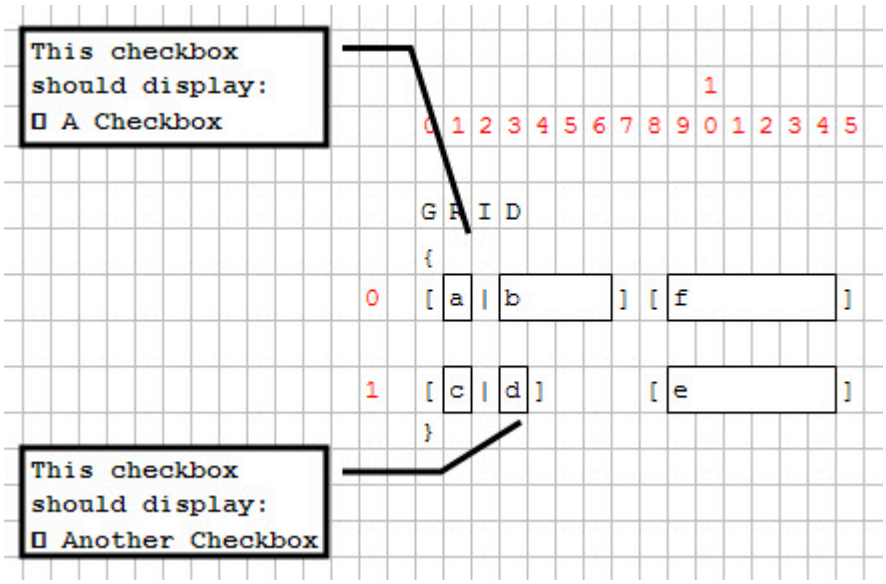


Figure 55: Grid layout with checkboxes

Then the minimum size of each widget and the layout is computed.

Cells (0,1) and (1,3) contain a checkbox; these checkboxes will enlarge columns 1 and 3.

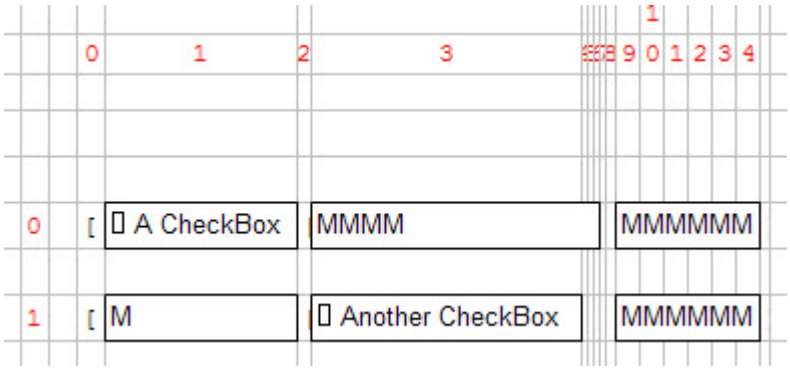


Figure 56: Enlarged columns

Because the `EDIT` field "c" is defined to have the same width as checkbox "a", it will be much larger as expected.

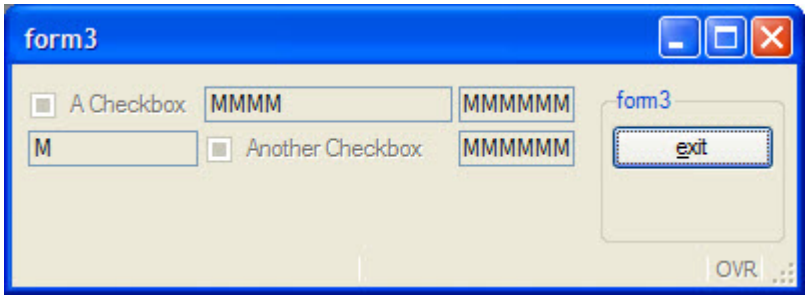


Figure 57: Resulting form

To avoid this visual result, you must assign a realistic number of grid cells for each form item.

```
GRID
{
[a         |b  ][f     ]
[c|d           ][e     ]
}
```

Even if the grid area is wider in the source form file, the real graphical result will be smaller.

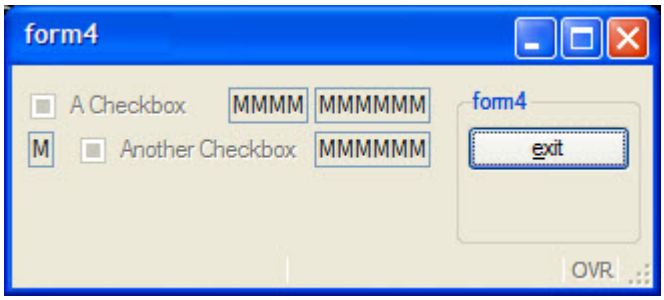


Figure 58: Resulting form

Complex grid layout example
Describes how form item align in grid-based front-ends with an example.

These diagrams show the virtual grid of a complex form, with several field item tags.

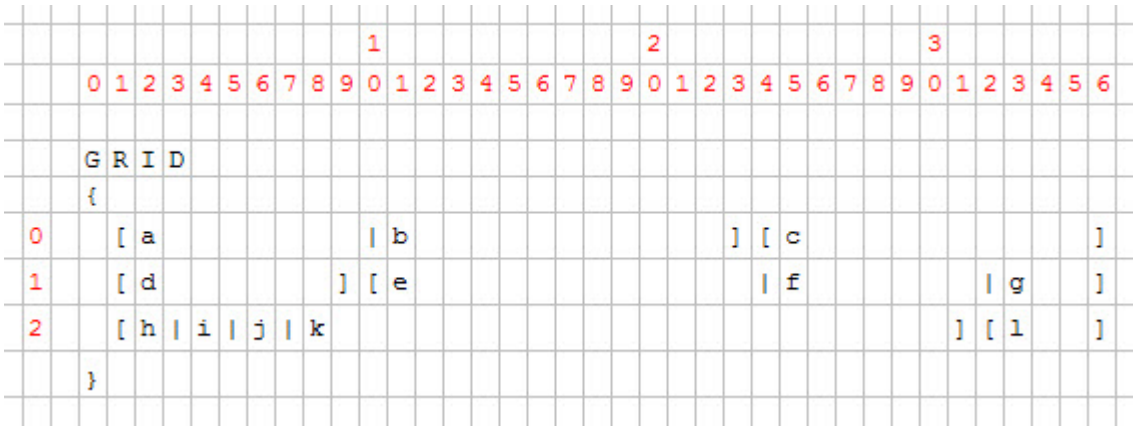


Figure 59: Grid containing several fields

For each form field, the position and the number of cells is computed by the form compiler.

At runtime, the front-end creates the widgets and sets them on the virtual grid.

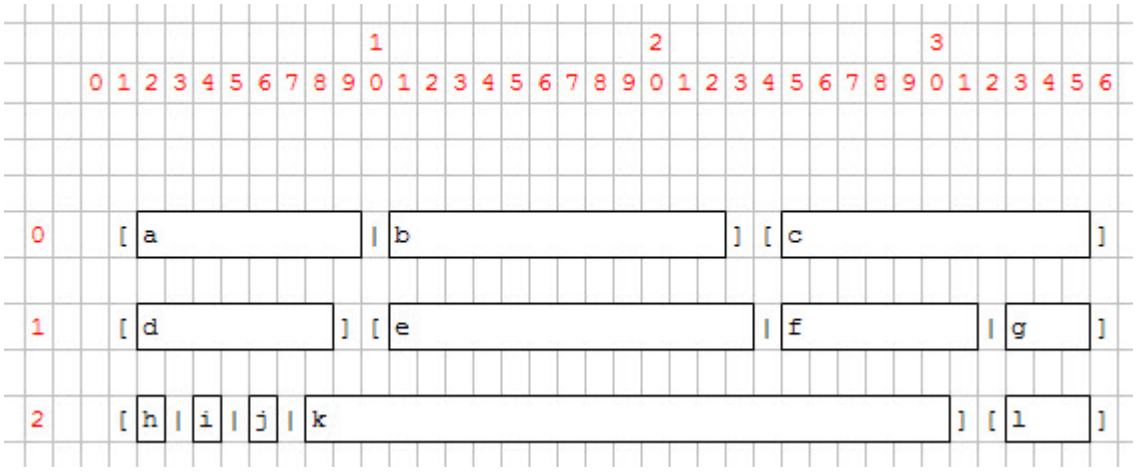


Figure 60: Widgets set on grid

Once widgets are on the grid, their minimum size is computed according to their size, to `SIZEPOLICY` and the `SAMPLE` attributes.

Then the sizes of the grid cells adapt to the size of the widgets.

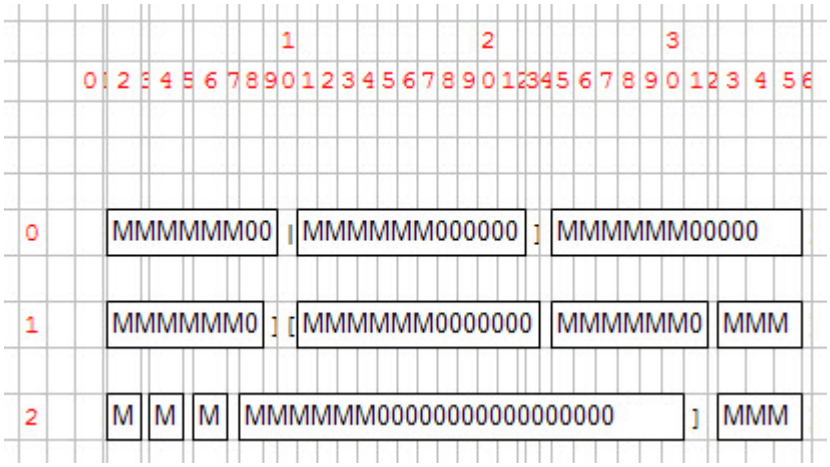


Figure 61: Widget size computations

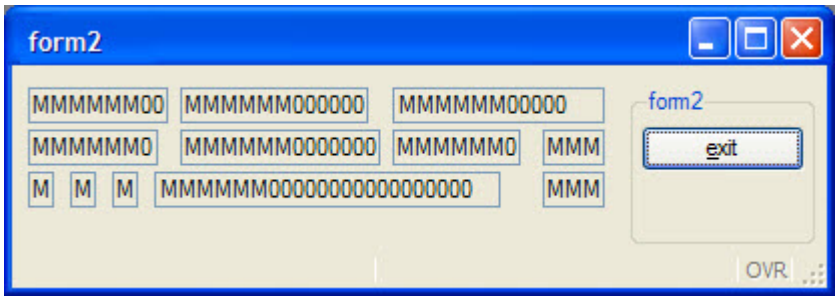


Figure 62: Widgets in rendered form

In this screen shot, the fields k and c are much bigger than expected:

- Field g and l make columns 33, 34 and 35 bigger than the others,
- Field f extends columns 25 to 31.
- As field c has to fill columns 25 to 35, its size grows; the same for field k.

Some fields are proportionally bigger than others because some parameters are variable, while others are fixed.

The width of the widget is the sum of border width, plus the content width (depending on SIZEPOLICY and the SAMPLE attributes).

Since the default SAMPLE is MMMMMM000..., the graphical width of a field is not linearly proportional to the width defined in the form file. For example, a field of 1 will be as wide as 2 borders + 1 'M'. A field of 10 will be as wide as 2 borders + 6 'M' + 4 '0'. This means that a field of 1 is far from being 10 times smaller than a field of 10.

Using hbox tags to align form items

The hbox tag concept has been introduced to bypass the limitations of the character-based grid in forms.

An hbox tag defines a widget container that will gather the child widgets horizontally, like the horizontal box container. All widgets inside this container are no longer dependent on the parent grid.

- Defining hbox tags in grids on page 1013
- Spacer items in hbox tags on page 1015
- Widget size within hbox tags on page 1016

Defining hbox tags in grids

An hbox tag is defined by using a : colon in an item tag delimited by square braces.

This example creates a hbox container containing widgets a, b and c. These widgets won't be aligned in the grid.

```
GRID
{
[a:b:c   ]
[d|e|f   ]
}
END
```

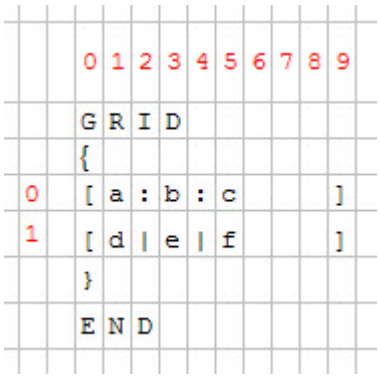


Figure 63: Using an hbox tag

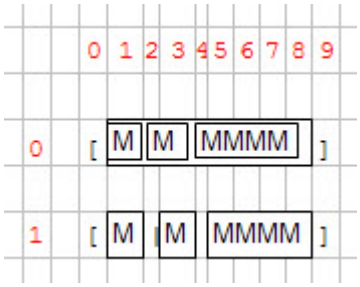


Figure 64: HBox tag rendering

Hbox tags are useful when the form contains large widgets in a small number of cells in one row, and don't want to have dependencies.

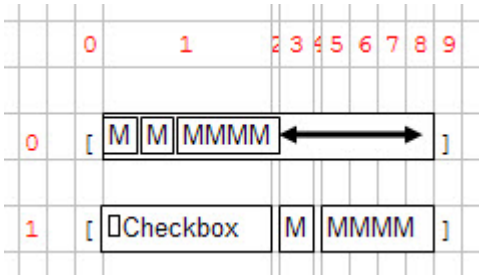


Figure 65: Using an HBox tag

We can modify the Form item dependencies in grids on page 1009 example, using hbox tags:

```
GRID
{
[a:b   ][f     ]
[c:d   ][e     ]
}
END
```

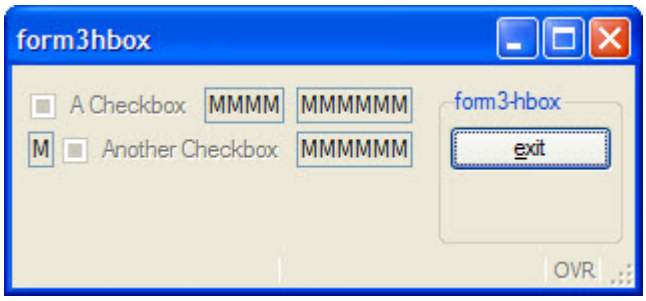


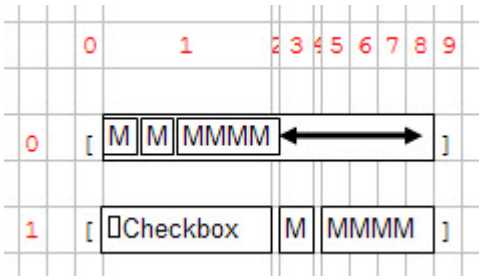
Figure 66: HBox rendering

Spacer items in hbox tags

HBox tags also introduces the spacer items concept: when a grid hbox is created, the content may be smaller than the container.

```
GRID
{
[a   :b   :c   ]
[d   :e   :f   ]
}
END

ATTRIBUTES
```

Figure 67: Spacer items



Because of the checkbox, the cell 1 is very large, and then the hbox is larger than the three fields. A spacer item object is automatically created by the form compiler; the role of the spacer item is to take all the free space in the container. Then all the widgets are packed to the left side of the hbox.

By default, a spacer item is created at the right of the container, but the spacer can also be defined in another place:

```
GRID
{
[a      :b      :c       ] <- default: spacer on the right
[ :d    :e      :f       ] <- spacer on the left
[g      :     :h         ] <- spacer between g and h
[i: :j: :k    : :l       ] <- multiple spacers (between i and j, j and k, k
 and l
}
END
```

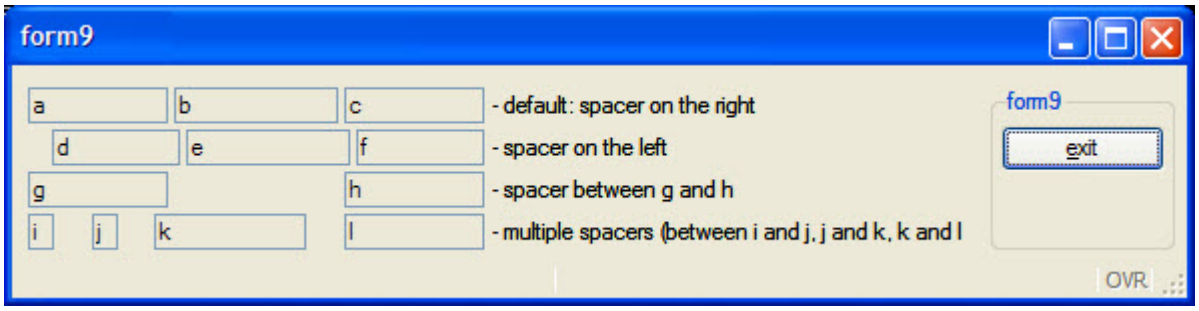


Figure 68: Form using spacers

Widget size within hbox tags

By default, the real width of `BUTTONEDIT`, `DATEEDIT`, `DATETIMEEDIT` and `COMBOBOX` widgets are computed as follows:

```
if item-tag-width > 2
   real-width = item-tag-width - 2
else
   real-width = item-tag-width
```

Where *item-tag-width* represents the number of characters used in the form layout by the item tag, to define the width of the element.

If the default widget size computing does not satisfy the needs, it is possible to specify the exact with of a `BUTTONEDIT`, `DATEDIT` or `COMBOBOX` with an hbox tag, combined to the `SAMPLE` attribute.

The hbox tag can be used with a `:` (colon) and `-` (dash) marker to define the exact number of characters the field can display, while the `SAMPLE` attribute will define the size.

For example:

```
LAYOUT
GRID
{
 ButtonEdit A  [ba     ]
 ButtonEdit B  [bb:    ]
 ButtonEdit C  [bc   : ]
 ButtonEdit D  [bd  -: ]
}
END
END
ATTRIBUTES
BUTTONEDIT ba = FORMONLY.ba, SAMPLE="0",  ACTION=zoom1;
BUTTONEDIT bb = FORMONLY.bb, SAMPLE="M",  ACTION=zoom2;
BUTTONEDIT bc = FORMONLY.bc, SAMPLE="Pi", ACTION=zoom3;
BUTTONEDIT bd = FORMONLY.bd, SAMPLE="0",  ACTION=zoom4;
END
```

Here the `ba` item tag occupies 7 grid columns and gets a real width of 5 (7-2). The `SAMPLE` attribute makes the edit field part as large as 5 characters '0' in the current font, so with this field you can input or display only 5 digits.

The `bb` item tag, which is in an hbox tag that occupies 7 grid columns, gets a width of 2. Since the `SAMPLE` attribute is "M", one can input 2 characters as wide as an "M".

The `bc` item tag, which is in an hbox tag that occupies 7 grid columns, gets a width of 3 (5-2). Since the `SAMPLE` attribute is "Pi", the edit field part will be as large as the word "Pi". (If `SAMPLE` contains more than 1 character it must have the same number of characters as in the field definition).

When using an hbox tag, one can explicitly specify the width of the field with the dash size indicator: The `bd`, which is in an hbox tag that occupies 7 grid columns, gets a width of 4 (because of the dash size indicator). Since the `SAMPLE` attribute is "0", the edit field part will be as large as 4 digits.

Stack-based layout
A form file can define a stack-based layout within a tree of stack items.

In a `.per` form specification file, the `LAYOUT` section defines a tree of layout containers, which hold layout items such as labels and form fields.

Use the `STACK` layout containers, to define a logical grouping of form elements, to be rendered vertically by the front end.

> **Important:** This feature is experimental and specific to mobile programming (STACK layout is not supported by GWC-JS and GDC). The syntax/name and semantics/behavior may change in a future version.

The `STACK` container defines a tree of stack containers, which holds a set of stack items such as form fields:

```
LAYOUT
STACK
   GROUP g1(TEXT="Customer info")
       EDIT customer.cust_num, NOENTRY, TITLE="Id:";
       EDIT customer.cust_name, TITLE="Name:";
       EDIT customer.cust_address, TITLE="Address:";
   END
END
END
```

There is no such thing as x,y coordinates in a stack container: The form element position definition is abstract and relative to other elements. Arranging form elements logically allows more flexibility in the final rendering of the form on the front-end.

Stack-based forms are typically used in mobile application design, to get a similar, but adaptable layout rendering on different mobile device brands.

The visual result of the above form definition would look as follows on an iOS mobile device:

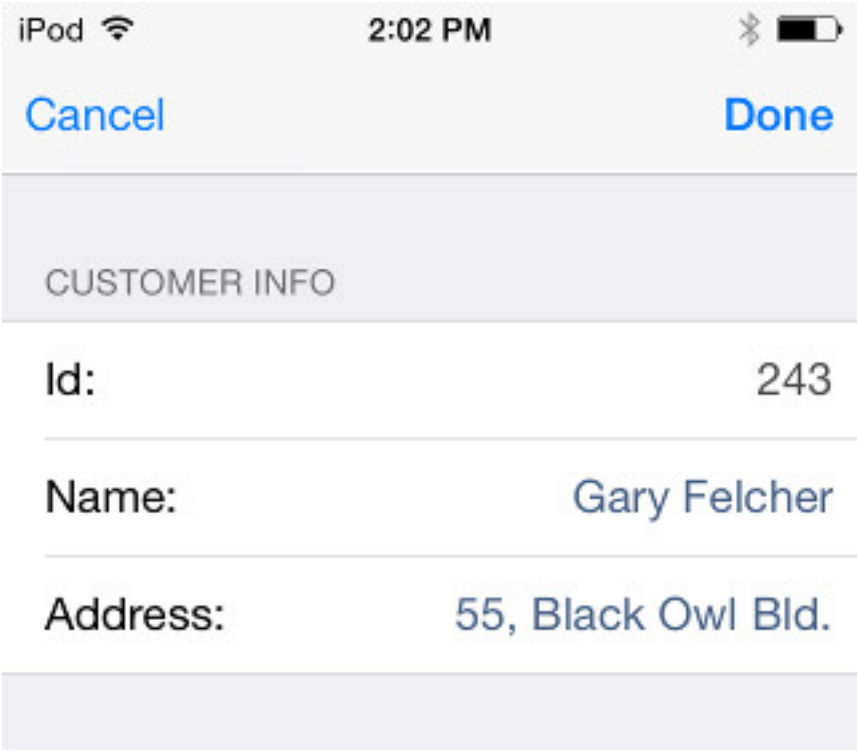


Figure 69: iOS stacked form 2

Label internationalization
Define form files with stacked containers for different languages.

To internationalize your application, define `TITLE` attributes using `%"`*`string-id`*`"` localized strings, in stack containrs and stack item definitions:

```
-- myform.per
LAYOUT
 STACK
   GROUP (TEXT=%"group.custinfo")
       EDIT customer.cust_num, NOENTRY, TITLE=%"cust.label.id";;
       EDIT customer.cust_name, TITLE=%"cust.label.name";
       EDIT customer.cust_address, TIELE=%"cust.label.address";
   END
 END
ED

-- myapp.str
"group.custinfo"     = "Customer information"
"cust.label.id"      = "Id:"
"cust.label.name"    = "Name:"
"cust.label.address" = "Address:"
```

If more space is needed for text fields, remove field labels and add a `COMMENT` attribute to show a grayed text inside empty fields:

```
-- myform.per
LAYOUT
 STACK
   GROUP (TEXT=%"group.custinfo")
       EDIT customer.cust_num, NOENTRY, TITLE=%"cust.label.id";;
       EDIT customer.cust_name, COMMENT=%"cust.comment.name";
       EDIT customer.cust_address, COMMENT=%"cust.comment.address";
```

```
   END
 END
END

-- myapp.str
"group.custinfo"       = "Customer information"
"cust.label.id"        = "Id:"
"cust.comment.name"    = "Customer's name"
"cust.comment.address" = "Customer's address"
```

The visual result for the about stack-based form will look like this on an iOS device:

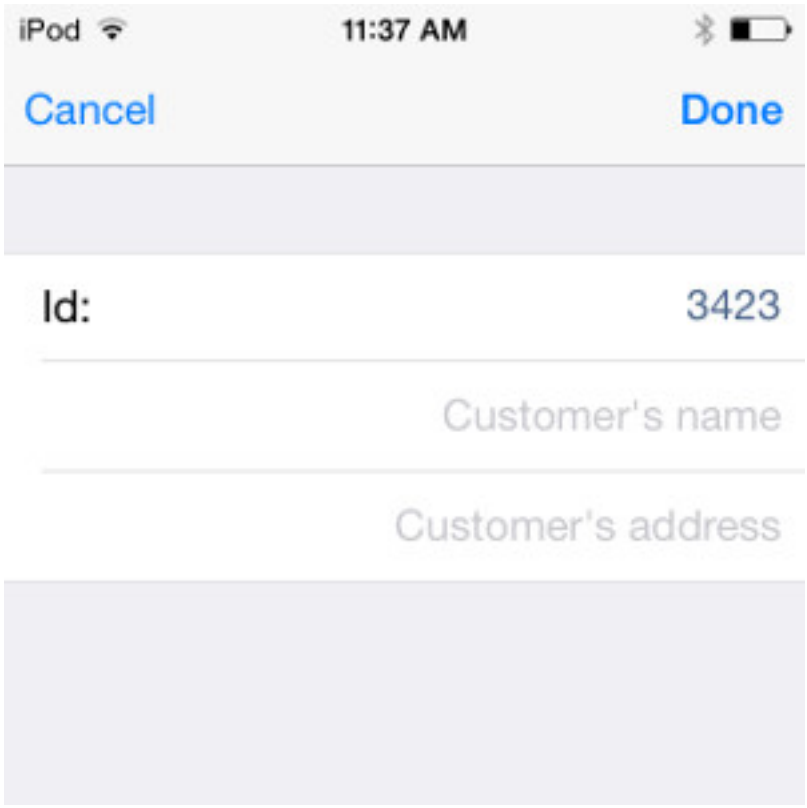


Figure 70: iOS stacked form 3

Stacked group rendering
Groups render in a native way on front-ends supporting the stacked layout.

Use GROUP containers in your form definition, to control the stacked layout: Fields and other form elements such as buttons can be grouped together by domain.

The header of a group box is defined by the TEXT attribute of the GROUP container.

For example, in a form designed for customer data input, customer identification (number, name) should appear in a dedicated group, while address information (street, zip code, state, country fields) should appear under another group:

```
-- myform.per
LAYOUT
 STACK
  GROUP g1 (TEXT=%"cust.group1")
    EDIT FORMONLY.id, TITLE=%"cust.label.id";
    EDIT FORMONLY.name, TITLE=%"cust.label.name";
  END
  GROUP g2  (TEXT=%"cust.group2")
```

```
    LABEL : l_street, TEXT=%"cust.label.street";
    TEXTEDIT FORMONLY.street, HEIGHT=3;
    EDIT FORMONLY.zipcode, TITLE=%"cust.label.zipcode";
    EDIT FORMONLY.state, TITLE=%"cust.label.state";
    EDIT FORMONLY.country, TITLE=%"cust.label.country";
  END
 END
END

-- myapp.str
"cust.group1"        = "Customer id"
"cust.label.id"      = "Id:"
"cust.label.name"    = "Name:"
"cust.group2"        = "Address"
"cust.label.street"  = "Street:"
"cust.label.zipcode" = "Zip Code:"
"cust.label.state"   = "State:"
"cust.label.country" = "Country:"
```

This code example will render as follows on an iOS mobile device:

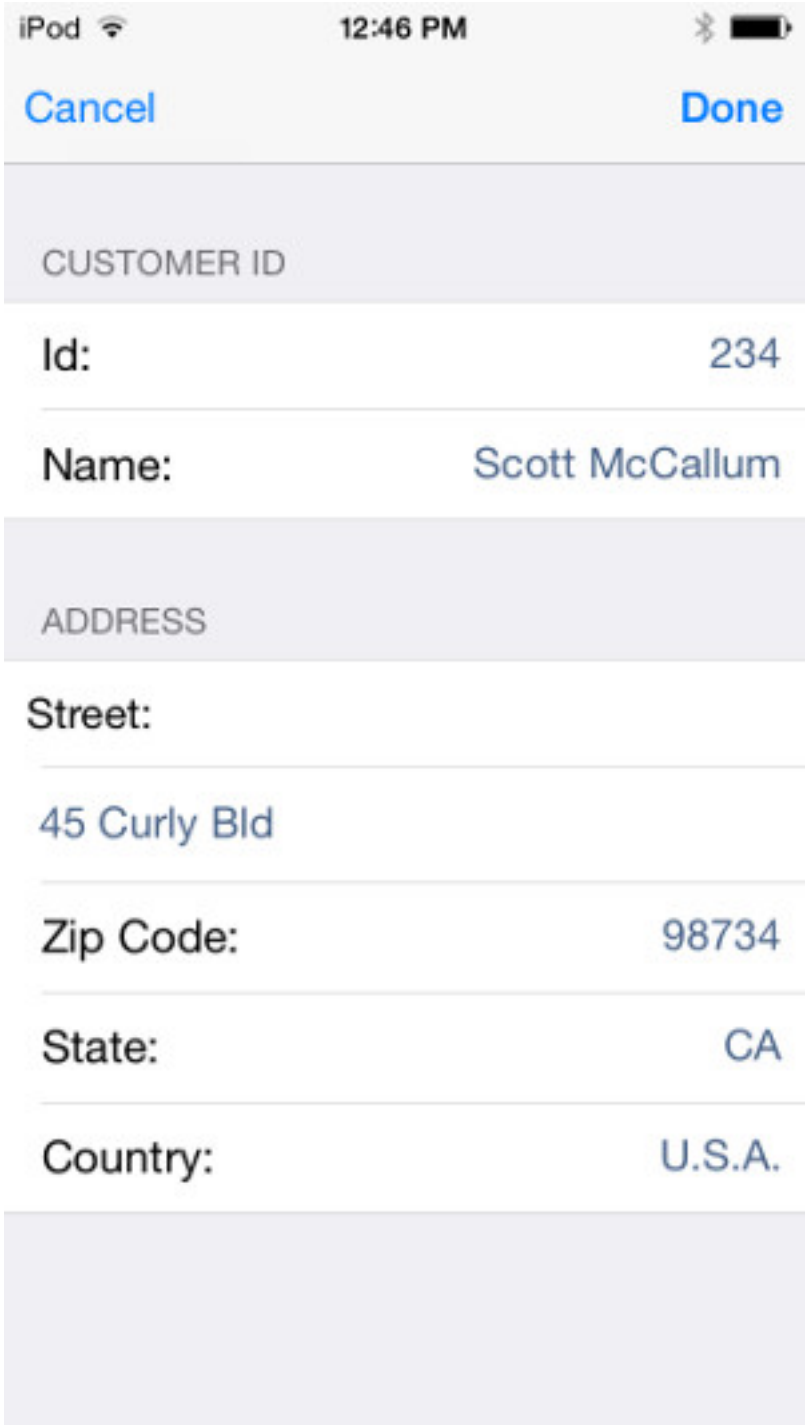


Figure 71: iOS stacked form 4

Toolbars

Toolbars define a bar of buttons that appears at the top of application forms.

- Understanding toolbars on page 1022
- Syntax of a toolbar file (.4tb) on page 1022
- Using toolbars on page 1023
  - Defining toolbars in the form file on page 1024
  - Loading a toolbar from an XML file on page 1024

- Loading a default toolbar from an XML file on page 1024
- Creating the toolbar manually with DOM on page 1024
- Toolbars on mobile devices on page 1025
- Examples on page 1025
  - Example 1: Toolbar in XML format on page 1025
  - Example 2: Toolbar created dynamically on page 1026
  - Example 3: Toolbar section in form file on page 1026

Understanding toolbars

A toolbar defines action views presented as a set of buttons that can trigger events in an interactive instruction.

This section describes how to define toolbars with XML in files or in programs as global/default toolbars; it is also possible to define toolbars in forms with the `TOOLBAR` section, as form-specific toolbars.

Toolbar files can be loaded by a program with the methods `ui.Interface.loadToolBar()` (for default toolbars) or `ui.Form.loadToolBar()` (for form-initializers).

A global toolbar is displayed by default in all windows, or in the global window container when using a window container. The form-specific toolbar is displayed in the form where it is defined. The position and visibility of toolbars can be controlled with a window style attribute. Typical "modal windows" do not display toolbars.

The toolbar items (or buttons) are enabled according to the `ON ACTION` handlers defined by the current interactive instruction. A toolbar item is bound to an action handler by name.. A click on the toolbar button will execute the user code in the action handler.

Toolbar elements can get a `style` attribute in order to use a specific rendering/decoration following presentation style definitions.

The DOM tag names are case sensitive; `Toolbar` is different from `ToolBar`.

When binding to an action, make sure that you are using the right value in the `name` attribute. As `ON ACTION` and `COMMAND` generate lowercase identifiers, it is recommended to use lowercase names.

It is recommended that you define the decoration of toolbar items for common actions with action defaults.

Syntax of a toolbar file (.4tb)

A toolbar file defines a set of buttons in a dedicated area of a window.

Toolbar XML syntax

```
<ToolBar [ toolbar-attribute="value" [...]] >
  { <ToolBarSeparator separator-attribute="value" [...] />
  | <ToolBarItem item-attribute="value" [...] />
  } [...]
</ToolBar>
```

1. *toolbar-attribute* defines a property of the toolbar.
2. *item-attribute* defines a property of the toolbar item.

Toolbar XML attributes

Table 261: ToolBar node attributes

Attribute	Type	Description
`style`	`STRING`	Use to decorate the element with a presentation style.

Attribute	Type	Description
tag	STRING	User-defined attribute to identify the node.
name	STRING	Identifies the toolbar.
buttonTextHidden	INTEGER	Defines if the text of toolbar buttons must appear by default.

Table 262: ToolBarItem node attributes

Attribute	Type	Description
name	STRING	Identifies the action corresponding to the toolbar button. Can be prefixed with the sub-dialog identifier.
style	STRING	Use to decorate the element with a presentation style.
tag	STRING	User-defined attribute to identify the node.
text	STRING	The text to be displayed in the toolbar button.
comment	STRING	The message to be shown as tooltip when the user selects a toolbar button.
hidden	INTEGER	Indicates if the item is hidden.
image	STRING	The icon to be used in the toolbar button.

Table 263: ToolBarSeparator node attributes

Attribute	Type	Description
name	STRING	Identifies the toolbar separator.
style	STRING	Use to decorate the element with a presentation style.
tag	STRING	User-defined attribute to identify the node.
hidden	INTEGER	Indicates if the separator is hidden.

Using toolbars

To use toolbars, you must understand how they work and how to structure the code.

- Understanding toolbars on page 1022
- Defining toolbars in the form file on page 1024
- Loading a toolbar from an XML file on page 1024
- Loading a default toolbar from an XML file on page 1024

- Creating the toolbar manually with DOM on page 1024
- Toolbars on mobile devices on page 1025

Defining toolbars in the form file
Toolbars can be defined in the form specification file within the `TOOLBAR` section.

Form toolbars are only displayed in the window where the form is loaded. Only one toolbar can be defined in a form file. Toolbar button attributes that are common to topmenu options should be centralized in action defaults.

Example

```
TOOLBAR tb
  ITEM accept ( TEXT="Ok", IMAGE="ok" )
  ITEM cancel ( TEXT="Cancel", IMAGE="cancel" )
  SEPARATOR
  ...
END

LAYOUT
GRID
{
  ...
```

Loading a toolbar from an XML file

To load a toolbar definition file (`4tb`) for a form, use the utility method provided by the `ui.Form` built-in class.

```
CALL myform.loadToolbar("standard")
```

Loading a default toolbar from an XML file
To load a default toolbar from an XML definition file, use the utility method provided by the `ui.Interface` built-in class.

```
CALL ui.Interface.loadToolbar("standard")
```

The default toolbar will be displayed in all forms.

Creating the toolbar manually with DOM

This example shows how to create a toolbar in all forms by using the default initialization function and the `om.DomNode` class:

```
CALL ui.Form.setDefaultInitializer("myinit")
OPEN FORM f1 FROM "form1"
DISPLAY FORM f1
...
FUNCTION myinit(form)
  DEFINE form ui.Form
  DEFINE f om.DomNode
  LET f = form.getNode()
  ...
END FUNCTION
```

After getting the DOM node of the form, create a node with the "`ToolBar`" tag name:

```
DEFINE tb om.DomNode
LET tb = f.createChild("ToolBar")
```

For each toolbar button, create a sub-node with the "`ToolBarItem`" tag name and set the attributes to define the button:

```
DEFINE tbi om.DomNode
LET tbi = tb.createChild("ToolBarItem")
CALL tbi.setAttribute("name","update")
CALL tbi.setAttribute("text","Modify")
CALL tbi.setAttribute("comment","Modify the current record")
CALL tbi.setAttribute("image","change")
```

If needed, you can create a "`ToolBarSeparator`" node to separate toolbar buttons:

```
DEFINE tbs om.DomNode
LET tbs = tb.createChild("ToolBarSeparator")
```

Toolbars on mobile devices
Toolbars can be used to control action view rendering on mobile devices.

On mobile devices, actions render usually as default action views, that display implicitly in dedicated panes on the screen. When displaying forms on a mobile front-end, you can use a toolbar to control the rendering of the actions.

Using toolbars for Android™ devices (GMA)

On Android devices, a `TOOLBAR` can be used to define the action views that appear in the Android action bar. Toolbar action views are listed first and ordered as they are defined in the `TOOLBAR` section, followed by the default action views for remaining actions that are not part of the `TOOLBAR` definition.

Using toolbars for iOS devices (GMI)

On iOS devices, a `TOOLBAR` renders as the iOS toolbar panel. This toolbar appears at the bottom of the screen, displaying a icon or text for each toolbar item. If there is not enough space to render all toobar items, a three-dot overflow icon appears on the right, to show up the remaining toolbar items.

The `iosSeparatorStretch` toolbar style attribute can be used to stretch the separators to give more space between action buttons.

Examples
Examples showing the various ways to define a toolbar.

- Example 1: Toolbar in XML format on page 1025
- Example 2: Toolbar created dynamically on page 1026
- Example 3: Toolbar section in form file on page 1026

Example 1: Toolbar in XML format

```
<ToolBar style="mystyle">
   <ToolBarItem name="f5" text="List" image="list" />
   <ToolBarSeparator/>
   <ToolBarItem name="query" text="Query" image="search" />
   <ToolBarItem name="add" text="Append" image="add" />
   <ToolBarItem name="delete" text="Delete" image="delete" />
   <ToolBarItem name="modify" text="Modify" image="change" />
   <ToolBarSeparator/>
   <ToolBarItem name="f1" text="Help" image="list" />
   <ToolBarSeparator/>
   <ToolBarItem name="quit" text="Quit" image="quit" />
</ToolBar>
```

Example 2: Toolbar created dynamically

```
MAIN
  DEFINE aui om.DomNode
  DEFINE tb  om.DomNode
  DEFINE tbi om.DomNode
  DEFINE tbs om.DomNode

  LET aui = ui.Interface.getRootNode()

  LET tb = aui.createChild("ToolBar")

  LET tbi = createToolBarItem(tb,"f1","Help","Show help","help")
  LET tbs = createToolBarSeparator(tb)
  LET tbi = createToolBarItem(tb,"upd","Modify","Modify current
 record","change")
  LET tbi = createToolBarItem(tb,"del","Remove","Remove current
 record","delete")
  LET tbi = createToolBarItem(tb,"add","Append","Add a new record","add")
  LET tbs = createToolBarSeparator(tb)
  LET tbi = createToolBarItem(tb,"xxx","Exit","Quit application","quit")

  MENU "Example"
    COMMAND KEY(F1)
      DISPLAY "F1 action received"
    COMMAND "upd"
      DISPLAY "Update action received"
    COMMAND "Del"
      DISPLAY "Delete action received"
    COMMAND "Add"
      DISPLAY "Append action received"
    COMMAND "xxx"
      EXIT PROGRAM
  END MENU

END MAIN

FUNCTION createToolBarSeparator(tb)
  DEFINE tb om.DomNode
  DEFINE tbs om.DomNode
  LET tbs = tb.createChild("ToolBarSeparator")
  RETURN tbs
END FUNCTION

FUNCTION createToolBarItem(tb,n,t,c,i)
  DEFINE tb om.DomNode
  DEFINE n,t,c,i VARCHAR(100)
  DEFINE tbi om.DomNode
  LET tbi = tb.createChild("ToolBarItem")
  CALL tbi.setAttribute("name",n)
  CALL tbi.setAttribute("text",t)
  CALL tbi.setAttribute("comment",c)
  CALL tbi.setAttribute("image",i)
  RETURN tbi
END FUNCTION
```

Example 3: Toolbar section in form file

```
TOOLBAR ( STYLE="mystyle" )
  ITEM accept ( TEXT="Ok", IMAGE="ok" )
  ITEM cancel ( TEXT="cancel", IMAGE="cancel" )
  SEPARATOR
  ITEM editcut   -- Gets decoration from action defaults
```

```
  ITEM editcopy  -- Gets decoration from action defaults
  ITEM editpaste -- Gets decoration from action defaults
  SEPARATOR
  ITEM append ( TEXT="Append", IMAGE="add" )
  ITEM update ( TEXT="Update", IMAGE="modify" )
  ITEM delete ( TEXT="Delete", IMAGE="del" )
  ITEM search ( TEXT="Search", IMAGE="find" )
END
```

Topmenus

Topmenus define typical pull-down menus that appear at the top of application forms.

- Understanding topmenus on page 1027
- Syntax of a topmenu file (.4tm) on page 1028
- Using topmenus on page 1030
  - Defining the topmenu in a form file on page 1030
  - Loading a topmenu from an XML file on page 1030
  - Loading a default topmenu from an XML file on page 1030
  - Creating the topmenu dynamically on page 1030
  - Topmenus on mobile devices on page 1031
- Examples on page 1033
  - Example 1: Topmenu in XML format on page 1033
  - Example 2: Topmenu section in form file on page 1033

Understanding topmenus

A topmenu defines a graphical menu that holds views for actions controlled in programs with `ON ACTION` handlers. A topmenu renders to the user according to the front-end platform standards. On a desktop / web front-end, the topmenu appears as a typical pull-down menu. On mobile devices, a topmenu displays as a flat list of options (Android™), and as a set of option screens the user can drill down (iOS).

This section describes how to define topmenus with XML in files or in programs as global/default topmenus; it is also possible to define topmenus in forms with the `TOPMENU` section, as form-specific topmenus.

Topmenu files can be loaded by program with the methods `ui.Interface.loadTopMenu()` (for default topmenus) or `ui.Form.loadTopMenu()` (for form-initializers).

In the abstract user interface tree, the `TopMenu` node must be created under the `Form` node, and must contain `TopMenuGroup` nodes. The `TopMenuGroup` nodes group topmenu commands and other topmenu groups. A `TopMenuCommand` is a leaf node in the topmenu tree that will trigger an action:

```
TopMenu
 +- TopMenuGroup
  +- TopMenuCommand
  +- TopMenuCommand
  +- TopMenuCommand
 +- TopMenuGroup
  +- TopMenuGroup
   +- TopMenuCommand
   +- TopMenuCommand
  +- TopMenuGroup
   +- TopMenuCommand
   +- TopMenuCommand
   +- TopMenuCommand
```

The topmenu options are enabled according to the `ON ACTION` handlers defined by the current interactive instruction. A topmenu option is bound to an action handler by name. Selecting the topmenu option will execute the user code in the action handler.

Topmenu elements can get a `style` attribute in order to use a specific rendering/decoration following presentation style definitions.

The DOM tag names are case sensitive; `Topmenu` is different from `TopMenu`.

When binding to an action, make sure that you are using the right value in the `name` attribute. As `ON ACTION` and `COMMAND` generate lowercase identifiers, it is recommended to use lowercase names.

It is recommended that you define the decoration of topmenu options for common actions with action defaults.

Images cannot be displayed for the first level of `TopMenuGroup` elements.

Syntax of a topmenu file (.4tm)
A topmenu file defines a tree of menu options to be displayed in the header of a window.

Topmenu XML syntax

```
<TopMenu [ topmenu-attribute="value" [...] ] >
    group
    [...]
  </TopMenu>
```

where *group* is:

```
<TopMenuGroup group-attribute="value" [...]>
    { <TopMenuSeparator separator-attribute="value" [...] />
    | <TopMenuCommand command-attribute="value" [...] />
    | group
    } [...]
  </TopMenuGroup>
```

1. *topmenu-attribute* defines a property of the `TopMenu`.
2. *group-attribute* defines a property of a `TopMenuGroup`.
3. *command-attribute* defines a property of a `TopMenuCommand`.
4. *separator-attribute* defines a property of a `TopMenuSeparator`.

Topmenu XML attributes

Table 264: TopMenu node attributes

Attribute	Type	Description
`name`	`STRING`	Identifies the topmenu.
`style`	`STRING`	Can be used to decorate the element with a presentation style.
`tag`	`STRING`	User-defined attribute to identify the node.

Table 265: TopMenuCommand node attributes

Attribute	Type	Description
name	STRING	Identifies the action corresponding to the topmenu command. Can be prefixed with the sub-dialog identifier.
style	STRING	Can be used to decorate the element with a presentation style.
tag	STRING	User-defined attribute to identify the node.
text	STRING	The text to be displayed in the pull-down menu option.
comment	STRING	The message to be shown for this element.
hidden	INTEGER	Indicates if the command is hidden.
image	STRING	The icon to be used in the pull-down menu option.
acceleratorName	STRING	Defines the accelerator name to be display on the left of the menu option text. Note this attribute is only used for decoration (you must also define an action default accelerator).

Table 266: TopMenuGroup node attributes

Attribute	Type	Description
name	STRING	Identifies the topmenu group.
style	STRING	Can be used to decorate the element with a presentation style.
tag	STRING	User-defined attribute to identify the node.
text	STRING	The text to be displayed in the pull-down menu group.
comment	STRING	The message to be shown for this element.
hidden	INTEGER	Indicates if the group is hidden.
image	STRING	The icon to be used in the pull-down menu group.

Table 267: Separator-attributes for the TopMenuSeparator node

Attribute	Type	Description
name	STRING	Identifies the topmenu separator.
style	STRING	Can be used to decorate the element with a presentation style.
tag	STRING	User-defined attribute to identify the node
hidden	INTEGER	Indicates if the separator is hidden.

Using topmenus

To use topmenus, you must understand how they work and how to structure the code.

Defining the topmenu in a form file

Topmenus can be defined in the form specification file within the TOPMENU section.

Form topmenus will only be displayed in the window where the form is loaded. Only one topmenu can be defined in a form file. Topmenu item attributes that are common to toolbar buttons should be centralized in action defaults.

Example

```
TOPMENU tm
  GROUP form (TEXT="Form")
    COMMAND help (TEXT="Help", IMAGE="quest")
    COMMAND quit (TEXT="Quit")
  END
  ...
END

LAYOUT
GRID
{
  ...
```

Loading a topmenu from an XML file

To load a .4tm topmenu definition file for a form, use the utility method provided by the ui.Form built-in class:

```
CALL myform.loadTopMenu("standard")
```

Loading a default topmenu from an XML file

To load a default topmenu from an XML definition file, use the utility method provided by the ui.Interface built-in class.

```
CALL ui.Interface.loadTopMenu("standard")
```

The default topmenu will be displayed in all forms.

Creating the topmenu dynamically

This example shows how to create a topmenu in all forms by using the default initialization function and the om.DomNode class:

```
CALL ui.Form.setDefaultInitializer("myinit")
OPEN FORM f1 FROM "form1"
```

```
DISPLAY FORM f1
...
FUNCTION myinit(form)
  DEFINE form ui.Form
  DEFINE f om.DomNode
  LET f = form.getNode()
  ...
END FUNCTION
```

After getting the DOM node of the form, create a node with the "`TopMenu`" tag name:

```
DEFINE tm om.DomNode
LET tm = f.createChild("TopMenu")
```

For each Topmenu group, create a subnode with the "`TopMenuGroup`" tag name and set the attributes to define the group:

```
DEFINE tmg om.DomNode
LET tmg = tm.createChild("TopMenuGroup")
CALL tmg.setAttribute("text","Reports")
```

For each Topmenu option, create a sub-node in a group node with the "`TopMenuCommand`" tag name and set the attributes to define the option:

```
DEFINE tmi om.DomNode
LET tmi = tmg.createChild("TopMenuCommand")
CALL tmi.setAttribute("name","report")
CALL tmi.setAttribute("text","Order report")
CALL tmi.setAttribute("comment","Orders entered today")
CALL tmi.setAttribute("image","smiley")
```

If needed, you can create a "`TopMenuSeparator`" node inside a group, to separate menu options:

```
DEFINE tms om.DomNode
LET tms = tmg.createChild("TopMenuSeparator")
```

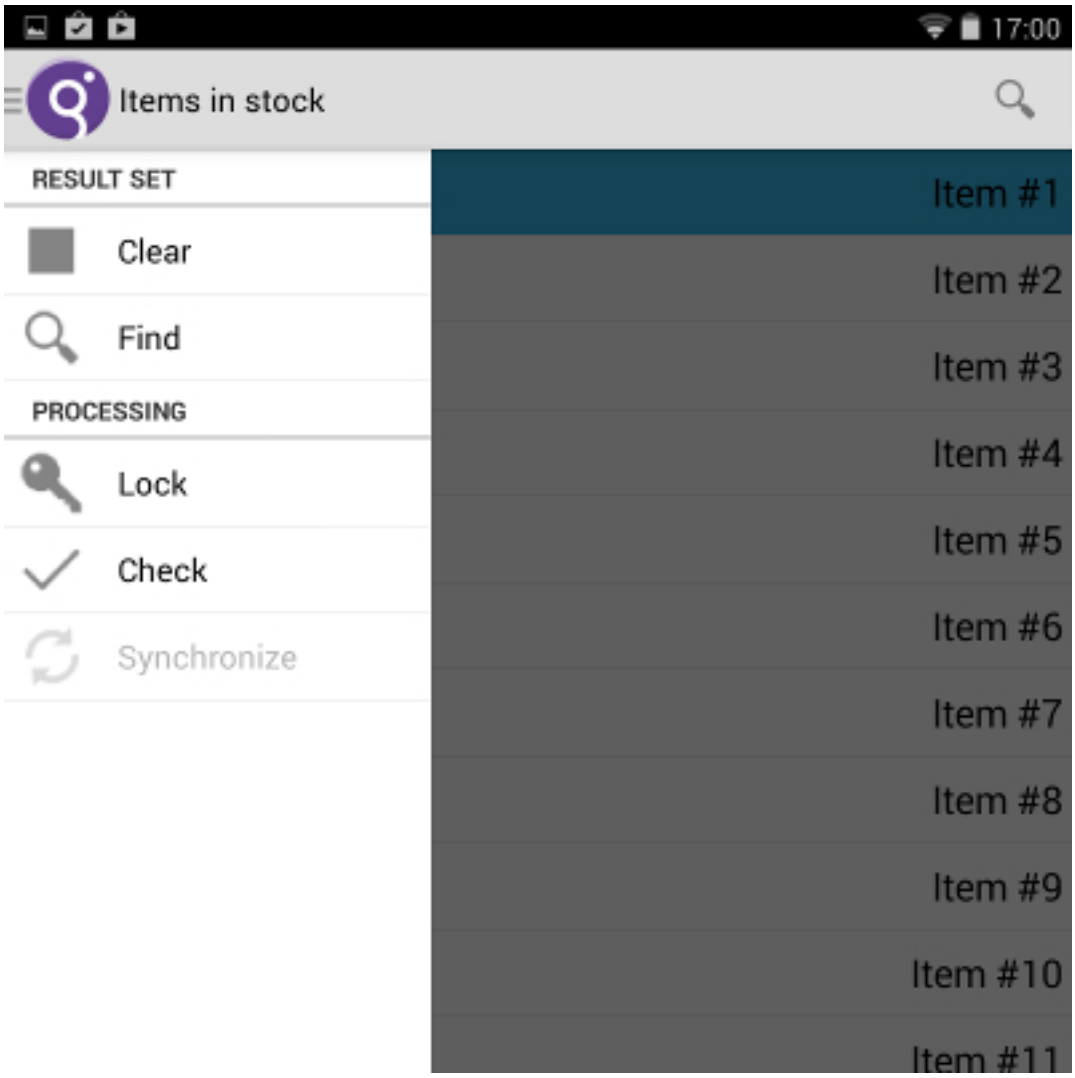
Topmenus on mobile devices
Topmenus can be used to implement a general options menu in mobile apps.

On mobile devices, actions render usually as default action views, that display implicitly in dedicated panes on the screen. When displaying forms on a mobile front-end, you can use a topmenu to get a list of options the end user can choose from.

Using topmenus for Android™ devices (GMA)

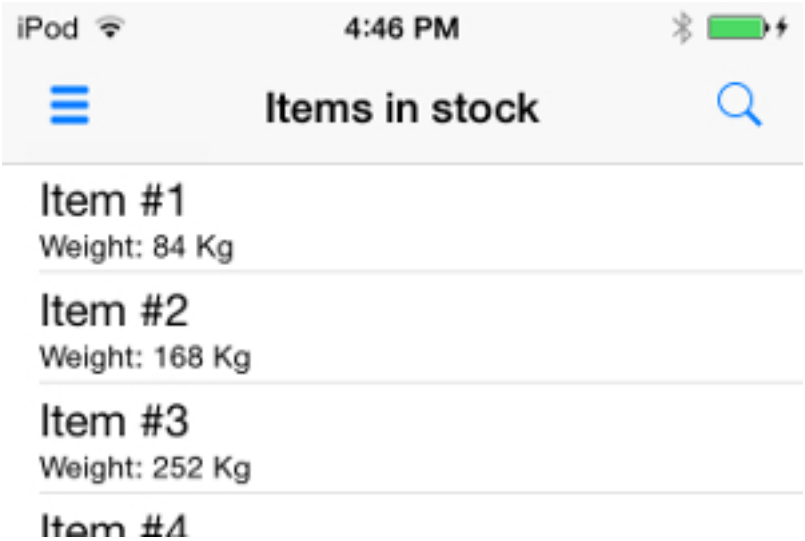
On Android devices, a `TOPMENU` renders as a menu icon on the top left of the screen, in the Android action bar. When the user taps on this icon, a list with topmenu items shows up. Selecting an option fires the corresponding action handler is fired.

> **Note:** On Android, the topmenu can only display one level of options (no tree of options is possible).



Using topmenus for iOS devices (GMI)

On iOS devices, a `TOPMENU` renders as a menu icon on the top left corner of the device screen, in the iOS navigation controller. When the user taps on this icon, a new view appears with the first level of topmenu items. The user can the drill down to a next level, select an option if it's a leaf item, or tap on the back button to move one level up in the topmenu tree. Selecting a leaf item will fire the corresponding action handler and close the menu.



Examples

Example 1: Topmenu in XML format

```
<TopMenu>
   <TopMenuGroup text="Form" style="mystyle">
      <TopMenuCommand name="help" text="Help" image="quest" />
      <TopMenuCommand name="quit" text="Quit" acceleratorName="alt-F4"/>
   </TopMenuGroup>
   <TopMenuGroup text="Edit">
      <TopMenuCommand name="accept" text="Validate" image="ok" />
      <TopMenuCommand name="cancel" text="Cancel" image="cancel" />
      <TopMenuSeparator/>
      <TopMenuCommand name="editcut" text="Cut" />
      <TopMenuCommand name="editcopy" text="Copy" />
      <TopMenuCommand name="editpaste" text="Paste" />
   </TopMenuGroup>
   <TopMenuGroup text="Records">
      <TopMenuCommand name="append" text="Add" image="add" />
      <TopMenuCommand name="delete" text="Remove" image="delete" />
      <TopMenuCommand name="update" text="Modify" image="change" />
      <TopMenuSeparator/>
      <TopMenuCommand name="search" text="Query" image="find" />
   </TopMenuGroup>
</TopMenu>
```

Example 2: Topmenu section in form file

```
TOPMENU
  GROUP form (TEXT="Form", STYLE="mystyle" )
     COMMAND help (TEXT="Help", IMAGE="quest")
     COMMAND quit (TEXT="Quit", ACCELERATOR=ALT-F4)
  END
  GROUP edit (TEXT="Edit")
     COMMAND accept (TEXT="Validate", IMAGE="ok")
     COMMAND cancel (TEXT="Cancel", IMAGE="cancel")
     SEPARATOR
     COMMAND editcut   -- Gets decoration from action defaults
     COMMAND editcopy  -- Gets decoration from action defaults
     COMMAND editpaste -- Gets decoration from action defaults
  END
  GROUP records (TEXT="Records")
     COMMAND append (TEXT="Add", IMAGE="add")
```

```
      COMMAND delete (TEXT="Remove", IMAGE="del")
      COMMAND update (TEXT="Modify", IMAGE="change")
      SEPARATOR
      COMMAND search (TEXT="Search", IMAGE="find")
  END
END
```

Dialog instructions

This section describes the dialog instructions to control application forms and the concepts related to dialog implementation.

- Static display (DISPLAY/ERROR/MESSAGE/CLEAR) on page 1034
- Prompt for values (PROMPT) on page 1042
- Ring menus (MENU) on page 1048
- Record input (INPUT) on page 1060
- Read-only record list (DISPLAY ARRAY) on page 1075
- Editable record list (INPUT ARRAY) on page 1098
- Query by example (CONSTRUCT) on page 1128
- Multiple dialogs (DIALOG) on page 1144
- Parallel dialogs (START DIALOG) on page 1199

Static display (DISPLAY/ERROR/MESSAGE/CLEAR)

This section explains the instructions displaying static information to application forms, such as DISPLAY, ERROR, MESSAGE, CLEAR.

- Display of data and messages on page 1034
- DISPLAY (to stdout) on page 1035
- MESSAGE on page 1035
- ERROR on page 1036
- DISPLAY TO on page 1037
- DISPLAY BY NAME on page 1038
- CLEAR FORM on page 1040
- CLEAR SCREEN ARRAY on page 1040
- CLEAR field-list on page 1041
- SCROLL on page 1042

Display of data and messages

The values contained in program variables can be displayed to the current form with the `DISPLAY BY NAME` or `DISPLAY TO` instruction. Forms can be cleared with the `CLEAR FORM` or `CLEAR` *`field-list`* instructions. Complete record lists (in `SCROLLGRID`, `TABLE` or `TREE` containers) can be cleared with the `CLEAR SCREEN ARRAY` instruction. Application messages and warnings can be displayed to the user with the `MESSAGE` and `ERROR` instructions.

The `DISPLAY BY NAME/TO` instructions are not interactive, and are usually not needed if the program is always in the context of a dialog controlling the form fields: The data of the program variables will be displayed in form fields when the dialog starts, if the `WITHOUT DEFAULTS` option is specified, and during the dialog execution, form fields will be automatically synchronized with the program variables when using the `UNBUFFERED` option.

DISPLAY (to stdout)

The DISPLAY instruction displays text in line mode to the standard output channel.

Syntax

```
DISPLAY expression
```

1. *expression* is any expression supported by the language.

Usage

The DISPLAY instruction can be used to print information to the standard output channel (stdout) of the terminal the program is attached to.

The *expression* is typically a list of string constants and program variables separated by the comma concatenation operator.

Before displaying to the standard output channel, the expression is converted to a character string. The values contained in variables are formatted according to the data types and environment settings such as DBDATE and DBMONEY.

Example

```
MAIN
  DISPLAY "Today's date is: ", TODAY
END MAIN
```

MESSAGE

The MESSAGE instruction displays a message to the user.

Syntax

```
MESSAGE message [,...]
  [ ATTRIBUTES ( display-attribute [,...] ) ]
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
| STYLE = "style-name"
}
```

1. *expression* is any expression supported by the language.
2. *style-name* is a presentation style name.

Usage

The MESSAGE instruction displays a message to the user in an interactive program.

In TUI mode, the text is displayed in the comment line of the current window.

In GUI mode, the text is displayed in a specific area, depending on the STYLE attribute.

When you specify the STYLE attribute, you can reference a style defined in the presentation styles file. This allows you to display errors or messages in GUI mode with more sophisticated visual effects as the regular TTY attributes. Advanced automatic rendering can be obtained with message specific style attributes. If

you want to apply automatically a style to all program messages displayed with the MESSAGE instruction, you can use the `:message` pseudo selector in the style definition.

Example

```
INPUT BY NAME custrec.* ...
  BEFORE INPUT
    MESSAGE "Enter customer data."
  ...
```

ERROR
The ERROR instruction displays an error message to the user.

Syntax

```
ERROR expression
  [ ATTRIBUTES ( display-attribute [,...] ) ]
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
| STYLE = "style-name"
}
```

1. *expression* is any expression supported by the language.
2. *style-name* is a presentation style name.

Usage

The ERROR instruction displays an error message to the user in an interactive program.

In TUI mode, the error text is displayed in the error line of the current window.

In GUI mode, the text is displayed in a specific area, depending on the STYLE attribute.

When you specify the STYLE attribute, you can reference a style defined in the presentation styles file. This allows you to display errors or messages in GUI mode with more sophisticated visual effects as the regular TTY attributes. Advanced automatic rendering can be obtained with message specific style attributes. If you want to apply automatically a style to all program warnings displayed with the ERROR instruction, you can use the `:error` pseudo selector in the style definition.

Example

```
...
IF sqlca.sqlcode THEN
  ERROR "Database update failed (" || sqlca.sqlcode || ")"
    ATTRIBUTES(STYLE="important")
    ...
END IF
...
```

DISPLAY TO

The DISPLAY TO instruction displays data to form fields explicitly.

Syntax

```
DISPLAY expression [,...] TO field-spec [,...]
  [ ATTRIBUTES ( display-attribute [,...] ) ]
```

where *field-spec* is:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

1. *expression* is any expression supported by the language.
2. *field-name* is the identifier of a field of the current form.
3. *table-name* is the identifier of a database table of the current form.
4. *screen-record* is the identifier of a screen record of the current form.
5. *screen-array* is the screen array that will be used in the form.

Usage

A DISPLAY TO statement copies the data from program variables to the form fields specified after the TO keyword.

When the program variables do not have the same names as the form fields, you must use the TO clause to explicitly map the variables to fields. You can list the fields individually, or you can use the *screen-record*.* or *screen-record*[n].* notation, where *screen-record*[n].* specifies all the fields in line n of a screen array.

In the next example, the values in the p_items program record are displayed in the first row of the s_items screen array:

```
DISPLAY p_items.* TO s_items[1].*
```

The expanded list of screen fields must correspond in order and in number to the expanded list of identifiers after the DISPLAY keyword. Identifiers and their corresponding fields must have the same or compatible data types. For example, the next DISPLAY statement displays the values in the p_customer program record in fields of the s_customer screen record:

```
DISPLAY p_customer.* TO s_customer.*
```

For this example, the p_customer program record and the s_customer screen record require compatible declarations. The following DEFINE statement declares the p_customer program record:

```
DEFINE p_customer RECORD
```

```
  customer_num LIKE customer.customer_num,
  fname LIKE customer.fname,
  lname LIKE customer.lname,
  phone LIKE customer.phone
END RECORD
```

This fragment of a form specification declares the s_customer screen record:

```
ATTRIBUTES
 f000 = customer.customer_num;
 f001 = customer.fname;
 f002 = customer.lname;
 f003 = customer.phone;
END
```

The DISPLAY TO instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

DISPLAY TO changes the touched flag

The DISPLAY TO statement changes the 'touched' status of the target fields. When you modify a field value with this instruction, the FIELD_TOUCHED( ) operator returns true and the ON CHANGE and ON ROW CHANGE triggers may be invoked if the current field value was changed with a DISPLAY BY NAME.

In dialogs controlling field input such as INPUT or INPUT ARRAY, use the UNBUFFERED attribute to display data to fields automatically without changing the 'touched' status of fields. The UNBUFFERED clause will make automatic form field and program variable synchronization. When using the UNBUFFERED mode, the touched flag can be set with DIALOG.setFieldTouched( ) if you want to get the same effect as a DISPLAY BY NAME

Specifying TTY attributes in the DISPLAY BY NAME statement

The ATTRIBUTES clause temporarily overrides any default display attributes or any attributes specified in the OPTIONS or OPEN WINDOW statements for the fields. When the DISPLAY TO statement completes execution, the default display attributes are restored.In a DISPLAY TO statement, any screen attributes specified in the ATTRIBUTES clause apply to all the fields that you specify after the TO keyword.

The REVERSE, BLINK, INVISIBLE, and UNDERLINE attributes are not sensitive to the color or monochrome status of the terminal, if the terminal is capable of displaying these intensity modes. The ATTRIBUTES clause can include zero or more of the BLINK, REVERSE, and UNDERLINE attributes, and zero or one of the other attributes. That is, all of the attributes except BLINK, REVERSE, and UNDERLINE are mutually exclusive.

The DISPLAY TO statement ignores the INVISIBLE attribute, regardless of whether you specify it in the ATTRIBUTES clause.

DISPLAY BY NAME

The DISPLAY BY NAME instruction displays data to form fields explicitly *by name*.

Syntax

```
DISPLAY BY NAME { variable | record.* } [,...]
  [ ATTRIBUTES ( display-attribute [,...] ) ]
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | NORMAL
| REVERSE | BLINK | UNDERLINE
```

```
}
```

1. *variable* is a program variable that has the same name as a form field.
2. *record.** is a record variable that has members with the same names as form fields.

Usage

A `DISPLAY BY NAME` statement copies the data from program variables to the form fields associated to the variables by name. The program variables used in `DISPLAY BY NAME` must have the same name as the form fields where they have to be displayed. The language ignores any record structure name prefix when matching the names. The names must be unique and unambiguous; if not, the instruction raises an error.

For example, the following statement displays the values for the specified variables in the form fields with corresponding names (`company` and `address1`):

```
DISPLAY BY NAME p_customer.cust_company,
                p_customer.cust_address1
```

The `DISPLAY BY NAME` instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

DISPLAY BY NAME uses the default screen record

Unlike the `DISPLAY TO` instruction where you can explicitly specify a screen record or screen array, `DISPLAY BY NAME` displays data to the screen fields of the default screen records. The default screen records are those having the names of the tables defined in the `TABLES` section of the form specification file. When the form fields define a record list in the layout, only the first row can be referenced with the default screen record. In the next example, the form contains a static record list definition in the layout.

```
SCHEMA mystock
SCREEN
{
[f01  |f02    |f03          ]
[f01  |f02    |f03          ]
[f01  |f02    |f03          ]
[f01  |f02    |f03          ]
}
END
TABLES
customer
END
ATTRIBUTES
f01 = customer.cust_key;
f02 = customer.cust_name;
f03 = customer.cust_address;
END
```

In the program, a `DISPLAY BY NAME` statement will display the data in the first line of the record list in the form:

```
DISPLAY BY NAME record_cust.*
```

DISPLAY BY NAME changes the touched flag

The `DISPLAY BY NAME` statement changes the 'touched' status of the target fields. When you modify a field value with this instruction, the `FIELD_TOUCHED()` operator returns true and the `ON CHANGE` and `ON ROW CHANGE` triggers may be invoked if the current field value was changed with a `DISPLAY BY NAME`.

In dialogs controlling field input such as `INPUT` or `INPUT ARRAY`, use the `UNBUFFERED` attribute to display data to fields automatically without changing the 'touched' status of fields. The `UNBUFFERED` clause will make automatic form field and program variable synchronization. When using the `UNBUFFERED` mode, the touched flag can be set with `DIALOG.setFieldTouched()` if you want to get the same effect as a `DISPLAY BY NAME` statement.

Specifying TTY attributes in the DISPLAY BY NAME statement

The `ATTRIBUTES` clause temporarily overrides any default display attributes or any attributes specified in the `OPTIONS` or `OPEN WINDOW` statements for the fields. When the `DISPLAY BY NAME` statement completes execution, the default display attributes are restored.

The `REVERSE`, `BLINK`, `INVISIBLE`, and `UNDERLINE` attributes are not sensitive to the color or monochrome status of the terminal, if the terminal is capable of displaying these intensity modes. The `ATTRIBUTES` clause can include zero or more of the `BLINK`, `REVERSE`, and `UNDERLINE` attributes, and zero or one of the other attributes. That is, all of the attributes except `BLINK`, `REVERSE`, and `UNDERLINE` are mutually exclusive.

The `DISPLAY BY NAME` statement ignores the `INVISIBLE` attribute, regardless of whether you specify it in the `ATTRIBUTES` clause.

CLEAR FORM

The `CLEAR FORM` instruction clears all fields in the current form.

Syntax

```
CLEAR FORM
```

Usage

The `CLEAR FORM` instruction clears all form fields of the current form. It has no effect on any part of the screen display except the form fields.

Similarly to `CLEAR field-list`, the `CLEAR FORM` instruction is typically used when the program is not inside a dialog block execution controlling the form fields. For example, after a database query with a `CONSTRUCT` instruction, you might want to clear all search criteria entered by the user with this instruction, to cleanup the form.

The `CLEAR FORM` instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

Example

```
CONSTRUCT BY NAME sql
   ON cust_name, cust_address, ...
   ...
END CONSTRUCT
CLEAR FORM
...
```

CLEAR SCREEN ARRAY

The `CLEAR SCREEN ARRAY` instruction clears the values of all rows of the form list identified by the specified screen array.

Syntax

```
CLEAR SCREEN ARRAY screen-array.*
```

1. *screen-array* is a screen array specified in the form.

Usage

After executing a `DISPLAY ARRAY` or `INPUT ARRAY` instruction, values remain in the form list identified by the screen array.

The `CLEAR SCREEN ARRAY` instruction automatically clears all rows of the list, regardless of the view: a `TABLE`, `TREE`, `SCROLLGRID`, or in a matrix of fields (an old-style/text-mode static screen array).

The `CLEAR SCREEN ARRAY` instruction replaces code which clears each individual row through the use of a loop:

```
-- Clearing each row individually
FOR i=1 TO <screen-array-length>
  CLEAR screen-array[i].*
END FOR
-- Unique instruction to clear a list
CLEAR SCREEN ARRAY screen-array.*
```

Using the `CLEAR SCREEN ARRAY` instruction eliminates the need for calculating the screen array length, a value which can change when using a `TABLE` container, that can be resized.

The `CLEAR SCREEN ARRAY` instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

Example

```
...
   DISPLAY ARRAY cust_arr TO sa.*
   ...
   CLEAR SCREEN ARRAY sa.*
   ...
```

CLEAR *field-list*

The `CLEAR` *`field-list`* instruction clears specific fields in the current form.

Syntax

```
CLEAR field-list
```

where *field-list* is:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
}
[,...]
```

1. *field-name* is the identifier of a field of the current form.
2. *table-name* is the identifier of a database table of the current form.
3. *screen-record* is the identifier of a screen record of the current form.
4. *screen-array* is the screen array that will be used in the form.

Usage

The CLEAR *field-list* instruction can be used to clear the content of the specified form fields

Similarly to CLEAR FORM, the CLEAR *field-list* is typically used when the program is not inside a dialog block execution controlling the form fields. For example, after a database query with a CONSTRUCT instruction, you might want to clear all search criteria entered by the user with this instruction, to cleanup the form.

The CLEAR *field-list* instruction is usually not needed if the program is always in the context of a dialog controlling the form fields.

Example

```
CONSTRUCT BY NAME sql
   ON s_customer.*
   ...
END CONSTRUCT
CLEAR s_customer.*
...
```

SCROLL
The SCROLL instruction moves data rows up or down in a screen array.

Syntax

```
SCROLL field-list { UP | DOWN } [ BY lines ]
```

where *field-list* is:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

1. *field-name* is the identifier of a field of the current form.
2. *table-name* is the identifier of a database table of the current form.
3. *screen-record* is the identifier of a screen record of the current form.
4. *screen-array* is the name of the screen array used of the current form.
5. *lines* is an integer expression that specifies how far (in lines) to scroll the display.

Usage

The SCROLL instruction specifies vertical movements of displayed values in all or some of the fields of a screen array within the current form.

The SCROLL instruction is supported for applications running in TUI mode, to scroll screen array rows when no interactive instruction is executing. In a GUI application, use a TABLE container with a DISPLAY ARRAY instruction.

Prompt for values (PROMPT)

The PROMPT instruction provides unique field input in an automatic popup window.

- Understanding the PROMPT instruction on page 1043

- Syntax of PROMPT instruction on page 1043
- Using simple prompt inputs on page 1044
  - PROMPT programming steps on page 1044
  - PROMPT instruction configuration on page 1045
  - Default actions in PROMPT on page 1045
  - Interaction blocks on page 1046
- Examples on page 1047
  - Example 1: Simple PROMPT statements on page 1047
  - Example 2: Simple PROMPT with Interrupt Checking on page 1047
  - Example 3: PROMPT with ATTRIBUTES and ON ACTION handlers on page 1047

Understanding the PROMPT instruction

Use the PROMPT instruction to query for a single value from the user.

PROMPT requires the text of the question to be displayed to the user and the variable that receives the value entered by the user. The variable can be of any simple data type except TEXT and BYTE.

The runtime system displays the question in the prompt area, waits for the user to enter a value, reads whatever value was entered until the user validates (for example with the Enter key), and stores this value in a response variable. The prompt dialog remains visible until the user enters a response.

The prompt finishes after ON IDLE, ON ACTION, or ON KEY block execution (to ensure backwards compatibility).

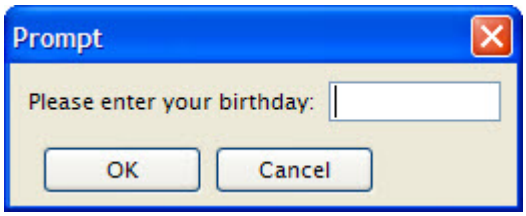
Prompt display in TUI mode

In TUI mode, the PROMPT question and input field is displayed in the prompt line of the current window, which is defined by the OPTIONS PROMPT LINE instruction or with the ATTRIBUTES clause of OPEN WINDOW. If the prompt line is not as wide as the prompt string, runtime error -1146 occurs.

Prompt display in GUI mode

In GUI mode, the PROMPT instruction opens a modal window with an OK and a Cancel button, and waits for input from the user.

Figure 72: PROMPT window



Syntax of PROMPT instruction

The PROMPT statement assigns a user-supplied value to a variable.

Syntax

```
PROMPT question
    [ ATTRIBUTES ( display-attribute [,...] ) ]
    FOR [CHAR[ACTER]] variable
    [ HELP number ]
    [ ATTRIBUTES ( control-attribute [,...] ) ]
```

```
[ dialog-control-block
   [...]
   END PROMPT ]
```

where *dialog-control-block* is one of:

```
{ ON IDLE seconds
| ON TIMER seconds
| ON ACTION action-name
| ON KEY ( key-name [,...] )
}
    statement
    [...]
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

where *control-attribute* is:

```
{ ACCEPT [ = boolean ]
| CANCEL [ = boolean ]
| CENTURY = "century-spec"
| FORMAT = "format-spec"
| PICTURE = "picture-spec"
| SHIFT = { "up" | "down" }
| HELP = help-number
| COUNT = row-count
| UNBUFFERED [ = boolean ]
| WITHOUT DEFAULTS [ = boolean ]
}
```

1. *question* is a string expression displayed as a message for the input of the value.
2. *variable* is the name of the variable that receives the data typed by the user.
3. The `FOR CHAR` clause exits the prompt statement when the first character has been typed.
4. *number* is the help message number to be displayed when the user presses the help key.
5. *key-name* is an hot-key identifier (such as `F11` or `Control-z`).
6. *action-name* identifies an action that can be executed by the user.
7. *seconds* is an integer literal or variable that defines a number of seconds.
8. *statement* is an instruction that is executed when the user presses the key defined by *key-name.*
9. *century-spec* is a string specifying the century input rule, like the `CENTURY` attribute.
10. *format-spec* is a string defining the display format for the prompt field, like the `FORMAT` attribute.
11. *picture-spec* is a string defining the input format for the prompt field, like the `PICTURE` attribute.

Using simple prompt inputs

To use simple prompt inputs, you must understand how they work and how to structure the code.

PROMPT programming steps

To use the `PROMPT` statement, you must:

1. Declare a program variable with the `DEFINE` statement.
2. Set the `INT_FLAG` variable to `FALSE`.
3. Define the `PROMPT` statement, with dialog control blocks to control the instruction. Use the `FOR CHAR` clause if a single character is to be entered.

4. After executing the PROMPT, check the INT_FLAG variable to determine whether the input was validated or canceled by the user.

PROMPT instruction configuration

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help while executing the instruction. The predefined *help* action is automatically created by the runtime system. You can bind action views to the *help* action.

The HELP clause overrides the HELP attribute.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the *accept* default action.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.

Default actions in PROMPT

When a PROMPT instruction executes, the runtime system creates a set of default actions.

According the invoked default action, field validation occurs and different PROMPT control blocks are executed.

This table lists the default actions created for this dialog:

Table 268: Default actions created for the PROMPT dialog

Default action	Description
accept	Validates the PROMPT dialog (validates field criteria) *Creation can be avoided with the* ACCEPT *attribute.*
cancel	Cancels the PROMPT dialog (no validation, int_flag is set) *Creation can be avoided with the* CANCEL *attribute.*
close	By default, cancels the PROMPT dialog (no validation, int_flag is set) Default action view is hidden. See Implementing the close action on page 1337.
help	Shows the help topic defined by the HELP clause. *Only created when a* HELP *clause is defined.*

Interaction blocks

ON ACTION block

You can use `ON ACTION` blocks to execute a sequence of instructions when the user raises a specific action. This is the preferred solution compared to `ON KEY` blocks, because `ON ACTION` blocks use abstract names to control user interaction.

> **Important:** The `PROMPT` instruction is automatically finished after `ON IDLE`, `ON ACTION`, `ON KEY` block execution.

ON IDLE block

The `ON IDLE` *`seconds`* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of `ON IDLE` must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

> **Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An `ON KEY (`*`key-name`*`)` block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The `ON KEY` block is supported for backward compatibility with TUI mode applications.

An `ON KEY` block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, `ON KEY(F5,F6)` creates two actions with the names `f5` and `f6`. Each action object will get an `ACCELERATORNAME` assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON TIMER block

The `ON TIMER` *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

Examples

Example 1: Simple PROMPT statements

```
MAIN
  DEFINE birth DATE
  DEFINE chkey CHAR(1)
  PROMPT "Please enter your birthday: " FOR birth
  DISPLAY "Your birthday is: " || birth
  PROMPT "Now press a key... " FOR CHAR chkey
  DISPLAY "You pressed: " || chkey
END MAIN
```

Example 2: Simple PROMPT with Interrupt Checking

```
MAIN
  DEFINE birth DATE
  LET INT_FLAG = FALSE
  PROMPT "Please enter your birthday: " FOR birth
  IF INT_FLAG THEN
    DISPLAY "Interrupt received."
  ELSE
    DISPLAY "Your birthday is: " || birth
  END IF
END MAIN
```

Example 3: PROMPT with ATTRIBUTES and ON ACTION handlers

```
MAIN
  DEFINE birth DATE
  LET birth = TODAY
  PROMPT "Please enter your birthday: " FOR birth
     ATTRIBUTES(WITHOUT DEFAULTS)
        ON ACTION action1
           DISPLAY "Action 1"
  END PROMPT
  DISPLAY "Your birthday is " || birth
END MAIN
```

Ring menus (MENU)

The MENU instruction implements a list of options the end user can choose from.

- Understanding ring menus on page 1048
- Syntax of the MENU instruction on page 1048
- MENU programming steps on page 1050
- Using ring menus on page 1050
  - Rendering modes of a menu on page 1050
  - Binding action views to menu options on page 1053
  - MENU instruction configuration on page 1053
  - Default actions in MENU on page 1054
  - MENU control blocks on page 1054
    - BEFORE MENU block on page 1054
  - MENU interaction blocks on page 1054
    - COMMAND [KEY()] "option" block on page 1054
    - COMMAND KEY() block on page 1055
    - ON ACTION block on page 1056
    - ON IDLE block on page 1046
- MENU control instructions on page 1057
  - SHOW/HIDE OPTION instruction on page 1057
  - EXIT MENU instruction on page 1058
  - CONTINUE MENU instruction on page 1058
- Examples on page 1059

Understanding ring menus

A *ring menu* defines a list of options that can trigger actions to execute associated program code. Ring menus are implemented with the MENU interactive instruction. A MENU block lists the possible actions that can be triggered in a given place in the program, with the associated program code to be executed.

```
MENU "Sample"
    COMMAND "Say hello"
       DISPLAY "Hello, world!"
    COMMAND "Exit"
       EXIT MENU
END MENU
```

A ring menu can only define a set of options for a given level of the program. You cannot define all menu options of your program in a single MENU instruction; you must implement nested menus.

The MENU instruction is mainly designed for text mode applications, displaying ring menus at the top of the screen. A typical TUI mode application starts with a global menu, defining general options to access subroutines, which in turn implement specific menus with database record handling options such as 'Append', 'Delete', 'Modify', and 'Search'. Ring menus can also be used in a GUI application, however, as this instruction does not handle form fields, other parts of the form are disabled during the menu dialog execution. In GUI applications, ring menus are typically used to open a modal window with Yes / No / Cancel options.

Syntax of the MENU instruction

The MENU instruction defines a set of options the end user can select to trigger actions in a program.

Syntax

```
MENU [title]
```

```
   [ ATTRIBUTES ( menu-attribute [,...] ) ]
   [ BEFORE MENU
       menu-statement
     [...]
   ]
   menu-option
   [...]
END MENU
```

where *menu-option* is one of:

```
{ COMMAND option-name
      [option-comment] [ HELP help-number ]
    menu-statement
    [...]
| COMMAND KEY ( key-name ) option-name
      [option-comment] [ HELP help-number ]
    menu-statement
    [...]
| COMMAND KEY ( key-name )
    menu-statement
    [...]
| ON ACTION action-name
             [ ATTRIBUTES ( action-attributes-menu ) ]
    menu-statement
    [...]
| ON IDLE seconds
    menu-statement
    [...]
| ON TIMER seconds
    menu-statement
    [...]
}
```

where *action-attributes-menu* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
| DISCLOSUREINDICATOR
    [,...] }
```

where *menu-statement* is:

```
{ statement
|  CONTINUE MENU
|  EXIT MENU
|  NEXT OPTION option
|  SHOW OPTION { ALL | option [,...] }
|  HIDE OPTION { ALL | option [,...] }
}
```

where *menu-attribute* is:

```
{ STYLE = { "default" | "popup" | "dialog" }
| COMMENT = "string"
| IMAGE = "string"
}
```

1. *title* is a string expression defining the title of the menu.
2. *menu-attribute* is an attribute that defines the behavior and presentation of the menu.
3. *key-name* is an hot-key identifier (like `F11` or `Control-z`).
4. *option-name* is a string expression defining the label of the menu option and identifying the action that can be executed by the user.
5. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
6. *help-number* is an integer that allows you to associate a help message number with the menu *option.*
7. *action-name* identifies an action that can be executed by the user.
8. *seconds* is an integer literal or variable that defines a number of seconds.
9. *action-name* identifies an action that can be executed by the user.
10. *action-attributes* are dialog-specific action attributes.

MENU programming steps

The following steps describe how to implement a `MENU` statement:

1. Create a `MENU` block with a title and write the end of the menu block with the `END MENU` keywords.
2. According to the type of menu rendering you need, add an `ATTRIBUTES` clause with the required `STYLE` attribute.
3. List all the options that you want to offer to the end user when the menu executes. Typical CRUD programs will implement "Append", "Modify", "Delete" operations for a given database application entity (customers, orders, items tables). Typical dialog box menus have "Yes" / "No" / "Cancel" options.
4. According to TUI or GUI mode, define action views (topmenu, toolbar or form buttons) for each menu action, and use either `COMMAND [KEY]` or `ON ACTION` clauses to define the menu options.
5. When the menu is not a popup or dialog menu, do not forget to implement an option to leave the menu with the `EXIT MENU` control instruction.
6. Implement the code to be executed in every option.

Using ring menus

To use ring menus, you must understand how they work and how to structure the code.

Rendering modes of a menu

When you add a style to a `MENU`'s attributes list, you define the look-and-feel of that menu and how that menu acts.

MENU rendering specification

The rendering mode of a `MENU` instruction can be controlled with the `STYLE` dialog attribute:

```
MENU "Test" ATTRIBUTES ( STYLE = "mode" )
   ...
END MENU
```

> **Note:** `MENU ... ATTRIBUTES(STYLE="mode")` is not a presentation style defined in a `4st` file: It defines a display mode, a rendering hint for front-ends.

The decoration of the different rendering modes of a `MENU` depends from the front-end type and the platform used. Consider testing the menu instruction with all front-ends that must be supported for end users.

Default MENU rendering

By default, if no `STYLE` attribute is used in the `MENU` instruction, each menu option will be displayed as a push button in a dedicated area of the current window, depending on the front end. This dedicated area is called the action frame.

Note that when an explicit action view (for ex, a `BUTTON` in form layout) is associated with a menu option, the default button will not appear in the action frame area.

The default rendering of a `MENU`, including the position of the action frame in the window, can be controlled with window presentation style attributes.

```
MAIN
  MENU "File"
    COMMAND "New"
      DISPLAY "New"
    COMMAND "Open"
      DISPLAY "Open"
    COMMAND "Save"
      DISPLAY "Save"
    COMMAND "Import"
      DISPLAY "Import"
    COMMAND "Quit"
      EXIT MENU
  END MENU
END MAIN
```

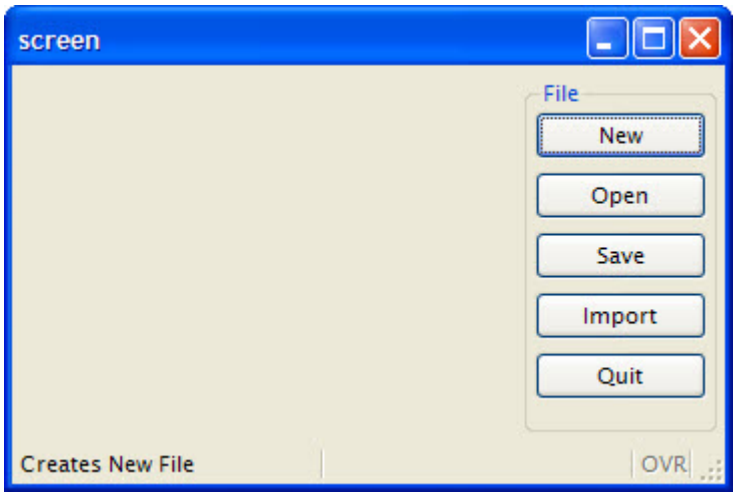


Figure 73: Default rendering of MENU with the Genero Desktop Client

Modal dialog MENU rendering

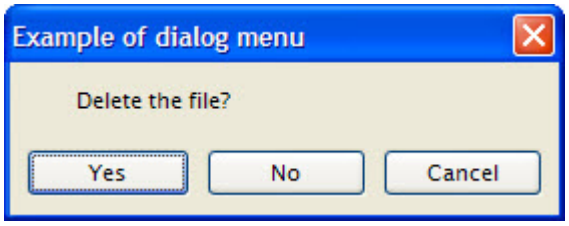
Menus can be rendered in a modal dialog window by specifying the `STYLE="dialog"` attribute in the `MENU` instruction.

```
MAIN
  MENU "Example of dialog menu"
    ATTRIBUTES ( STYLE="dialog", COMMENT="Delete the file?" )
    COMMAND "Yes"
      DISPLAY "Yes"
    COMMAND "No"
      DISPLAY "No"
    COMMAND "Cancel"
      DISPLAY "Cancel"
  END MENU
END MAIN
```

When the user clicks on an option, the `MENU` instruction automatically exits and the modal dialog window closes. There is no need for an `EXIT MENU` command.

With `STYLE="dialog"`, when the user clicks on an option, the `MENU` instruction automatically exits and the popup menu closes. There is no need for an `EXIT MENU` command.

Figure 74: MENU displayed as a modal dialog with the Genero Desktop Client



Popup MENU rendering

Menus can also be displayed as popup choice lists, when the `STYLE="popup"` attribute is used in the `MENU` instruction.

```
MAIN
   DEFINE r INTEGER
   MENU "test"
     COMMAND "popup"
        DISPLAY popup()
     COMMAND "quit"
        EXIT MENU
   END MENU
END MAIN

FUNCTION popup()
   DEFINE r INTEGER
   LET r = -1
   MENU "unused" ATTRIBUTES ( STYLE="popup" )
     COMMAND "Copy all"
        LET r = 1
     COMMAND "Copy current"
        LET r = 2
     COMMAND "Paste all"
        LET r = 3
     COMMAND "Paste current"
        LET r = 4
   END MENU
   RETURN r
END FUNCTION
```

With `STYLE="popup"`, when the user clicks on an option, the `MENU` instruction automatically exits and the popup menu closes. There is no need for an `EXIT MENU` command.

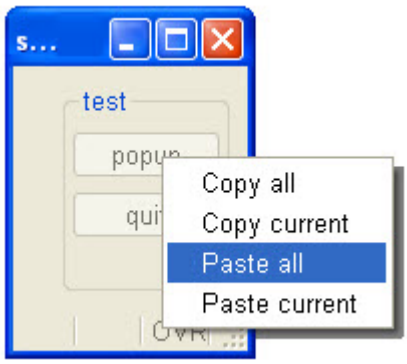


Figure 75: MENU displayed as popup list with the Genero Desktop Client

MENU rendering on mobile platforms

On mobile devices, the rendering of the MENU dialog depends on whether or not the current window has a form.

If a MENU is active and the current Window has no form, then the MENU is shown as a list of actions.

If a MENU is active and has a FORM, then the menu actions are rendered like all other dialog actions.

Binding action views to menu options

A MENU statement is a controller for user actions, defining action handlers triggered by action views. Actions views in the form file (i.e. toolbar buttons, topmenu items or push buttons) are bound to menu options by name. For example, if a MENU instruction defines `ON ACTION sendmail`, a form BUTTON with the name "sendmail" will be attached to that action handler.

When binding action views to menu option clauses, the action name is case sensitive. The compiler converts COMMAND labels and `ON ACTION` identifiers to lowercase to create the action name. It is recommended that you use all lowercase letters when defining the action name for action views and menu options.

Menu options can also be defined with the COMMAND clause. Unlike `ON ACTION`, the COMMAND clause takes a string literal as argument, that defines both the action name and the default text to be displayed in the default action view. For example, `COMMAND "Help"` will define the action name help and the default button text "Help". Action views must be bound with the action name in lower case (`help`).

When the menu is rendered as a popup of dialog box, no explicit action views need to be defined, default action views will be created and will get the decoration specified in action defaults.

MENU instruction configuration

The rendering and behavior of a MENU instruction can be configured with the ATTRIBUTES clause:

```
MENU "Question"
     ATTRIBUTES (
         STYLE="dialog",
         COMMENT="Do you want to commit your changes?"
      )
```

When the STYLE instruction attribute is set to '`default`' or when you do not specify the menu type, the runtime system generates a default decoration as a set of buttons in a specific area of the current window.

When the STYLE attribute is set to '`dialog`', the menu options appear as buttons at the bottom in a temporary modal window, in which you can define the message and the icon with the COMMENT and IMAGE attributes.

When the STYLE is set to '`popup`', the menu appears as a popup menu (contextual menu).

If the menu is a "dialog" or "popup", the dialog is automatically exited after any action clause such as ON ACTION, COMMAND or ON IDLE.

Default actions in MENU

When an MENU instruction executes, the runtime system creates a set of default actions.

Table 269: Default actions created for the MENU instruction

Default action	Control Block execution order
close	Created to execute COMMAND KEY(INTERRUPT) if used (can be overwritten with ON ACTION close) Default action view is hidden. See Implementing the close action on page 1337.
help	Shows the help topic defined by the HELP clause. Default action view is hidden.

Window close events can be trapped with COMMAND KEY(INTERRUPT) clause.

MENU control blocks
BEFORE MENU block

If the MENU block contains a BEFORE MENU clause, statements within this clause will be executed before the menu dialog starts.

This block is typically used to hide or disable some menu options according to the current context of the program. For example, when the current user is not allowed to create new records, the menu options can be disabled as follows:

```
MENU "Orders"
   BEFORE MENU
      CALL DIALOG.setActionActive("append", can_user_append() )
      ...
   COMMAND "Append" -- creates "append" action (lowercase)
      ...
   ...
END MENU
```

In TUI mode, the menu options can also be disabled, but they will still be displayed on the screen. The end user will see the option, but cannot select it. In this case it's more convenient to hide the option to the end user with the DIALOG.setActionHidden() method, instead of disabling the action.

MENU interaction blocks
COMMAND [KEY()] "option" block

The COMMAND [KEY(*key-name*)] "*option-name*" clause defines a menu action handler with a set of instructions to be executed when an action is invoked. The option text (*option-name*), converted to lowercase letters, defines the name of the action.

For example, when defining:

```
COMMAND "Hello"
```

The name of the action will be "hello" (not "Hello" with a capital H).

When used with the KEY() clause, the command specifies both accelerator keys and an option text. For backward compatibility, a coma-separated key list is supported in the KEY() specification. Consider using a single key for new developments, or prefer accelerator definition with action defaults.

Action defaults will be applied by using the action name defined by the option text (converted to lower case).

Explicit action views defined in the form (`BUTTON` in layout, `TOPMENU` or `TOOLBAR` items) will get all action defaults associated to the menu command, while default action views (i.e. buttons in the action frame) will be decorated with the menu option text and comment specified in the program (i.e. the `TEXT` and `COMMENT` attributes of the corresponding action defaults entry are not used for the default action views), however, other attributes such as the `IMAGE` will also be applied to default action views.

For example, when defining:

```
COMMAND "Hello" "This is the Hello option"
```

The name of the action will become "`hello`", the default action view button text will be "`Hello`", and the button hint will be "`This is the Hello option`", even if an action default defines a different text or comment for the "`hello`" action. If the corresponding action default defines a `IMAGE` icon, it will display in the default action view button.

The `KEY()` clause can specify up to four accelerator attributes for the action. The keys defined in the program will take precedence over accelerators defined with action defaults.

The first letter of the display text of a `COMMAND` menu clause can be used as default accelerator. When this first letter is not used by other menu option labels, pressing the key corresponding to that letter will execute that action. When the first letter is also used in other menu options, pressing the key will toggle the focus between all default action views that share the same letter. For example:

```
MENU
  COMMAND "Start"
    DISPLAY "Start"
  COMMAND "Stop"
    DISPLAY "Stop"
  COMMAND "Quit"
    EXIT MENU
END MENU
```

In this example, when pressing S on the keyboard, the focus will toggle between "Start" and "Stop"buttons, and the current option can be selected with the Return or Space key. When pressing Q, the "Quit" action will be fired.

To write abstract code without decoration in your programs, use the `ON ACTION` clause instead of `COMMAND [KEY]`, except if the action view must get the focus.

Note that if you use an ampersand (&) in the command name, some front-ends consider the letter following & as an Alt-key accelerator, and the letter will be underscored. However the ampersand forms part of the action name. For example, `COMMAND "&Save"` will create an action with the name "`&save`".

In TUI mode, actions created with `COMMAND [KEY]` do not get accelerators from action defaults; Only actions defined with `ON ACTION` will get accelerators of action defaults.

COMMAND KEY() block

The `COMMAND KEY(`*`key-name`*`)` block (without an option text) defines a menu action handler with a set of instructions to be executed when an action is invoked. The `KEY()` clause defines one or several accelerator keys separated by a comma. The specified key name must be one of the virtual keys.

For backward compatibility, a coma-separated key list is supported in the `KEY()` specification. Consider using a single key for new developments, or prefer accelerator definition with action defaults.

While a `COMMAND KEY(`*`key-name`*`) "`*`option-name`*`"` (with option text) defines the name of the action with the option text (converted to lowercase), a `COMMAND KEY(`*`key-name`*`)` (without option text), defines the action name from the last key in the `KEY()` list, converted to lowercase letters. For example, with `COMMAND KEY(F10,F12,Control-Z)`, the name of the action will be "`control-z`".

Action defaults will be applied by using the key name of the KEY() clause. With a list of keys, the last key name will be used to apply action defaults, because it defines the action name.

The KEY() clause can specify up to four accelerator attributes for the action. The keys defined in the program will take precedence over accelerators defined with action defaults.

By default, COMMAND KEY(*key-name*) actions are not decorated with a default action (i.e. a button in the action frame will not appear for these actions). However, by defining the text attribute within action defaults, the default action view button will be visible. This allows you to decorate existing COMMAND KEY(*key-name*) clauses with graphical buttons without changing the program code.

To write abstract code without decoration in your programs, use the ON ACTION clause instead of COMMAND [KEY], except if the action view must get the focus.

In TUI mode, actions created with COMMAND [KEY] do not get accelerators from action defaults; Only actions defined with ON ACTION will get accelerators of action defaults.

ON ACTION block

The ON ACTION *action-name* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
ON ACTION action-name
   instruction
   ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
ON ACTION zoom
   CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as INPUT, INPUT ARRAY and CONSTRUCT, if an action is specific to a field, add the INFIELD clause to have the action automatically enabled when the corresponding field gets the focus:

```
ON ACTION zoom INFIELD cust_city
   CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the ON ACTION handler needs to define its own attributes at the program level. This can be done by adding the ATTRIBUTES() clause of ON ACTION:

```
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
   CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The ON IDLE *seconds* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of ON IDLE must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

> **Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON TIMER block

The `ON TIMER` *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

MENU control instructions
SHOW/HIDE OPTION instruction
The `HIDE OPTION` and `SHOW OPTION` to hide or show `MENU` options.

Syntax:

```
{ HIDE | SHOW } OPTION
    { ALL
    | option-name [,...]
    }
```

Usage

The `SHOW OPTION` instruction will show/enable action views corresponding the listed menu options. The default action views (buttons in action frame) are made visible and the explicit action views (buttons in form) are enabled. The `HIDE OPTION` instruction will hide default action views and disable explicit action views.

Use the `ALL` keyword reference all menu options. In a menu that contains many options, you typically do a `HIDE OPTIONS ALL` followed by `HIDE OPTION` to show a subset of the menu options.

The `SHOW OPTION` and `HIDE OPTION` instructions are provided for backward compatibility. To hide and show default action views, use the `DIALOG.setActionHidden()` method instead. In GUI applications, you should rather disable actions, instead of hiding them to the end user.

Example

```
MENU "Customers"
   BEFORE MENU
      HIDE OPTION ALL
      SHOW OPTION "Add", "Exit"
   ...
```

EXIT MENU instruction

`EXIT MENU` terminates the execution of a `MENU` block.

Syntax

```
EXIT MENU
```

Usage

`EXIT MENU` statement terminates the `MENU` block an continues the program flow with the statement after the menu block.

Example

```
MENU "Stock"
  ...
  COMMAND "Exit"
     EXIT MENU
END MENU
```

CONTINUE MENU instruction

`CONTINUE MENU` resumes the execution of a `MENU` block.

Syntax

```
CONTINUE MENU
```

Usage

The `CONTINUE MENU` ignores the remaining instructions in the current program section of a `MENU` block, re-displays the menu options and gives the control back to the user to select a new menu option.

The statements following the `CONTINUE MENU` instruction are skipped.

Example

```
MENU "Stock"
  ...
  COMMAND "Exit"
    IF question("Exit the program?")==FALSE THEN
       CONTINUE MENU
    END IF
    CALL commit_changes()
    EXIT MENU
```

```
END MENU
```

Examples

Example 1: MENU with abstract action options

```
MENU
  ON ACTION new
      CALL newFile()
  ON ACTION open
      CALL openFile()
  ON ACTION save
      CALL saveFile()
  ON ACTION import
      LOAD FROM "infile.dat" INSERT INTO table
  ON ACTION quit
      EXIT PROGRAM
END MENU
```

Example 2: MENU with text-mode options

```
MENU "File"
  COMMAND KEY ( CONTROL-N ) "New" "Creates New File" HELP 101
    CALL newFile()
  COMMAND KEY ( CONTROL-O ) "Open" "Open existing File" HELP 102
    CALL openFile()
  COMMAND KEY ( CONTROL-S ) "Save" "Save Current File" HELP 103
    CALL saveFile()
  COMMAND "Import"
    LOAD FROM "infile.dat" INSERT INTO table
  COMMAND KEY ( CONTROL-Q ) "Quit" "Quit Program" HELP 201
    EXIT PROGRAM
END MENU
```

Example 3: MENU with STYLE="dialog"

The next code example implements typical message box utility functions implemented with `MENU` dialogs:

```
FUNCTION mbox_ync(title,msg)
    DEFINE title, msg STRING
    DEFINE res SMALLINT
    MENU title ATTRIBUTES(STYLE="dialog",COMMENT=msg)
        ON ACTION yes     LET res = 1
        ON ACTION no      LET res = 0
        ON ACTION cancel  LET res = -1
    END MENU
    RETURN res
END FUNCTION

FUNCTION mbox_yn(title,msg)
    DEFINE title, msg STRING
    DEFINE res BOOLEAN
    MENU title ATTRIBUTES(STYLE="dialog",COMMENT=msg)
        ON ACTION yes LET res = TRUE
        ON ACTION no  LET res = FALSE
    END MENU
    RETURN res
END FUNCTION

FUNCTION mbox_ok(title,msg)
    DEFINE title, msg STRING
    MENU title ATTRIBUTES(STYLE="dialog",COMMENT=msg)
```

```
        ON ACTION accept
    END MENU
END FUNCTION
```

Record input (INPUT)

The `INPUT` instruction provides single record input control in an application form.

- Understanding the INPUT instruction on page 1060
- Syntax of the INPUT instruction on page 1061
- INPUT programming steps on page 1062
- Using simple record inputs on page 1063
  - Variable binding in INPUT on page 1063
  - INPUT instruction configuration on page 1065
  - Default actions in INPUT on page 1066
  - INPUT control blocks on page 1066
    - INPUT control blocks execution order on page 1066
    - BEFORE INPUT block on page 1067
    - AFTER INPUT block on page 1068
    - BEFORE FIELD block on page 1069
    - ON CHANGE block on page 1069
    - AFTER FIELD block on page 1070
  - INPUT interaction blocks on page 1070
    - ON ACTION block on page 1056
    - ON IDLE block on page 1046
    - ON KEY block on page 1046
  - INPUT control instructions on page 1072
    - NEXT FIELD instruction on page 1121
    - ACCEPT INPUT instruction on page 1072
    - CONTINUE INPUT instruction on page 1073
    - EXIT INPUT instruction on page 1073
- Examples on page 1073
  - Example 1: INPUT with binding by field position on page 1073
  - Example 2: INPUT with binding by field name on page 1074

Understanding the INPUT instruction

The `INPUT` statement binds program variables to screen-records in forms for data entry in form fields. The `INPUT` statement uses the current form in the current window. Before executing the `INPUT` statement, record data must be fetched from the database table into the program variables using by the input statement.

During the `INPUT` statement execution, the user can edit the record fields, while the program controls the behavior of the instruction with control blocks.

To terminate the `INPUT` execution, the user can validate (or cancel) the dialog to commit (or invalidate) the modifications made in the record.

When the statement completes execution, the form is deactivated. After the user terminates the input (for example, with the "accept" key), the program must test the `INT_FLAG` variable to check if the dialog was validated (or canceled), and then can use the `INSERT` or `UPDATE` SQL statements to modify the appropriate database tables.

Syntax of the INPUT instruction

The `INPUT` statement supports data entry into the fields of the current form.

Syntax

```
INPUT { BY NAME { variable | record.* } [,...]
              [ WITHOUT DEFAULTS ]
      | variable | record.* } [,...]
              [ WITHOUT DEFAULTS ]
              FROM field-list
      }
  [ ATTRIBUTES (
         { display-attribute
         | control-attribute
         } [,...] ) ]
  [ HELP help-number ]
  [ dialog-control-block
    [...]
END INPUT ]
```

where *dialog-control-block* is one of:

```
{ BEFORE INPUT
| AFTER INPUT
| BEFORE FIELD field-spec [,...]
| AFTER FIELD field-spec [,...]
| ON CHANGE field-spec [,...]
| ON IDLE seconds
| ON TIMER seconds
| ON ACTION action-name
             [ INFIELD field-spec ]
             [ ATTRIBUTES ( action-attributes-input ) ]
| ON KEY ( key-name [,...] )
}
    dialog-statement
    [...]
```

where *action-attributes-input* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| VALIDATE = NO
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT INPUT
| CONTINUE INPUT
| EXIT INPUT
| NEXT FIELD
   { CURRENT
   | NEXT
   | PREVIOUS
   | field-spec
   }
}
```

where *field-list* defines a list of fields with one or more of:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *field-spec* identifies a unique field with one of:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
}
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

where *control-attribute* is:

```
{ ACCEPT [ = boolean ]
| CANCEL [ = boolean ]
| FIELD ORDER FORM
| HELP = help-number
| NAME = "dialog-name"
| UNBUFFERED [ = boolean ]
| WITHOUT DEFAULTS [ = boolean ]
}
```

1. *variable* is a program variable that will be filled by the `INPUT` statement.
2. *record.** is a record variable that will be filled by the `INPUT` statement.
3. *help-number* is an integer that allows you to associate a help message number with the instruction.
4. *field-name* is the identifier of a field of the current form.
5. *table-name* is the identifier of a database table of the current form.
6. *screen-record* is the identifier of a screen record of the current form.
7. *screen-array* is the screen array that will be used in the form.
8. *line* is a screen array line in the form.
9. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
10. *dialog-name* is the identifier of the dialog.
11. *seconds* is an integer literal or variable that defines a number of seconds.
12. *action-name* identifies an action that can be executed by the user.
13. *statement* is any instruction supported by the language.
14. *boolean* is a boolean expression evaluated when the dialog starts.
15. *action-attributes* are dialog-specific action attributes.

INPUT programming steps

The following steps describe how to use the `INPUT` statement:

1. Create a form specification file, with an optional screen record.

The screen record identifies the presentation elements to be used by the runtime system to display the records. if you omit the declaration of the screen record in the form file, the runtime system will use the default screen records created by the form compiler for each table listed in the `TABLES` section and for the `FORMONLY` pseudo-table.

2. Make sure that the program controls interruption handling with `DEFER INTERRUPT`, to manage the validation/cancellation of the interactive dialog.
3. Define a program record with the `DEFINE` instruction.

   The members of the program record must correspond to the elements of the screen record, by number and data types.
4. Open and display the form, using an `OPEN WINDOW` with the `WITH FORM` clause or the `OPEN FORM` / `DISPLAY FORM` instructions.
5. If needed, fill the program record with data, for example with a result set cursor.
6. Set the `INT_FLAG` variable to `FALSE`.
7. Write the `INPUT` statement to handle data input.
8. Inside the `INPUT` statement, control the behavior of the instruction with `BEFORE INPUT`, `BEFORE FIELD`, `AFTER FIELD`, `AFTER INPUT` and `ON ACTION` blocks.
9. After the interaction statement block, test the `INT_FLAG` predefined variable to check if the dialog was canceled (`INT_FLAG=TRUE`) or validated (`INT_FLAG=FALSE`).

   If the `INT_FLAG` variable is `TRUE`, you should reset it to `FALSE` to not disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.

Using simple record inputs

To use simple record inputs, you must understand how they work and how to structure the code.

Variable binding in INPUT

The `INPUT` instruction binds program variables (typically, members of a `RECORD`) are bound to the fields of a screen record of the current form, and synchronizes the data between field input buffers and program variables.

Binding variables and fields by name

The `INPUT BY NAME` *`variable-list`* instruction implicitly binds the fields to the program variables that have the same identifiers as the form field names. The program variables are typically defined within a record declared with a `LIKE table.*` based a database schema, to get the same names as the form fields defined with database column references. The runtime system ignores any record name prefix when making the match, only record member names matter. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exception.

```
SCHEMA stock
DEFINE custrec RECORD LIKE customer.*
...
INPUT BY NAME custrec.*
  ...
END INPUT
```

Binding variables and fields by position

The `INPUT` *`variable-list`* `FROM` *`field-list`* clause explicitly binds the variables to form fields by position. The form can include other fields that are not part of the specified variable list, but the number of variables or record members must equal the number of form fields listed in the `FROM` clause. Each variable must be of the same (or a compatible) data type as the corresponding form field. When the user enters

data, the runtime system checks the entered value against the data type of the variable, not the data type of the form field.

```
SCHEMA stock
DEFINE custrec RECORD LIKE customer.*,
       comment VARCHAR(100)
...
INPUT custrec.*, comment FROM sr_cust.*, cmt
  ...
END INPUT
```

When using the FROM clause with a screen record followed by a .* (dot star), keep in mind that program variables are bound to screen record fields by position, so you must make sure that the program variables are defined (or listed) in the same order as the screen array fields.

Serial column support

The program variables can be of any data type: The runtime system will adapt input and display rules to the variable type. If a variable is declared with the LIKE clause and uses a column defined as SERIAL / SERIAL8 / BIGSERIAL, the runtime system will treat the field as if it was defined with the NOENTRY attribute in the form file: Since values of serial columns are automatically generated by the database server, no user input is required for such fields.

The UNBUFFERED mode

The variables act as data model to display data or to get user input through the INPUT instruction. Always use the variables if you want to change some field values by program. When using the UNBUFFERED attribute, the instruction is sensitive to program variable changes: If you need to display new data during the INPUT execution, just assign the values to the program variables; the runtime system will automatically display the values to the screen:

```
INPUT p_items.* FROM s_items.* ATTRIBUTES ( UNBUFFERED )
  ON CHANGE code
    IF p_items.code = "A34" THEN
      LET p_items.desc = "Item A34"
    END IF
END INPUT
```

Handling default field values

When the INPUT instruction executes, any column default values are displayed in the screen fields, unless you specify the WITHOUT DEFAULTS keywords. The column default values are specified in the form specification file with the DEFAULT attribute, or in the database schema files.

If you specify the WITHOUT DEFAULTS option, however, the form fields display the current values of the variables when the INPUT statement begins. This option is available with both the BY NAME and the FROM binding clauses.

```
LET p_items.code = "A34"
INPUT p_items.* FROM s_items.* WITHOUT DEFAULTS
  BEFORE INPUT
    MESSAGE "You should see A34 in field 'code'..."
END INPUT
```

Using PHANTOM fields

If the program record has the same structure as a database table (this is the case when the record is defined with a LIKE clause), you may not want to display/use some of the columns. You can achieve this

by used PHANTOM fields in the screen record definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

INPUT instruction configuration

This section describes the options that can be specified in the ATTRIBUTES clause of the INPUT instruction. The options of the ATTRIBUTES clause override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. With the INPUT statement, the INVISIBLE attribute is ignored.

NAME option

The NAME attribute can be used to name the INPUT dialog. This is especially used to identify actions of the dialog.

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help while the focus is in any field used by the instruction. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

WITHOUT DEFAULTS option

Indicates if the fields controlled by INPUT must be filled (FALSE) or not (TRUE) with the column default values defined in the form specification file or the database schema files. The runtime system assumes that the field satisfies the REQUIRED attribute when WITHOUT DEFAULTS is used. If the WITHOUT DEFAULT option is not used, all fields defined with the REQUIRED attribute must be visited and modified. Fields not defined as NOT NULL can be left empty.

FIELD ORDER FORM option

By default, the tabbing order is defined by the variable binding list in the instruction description. You can control the tabbing order by using the FIELD ORDER FORM attribute: When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form fields. If this attribute is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.

The OPTIONS instruction can also change the behavior of the INPUT instruction, with the INPUT WRAP or FIELD ORDER FORM options.

UNBUFFERED option

Indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the traditional "buffered" mode.

When using the traditional "buffered" mode, program variable changes are not automatically displayed to form fields; You need to execute a DISPLAY TO or DISPLAY BY NAME. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of the current field is to use GET_FLDBUF().

If the "unbuffered" mode is used, program variables and form fields are automatically synchronized. You don't need to display explicitly values with a DISPLAY TO or DISPLAY BY NAME. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the accept default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.

Default actions in INPUT

When an INPUT instruction executes, the runtime system creates a set of default actions.

According the invoked default action, field validation occurs and different INPUT control blocks are executed.

This table lists the default actions created for this dialog:

Table 270: Default actions created for the INPUT dialog

Default action	Description
accept	Validates the INPUT dialog (validates fields and leaves the dialog) *Creation can be avoided with* ACCEPT *attribute.*
cancel	Cancels the INPUT dialog (no validation, INT_FLAG is set to TRUE) *Creation can be avoided with* CANCEL *attribute.*
close	By default, cancels the INPUT dialog (no validation, INT_FLAG is set to TRUE) Default action view is hidden. See Implementing the close action on page 1337.
help	Shows the help topic defined by the HELP clause. *Only created when a* HELP *clause is defined.*

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

```
INPUT BY NAME field1 ATTRIBUTES ( CANCEL=FALSE )
  ...
```

INPUT control blocks

INPUT control blocks execution order

This table shows the order in which the runtime system executes the control blocks in the INPUT instruction, according to the user action:

Table 271: Control Block Execution Order for INPUT

Context / User action	Control Block execution order
Entering the dialog	**1.** BEFORE INPUT **2.** BEFORE FIELD (first field)

Context / User action	Control Block execution order
Moving from field A to field B	**1.** `ON CHANGE` (if value has changed for field A) **2.** `AFTER FIELD` (for field A) **3.** `BEFORE FIELD` (for field B)
Changing the value of a field with a specific field like checkbox	**1.** `ON CHANGE`
Validating the dialog	**1.** `ON CHANGE` (if value has changed in current field) **2.** `AFTER FIELD` **3.** `AFTER INPUT`
Canceling the dialog	**1.** `AFTER INPUT`

BEFORE INPUT block

BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular `INPUT`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `BEFORE INPUT` is only executed once when the dialog is started.

The `BEFORE INPUT` block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
INPUT BY NAME cust_rec.* ...
   BEFORE INPUT
       MESSAGE "Input customer information"
       CALL DIALOG.setActionActive("check_info", is_super_user() )
       CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
   ...
```

The fields are initialized with the defaults values before the `BEFORE INPUT` block is executed. When the `INPUT` instruction uses the `WITHOUT DEFAULTS` option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the `DEFAULT` attributes of the form fields are used.

Use the `NEXT FIELD` control instruction in the `BEFORE INPUT` block, to jump to a specific field when the dialog starts.

BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE INPUT` block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not invoke the `BEFORE INPUT` block.

`BEFORE INPUT` is executed after the `BEFORE DIALOG` block andbefore the `BEFORE ROW`, `BEFORE FIELD` blocks.

In this example, the `BEFORE INPUT` block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
   BEFORE INPUT
       CALL DIALOG.setActionActive("validate_order", TRUE)
```

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular `INPUT`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `AFTER INPUT` is only executed once when dialog ends.

The `AFTER INPUT` block is executed after the user has validated or canceled the `INPUT` or `INPUT ARRAY` dialog with the accept or cancel default actions, or when the `ACCEPT INPUT` instruction is executed.

The `AFTER INPUT` block is not executed when the `EXIT INPUT` instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules depending from several fields. If the values entered by the user do not satisfy the constraints, use the `NEXT FIELD` instruction to force the dialog to continue. The `CONTINUE INPUT` instruction can be used instead of `NEXT FIELD`, when no particular field has to be select.

Before checking the validation rules, make sure that the `INT_FLAG` variable is `FALSE`: in case if the user cancels the dialog, the validation rules must be skipped.

```
INPUT BY NAME cust_rec.*
   WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
  ...

  AFTER INPUT
    IF NOT INT_FLAG THEN
       IF cust_rec.cust_address IS NOT NULL
          AND cust_rec.cust_zipcode IS NULL THEN
          ERROR "Address is incomplete, enter a zipcode."
          NEXT FIELD zipcode
       END IF
    END IF
END INPUT
```

To limit the validation to fields that have been modified by the end user, you can call the `FIELD_TOUCHED()` function or the `DIALOG.getFieldTouched()` method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER INPUT` block is executed when the focus is lost by a group of fields driven by an `INPUT` or `INPUT ARRAY` sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not invoke the `AFTER INPUT` block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

`AFTER INPUT` is executed after the `AFTER FIELD`, `AFTER ROW` blocks and before the `AFTER DIALOG` block.

Executing a `NEXT FIELD` in the `AFTER INPUT` control block will keep the focus in the group of fields. Within an `INPUT ARRAY` sub-dialog, `NEXT FIELD` will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the `AFTER INPUT` block is used to validate data and disable an action that can only be used in the current group:

```
INPUT BY NAME p_order.*
   AFTER INPUT
      IF NOT check_order_data(DIALOG) THEN
```

```
        NEXT FIELD CURRENT
      END IF
      CALL DIALOG.setFieldActive("validate_order", FALSE)
```

BEFORE FIELD block

For fields controlled by an INPUT, INPUT ARRAY or by a CONSTRUCT instructions, the BEFORE FIELD block is executed every time the cursor enters into the specified field.

For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
   BEFORE FIELD cust_status
     LET p_cust.cust_comment = NULL
     MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing the first time with a specific form, the BEFORE FIELD block might be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column would have been done in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

ON CHANGE block

The ON CHANGE block can be used to detect that a field changed by user input. The ON CHANGE block is executed if the value has changed since the field got the focus and if the modification flag is set. The ON CHANGE block can only be used for fields controlled by an INPUT or INPUT ARRAY dialog, it is not available in CONSTRUCT.

For editable fields defined as EDIT, TEXTEDIT or BUTTONEDIT, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and if the modification flag is set for the field. You leave the field when you validate the dialog, when you move to another field, or when you move to another row in an INPUT ARRAY. However, if the text edit field is defined with the COMPLETER attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in

a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

```
   ON CHANGE order_checked -- Defined as CHECKBOX
      CALL setup_dialog(DIALOG)
```

If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing the value of the current field by program in an ON ACTION block, the ON CHANGE block will be executed when leaving the field if the value is different from the reference value and if the modification flag is set (after previous user input or when the touched flag has been changed by program).

When using the NEXT FIELD instruction, the comparison value is reassigned as if the user had leaved and reentered the field. Therefore, when using NEXT FIELD in ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be invoked again if the value is different from the reference value. This denies to do field validation in ON CHANGE blocks: you must do validations in AFTER FIELD blocks and/or AFTER INPUT blocks.

AFTER FIELD block

In dialog parts driven by a simple INPUT, INPUT ARRAY or by a CONSTRUCT sub-dialog, the AFTER FIELD block is executed every time the focus leaves the specified field. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, based on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT or ACCEPT CONTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
   AFTER FIELD item_quantity
     IF p_item.item_quantity <= 0 THEN
        ERROR "Item quantity cannot be negative or zero"
        LET p_item.item_quantity = 0
        NEXT FIELD item_quantity
     END IF
```

INPUT interaction blocks
ON ACTION block

The ON ACTION *action-name* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
  ON ACTION action-name
     instruction
```

```
      ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
   ON ACTION zoom
      CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as `INPUT`, `INPUT ARRAY` and `CONSTRUCT`, if an action is specific to a field, add the `INFIELD` clause to have the action automatically enabled when the corresponding field gets the focus:

```
   ON ACTION zoom INFIELD cust_city
      CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the `ON ACTION` handler needs to define its own attributes at the program level. This can be done by adding the `ATTRIBUTES()` clause of `ON ACTION`:

```
   ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
      CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The `ON IDLE` *`seconds`* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of `ON IDLE` must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

> **Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An `ON KEY (`*`key-name`*`)` block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The `ON KEY` block is supported for backward compatibility with TUI mode applications.

An `ON KEY` block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, `ON KEY(F5,F6)` creates two actions with the

names `f5` and `f6`. Each action object will get an `ACCELERATORNAME` assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON TIMER block

The `ON TIMER` *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

INPUT control instructions

ACCEPT INPUT instruction

The `ACCEPT INPUT` instruction validates the `INPUT` instruction and exits the dialog block if no error is raised.

The `AFTER FIELD`, `ON CHANGE`, etc. control blocks will be executed.

The statements after the `ACCEPT INPUT` instruction will not be executed.

```
INPUT BY NAME cust_rec.*
  ...
  ON ACTION process_order
    CALL set_missing_defaults()
    ACCEPT INPUT
  ...
END INPUT
```

The `INPUT` instruction creates the default accept action to let the user validate the dialog. The `ACCEPT INPUT` instruction should only be used in specific cases when the default accept action is not appropriated.

CONTINUE INPUT instruction

`CONTINUE INPUT` skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog.

If this instruction is called in a control block that is not `AFTER INPUT`, further control blocks might be executed according to the context. `CONTINUE INPUT` instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER INPUT`, the focus returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. In this case the `BEFORE FIELD` of the current field will be invoked.

As alternative, use the `NEXT FIELD` control instruction to give the focus to a specific field and force the dialog to continue. However, unlike `CONTINUE INPUT`, the `NEXT FIELD` instruction will skip the further control blocks that are normally executed.

EXIT INPUT instruction

The `EXIT INPUT` instruction terminates the `INPUT` instruction and resumes the program execution at the instruction following the `INPUT` block.

Performing an `EXIT INPUT` instruction during a dialog is equivalent to cancel the dialog: No field validation will occur, and the `AFTER FIELD` or `AFTER INPUT` blocks will not be executed. The dialog is exited immediately. However, `INT_FLAG` will not be set to `TRUE` as when the cancel action is fired.

CLEAR instruction in dialogs

The `CLEAR` *`field-list`* and `CLEAR SCREEN ARRAY` *`screen-array.*`* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. `CLEAR` can be used outside any dialog instruction, such as the `DISPLAY BY NAME / TO` instructions.

When a dialog is configured with the `UNBUFFERED` mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with `DISPLAY` or `CLEAR` instruction in an `UNBUFFERED` dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an `UNBUFEFERED` dialog, just set to `NULL` the variables bound to the dialog. However, when using a `CONSTRUCT`, no program variables are associated to the dialog and no `UNBUFFERED` concept exits, and the `CLEAR` or `DISPLAY TO / BY NAME` instructions are the only way to modify the `CONSTRUCT` fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using `TABLE` containers, you can therefore use the `CLEAR SCREEN ARRAY` instruction to clear all rows of a list.

Examples
Example 1: INPUT with binding by field position

Form definition file (form1.per):

```
SCHEMA office

LAYOUT
GRID
{
  Customer id: [f001    ]
  First Name : [f002                    ]
  Last Name  : [f003                    ]
}
END
END

TABLES
  customer
```

```
END

ATTRIBUTES
  f001 = customer.id;
  f002 = customer.fname;
  f003 = customer.lname, UPSHIFT;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust(customer.*);
END
```

Program source code:

```
SCHEMA office

MAIN

  DEFINE custrec RECORD LIKE customer.*

  OPTIONS INPUT WRAP

  OPEN FORM f FROM "form1"
  DISPLAY FORM f

  LET INT_FLAG = FALSE
  INPUT custrec.* FROM sr_cust.*

  IF INT_FLAG = FALSE THEN
    DISPLAY custrec.*
    LET INT_FLAG = FALSE
  END IF

END MAIN
```

Example 2: INPUT with binding by field name

Form definition file "custlist.per" (same as in Example 1)

Program source code:

```
SCHEMA shop

MAIN

  DEFINE custrec RECORD LIKE customer.*
  DEFINE upd INTEGER

  DATABASE shop
  OPTIONS INPUT WRAP
  OPEN FORM f FROM "form1"
  DISPLAY FORM f

  LET custrec.id = arg_val(1)
  LET upd = (custrec.id < 0)

  LET INT_FLAG = FALSE
  INPUT BY NAME custrec.* ATTRIBUTES(UNBUFFERED, WITHOUT DEFAULTS=upd)
    BEFORE INPUT
      MESSAGE "Enter customer information..."
      IF upd THEN
         SELECT fname, lname INTO custrec.fname, customer.lname
            FROM customer WHERE customer.id = custrec.id
```

```
      END IF
    AFTER FIELD fname
      IF FIELD_TOUCHED(custrec.fname) AND custrec.fname IS NULL THEN
        LET custrec.lname = NULL
      END IF
    AFTER INPUT
      MESSAGE "Input terminated..."
  END INPUT

  IF INT_FLAG = FALSE THEN
    DISPLAY custrec.*
    LET INT_FLAG = FALSE
  END IF

END MAIN
```

Read-only record list (DISPLAY ARRAY)

The DISPLAY ARRAY instruction provides record list navigation in an application form, with optional record modification actions.

- Understanding the DISPLAY ARRAY instruction on page 1075
- Syntax of DISPLAY ARRAY instruction on page 1076
- DISPLAY ARRAY programming steps on page 1077
- Using read-only record lists on page 1078
  - Variable binding in DISPLAY ARRAY on page 1078
  - DISPLAY ARRAY instruction configuration on page 1079
  - Default actions in DISPLAY ARRAY on page 1080
  - DISPLAY ARRAY data blocks on page 1081
  - DISPLAY ARRAY control blocks on page 1082
  - DISPLAY ARRAY interaction blocks on page 1086
  - DISPLAY ARRAY control instructions on page 1095
- Examples on page 1095
  - Example 1: DISPLAY ARRAY using full list mode on page 1095
  - Example 2: DISPLAY ARRAY using paged mode on page 1096
  - Example 3: DISPLAY ARRAY using modification triggers on page 1097

Understanding the DISPLAY ARRAY instruction

The DISPLAY ARRAY is a dialog instruction designed to browse a list of records, binding a static or dynamic array model to a screen array of the current displayed form.

A DISPLAY ARRAY instruction supports additional features such as drag & drop, tree-view management, built-in sort and search, multi-row selection and list modification triggers. For a detailed description of these features, see Table views on page 1345.

Use the DISPLAY ARRAY instruction to let the end user browse in a list of rows, after fetching a result set from the database. The result set is produced with a database cursor executing a SELECT statement. The SELECT SQL statement is usually completed at runtime with a WHERE clause produced from a CONSTRUCT dialog. When the DISPLAY ARRAY statement completes execution, the program must test the INT_FLAG variable to check if the dialog was validated (or canceled) to take into account (or ignore) the row that was chosen by the user.

Syntax of DISPLAY ARRAY instruction

The DISPLAY ARRAY instruction controls the display of a program array on the screen.

Syntax

```
DISPLAY ARRAY array TO screen-array.*
  [ HELP help-number ]
  [ ATTRIBUTES ( { display-attribute
                 | control-attribute }
                    [,...]) ]
  [  dialog-control-block
  [...]
END DISPLAY ]
```

where *dialog-control-block* is one of:

```
{ BEFORE DISPLAY
| AFTER DISPLAY
| BEFORE ROW
| AFTER ROW
| ON IDLE seconds
| ON TIMER seconds
| ON ACTION action-name
             [ ATTRIBUTES ( action-attributes-display-array ) ]
| ON FILL BUFFER
| ON SELECTION CHANGE
| ON SORT
| ON APPEND [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON INSERT [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON UPDATE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON DELETE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON EXPAND ( row-index )
| ON COLLAPSE ( row-index )
| ON DRAG_START ( dnd-object )
| ON DRAG_FINISH ( dnd-object )
| ON DRAG_ENTER ( dnd-object )
| ON DRAG_OVER ( dnd-object )
| ON DROP ( dnd-object )
| ON KEY ( key-name [,...] )
}
  dialog-statement
  [...]
```

where *action-attributes-display-array* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
| ROWBOUND
    [,...] }
```

where *action-attributes-listmod-triggers* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
```

```
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *dialog-statement* is one of:

```
{ statement
| EXIT DISPLAY
| CONTINUE DISPLAY
| ACCEPT DISPLAY
}
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

where *control-attribute* is:

```
{ ACCEPT [ = boolean ]
| CANCEL [ = boolean ]
| KEEP CURRENT ROW [ = boolean ]
| HELP = help-number
| COUNT = row-count
| UNBUFFERED [ = boolean ]
| DETAILACTION = action-name
| DOUBLECLICK = action-name
| ACCESSORYTYPE = { DETAILBUTTON | DISCLOSUREINDICATOR | CHECKMARK }
}
```

1. *array* is a static or dynamic array containing the records you want to display.
2. *screen-array* is the name of the screen array used to display data.
3. *help-number* is an integer that associates a help message number with the instruction.
4. *action-name* identifies an action that can be executed by the user.
5. *seconds* is an integer literal or variable that defines a number of seconds.
6. *row-index* identifies the program variable which holds the row index corresponding to the tree view node that has been expanded or collapsed.
7. *dnd-object* references a `ui.DragDrop` variable defined in the scope of the dialog.
8. *key-name* is an hot-key identifier (such as `F11` or `Control-z`).
9. *statement* is any instruction supported by the language.
10. *row-count* defines the total number of rows for a static array.
11. *boolean* is a boolean expression that evaluates to `TRUE` or `FALSE`.
12. *action-attributes* are dialog-specific action attributes.

DISPLAY ARRAY programming steps

Follow this procedure to use the `DISPLAY ARRAY` dialog instruction.

The following steps describe how to use the `DISPLAY ARRAY` statement:

1. Create a form specification file containing a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the rows.
2. Make sure that the program controls interruption handling with `DEFER INTERRUPT`, to manage the validation/cancellation of the interactive dialog.
3. Define an array of records with the `DEFINE` instruction. The members of the program array must correspond to the elements of the screen array, by number and data types. Static or a dynamic

arrays can be used for the full list mode, but the paged mode requires a dynamic array. For new developments, use dynamic arrays in both cases.

4. Open and display the form, using `OPEN WINDOW WITH FORM` or the `OPEN FORM` / `DISPLAY FORM` instructions.
5. If you want to use the full list mode, fill the program array with data, typically with a result set cursor, counting the number of program records being filled with retrieved data.
6. Set the `INT_FLAG` variable to `FALSE`.
7. Write the `DISPLAY ARRAY` statement block. When using a static array, specify the number of rows with the `COUNT` attribute in the `ATTRIBUTES` clause, or call the `SET_COUNT()` function before the dialog block. With dynamic arrays, the number of rows is automatically known by the dialog. Consider using the `UNBUFFERED` mode in new developments.
8. If you want to use the paged mode, define the total number of rows with the `COUNT` attribute (can be -1 for infinite number of rows), and add the `ON FILL BUFFER` clause that will contain the code to fill the dynamic array with the expected rows from `fgl_dialog_getBufferStart()` to `fgl_dialog_getBufferLength()`.
9. If multi-row selection is needed, call the `ui.Dialog.setSelectionMode()` method in `BEFORE DISPLAY` to enable this mode.
10. Inside the `DISPLAY ARRAY` block, control the behvior of the instruction with `BEFORE ROW`, `AFTER ROW` and `ON ACTION` blocks.
11. After the interaction statement block, test the `INT_FLAG` predefined variable to check if the dialog was canceled (`INT_FLAG=TRUE`) or validated (`INT_FLAG=FALSE`). If the `INT_FLAG` variable is `TRUE`, reset it to `FALSE` to not disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.
12. If needed, get the current row with the `ARR_CURR()` built-in function after dialog execution. During dialog execution, you can also use `DIALOG.getCurrentRow()`.

Using read-only record lists

To use read-only record lists, you must understand how they work and how to structure the code.

Variable binding in DISPLAY ARRAY

The `DISPLAY ARRAY` statement binds the members of the array of record to the screen array fields specified with the `TO` keyword. Array members and screen array fields are bound by position (i.e. not by name). The number of members in the program array must match the number of fields in the screen record (that is, in a single row of the screen array).

```
SCHEMA stock
DEFINE cust_arr DYNAMIC ARRAY OF customer.*
...
DISPLAY ARRAY cust_arr TO sr.*
        ATTRIBUTES(UNBUFFERED)
   ...
END DISPLAY
```

Keep in mind that array members are bound to screen array fields by position, so you must make sure that the members of the array are defined in the same order as the screen array fields.

Note that the array is usually defined with a flat list of members with `ARRAY OF RECORD / END RECORD`. However, the array can be structured with sub-records and still be used with a `DISPLAY ARRAY` dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute):

```
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
                    item_data RECORD LIKE items.*,
                    it_image STRING,
```

```
                    it_count INTEGER
                END RECORD
...
DISPLAY ARRAY a_items TO sr.*
   ...
```

When using the UNBUFFERED attribute, the instruction is sensitive to program variable changes. This means that you do not have to DISPLAY the values; setting the program variable used by the dialog automatically displays the data in the corresponding form field.

```
ON ACTION change
   LET arr[arr_curr()].field1 = newValue()
```

If the program array has the same structure as a database table (this is the case when the array is defined with a DEFINE LIKE clause), you may not want to display/use some of the columns. You can achieve this by using PHANTOM fields in the screen array definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

DISPLAY ARRAY instruction configuration

This section describes the options that can be specified in the ATTRIBUTES clause of the DISPLAY ARRAY instruction. The options of the ATTRIBUTES clause override all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. With the DISPLAY ARRAY statement, the INVISIBLE attribute is ignored.

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help the DISPLAY ARRAY dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

COUNT option

When using a dynamic array, the number of rows to be displayed is defined by the number of elements in the dynamic array; the COUNT attribute is ignored.

When using a static array or the paged mode, the number of rows to be displayed is defined by the COUNT attribute. You can also use the SET_COUNT( ) built-in function, but it is supported for backward compatibility only. If you don't know the total number of rows for the paged mode, you can specify -1 for the COUNT attribute (or in the SET_COUNT( ) call before the dialog block): With COUNT=-1, the dialog will ask for rows by executing ON FILL BUFFER until you provide less rows as asked, or if you reset the number of rows to a value higher as -1 with ui.Dialog.setArrayLength().

KEEP CURRENT ROW option

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the default accept action. Use this attribute when you want to avoid dialog validation, or if you need to write a specific validation procedure by using ACCEPT DISPLAY.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. Use this attribute when you only need a validation action (accept), or when you want to write a specific cancellation procedure by using EXIT DISPLAY.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.

DOUBLECLICK option

The DOUBLICKCLICK option can be used to define the action that will be fired when the user chooses a row from the list. On front-end platforms using a mouse-device, this corresponds to a physical double-click on a row with the mouse. On mobile front-ends, this corresponds to a tap on the row with a finger. Note that this attribute can also be defined for the TABLE/TREE containers in form files; DOUBLECLICK in DISPLAY ARRAY attributes has a higher precedence as DOUBLECLICK in the form file. For more details, see Defining the action for a row choice on page 1360.

DETAILACTION option

> **Important:** This feature is only for mobile platforms.

The DETAILACTION attribute can be used to define the action that will be fired when the user selects the detail button of a row. The detail button is typically shown with a (i) icon on iOS devices. Note that the DOUBLECLICK attribute can be used to distinguish the action when the user selects the row instead of the detail button in the row. For more details, see Row configuration on iOS devices on page 1369.

ACCESSORTYPE option

> **Important:** This feature is only for mobile platforms.

The ACCESSORYTYPE attribute can be used to define the decoration of rows, typically used on a iOS device. Values can be DETAILBUTTON, DISCLOSUREINDICATOR, CHECKMARK to respectively get a (i), > or checkmark icon. For more details, see Row configuration on iOS devices on page 1369.

Default actions in DISPLAY ARRAY

When an DISPLAY ARRAY instruction executes, the runtime system creates a set of default actions.

According the invoked default action, field validation occurs and different DISPLAY ARRAY control blocks are executed.

This table lists the default actions created for this dialog:

Table 272: Default actions created for the DISPLAY ARRAY dialog

Default action	Description
accept	Validates the DISPLAY ARRAY dialog (validates current row selection) *Creation can be avoided with* ACCEPT *attribute.*
cancel	Cancels the DISPLAY ARRAY dialog (no validation, INT_FLAG is set to TRUE) *Creation can be avoided with* CANCEL *attribute.*
close	By default, cancels the DISPLAY ARRAY dialog (no validation, INT_FLAG is set to TRUE)

Default action	Description
	Default action view is hidden. See Implementing the close action on page 1337.
`help`	Shows the help topic defined by the `HELP` clause. *Only created when a* `HELP` *clause is defined.*
`nextrow`	Moves to the next row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`prevrow`	Moves to the previous row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`firstrow`	Moves to the first row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`lastrow`	Moves to the last row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`find`	Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value. *Only created if the context allows built-in find.*
`findnext`	Seeks to the next row matching the value entered during the fglfind dialog. *Only created if the context allows built-in find.*

The accept and cancel default actions can be avoided with the `ACCEPT` and `CANCEL` dialog control attributes:

```
DISPLAY ARRAY arr TO sr.* ATTRIBUTES( CANCEL=FALSE, ... )
  ...
```

DISPLAY ARRAY data blocks

Data blocks are dialog triggers that are invoked when the dialog controller needs data to feed the view with values.

Such blocks are typically used when record list data is provided dynamically, with the display array paged mode of when implementing dynamic tree-views.

ON FILL BUFFER block
The `ON FILL BUFFER` block is used to fill a page of rows into the dynamic array, according to an offset and a number of rows.

This data block is used in the `DISPLAY ARRAY` blocks.

The offset can be retrieved with the `FGL_DIALOG_GETBUFFERSTART()` built-in function and the number of rows to provide is defined by the `FGL_DIALOG_GETBUFFERLENGTH()` built-in function.

ON EXPAND block
The `ON EXPAND` block is executed when a tree view node is expanded (i.e. opened).

This data block is used to implement dynamic trees in a `DISPLAY ARRAY`, where nodes are added according to the nodes opened by the end user.

ON COLLAPSE block
The `ON COLLAPSE` block is executed when a tree view node is collapsed (i.e. closed).

This data block is used to implement dynamic trees in a `DISPLAY ARRAY`, where nodes are removed according to the nodes closed by the end user.

DISPLAY ARRAY control blocks
DISPLAY ARRAY control blocks execution order

This table shows the order in which the runtime system executes the control blocks in the `DISPLAY ARRAY` instruction, according to the user action:

Table 273: Control blocks execution order in `DISPLAY ARRAY`

Context / User action	Control Block execution order
Entering the dialog	**1.** `BEFORE DISPLAY` **2.** `BEFORE ROW`
Moving to a different row	**1.** `AFTER ROW` (the current row) **2.** `BEFORE ROW` (the new row)
Validating the dialog	**1.** `AFTER ROW` **2.** `AFTER DISPLAY`
Canceling the dialog	**1.** `AFTER ROW` **2.** `AFTER INPUT`
Firing the insert or append action for the `ON INSERT` block	**1.** `AFTER ROW` **2.** `ON INSERT` **3.** `BEFORE ROW`
Firing the delete action for the `ON DELETE` block	**1.** `AFTER ROW` **2.** `ON DELETE` **3.** `BEFORE ROW`

BEFORE DISPLAY block

BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a singular `DISPLAY ARRAY` instruction, or when used as parallel dialog, the `BEFORE DISPLAY` is only executed once when the dialog is started.

The BEFORE DISPLAY block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating actions the user is not allowed to execute.

```
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", is_super_user())
```

BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE DISPLAY block is executed when a DISPLAY ARRAY list gets the focus.

BEFORE DISPLAY is executed before the BEFORE ROW block.

In this example the BEFORE DISPLAY block enables an action and displays a message:

```
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("print_list", TRUE)
       MESSAGE "You are now in the list of items"
```

AFTER DISPLAY block

AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a singular DISPLAY ARRAY instruction, or when used as parallel dialog, the AFTER DISPLAY is only executed once when dialog is ended.

You typically implement dialog finalization in this block.

```
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       DISPLAY "Current row is: ", arr_curr()
```

AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a DISPLAY ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER DISPLAY block is executed when a DISPLAY ARRAY list loses the focus and goes to another sub-dialog.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER DISPLAY is executed after the AFTER ROW block.

In this example, the AFTER DISPLAY block disables an action that is specific to the current list:

```
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", FALSE)
```

BEFORE ROW block

BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE ROW block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, BEFORE ROW will be executed for the current row, but only if there are data rows in the array.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example, the BEFORE ROW block gets the new row number and displays it in a message:

```
DISPLAY ARRAY ...
   ...
   BEFORE ROW
     MESSAGE "We are on row # ", arr_curr()
   ...
```

BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the BEFORE DISPLAY or BEFORE INPUT blocks instead.

In DISPLAY ARRAY, BEFORE ROW is executed after the BEFORE DISPLAY block. In INPUT ARRAY, BEFORE ROW is executed before the BEFORE INSERT and BEFORE FIELD blocks and after the BEFORE INPUT blocks.

When the procedural dialog starts, BEFORE ROW will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, BEFORE ROW will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular DISPLAY ARRAY or INPUT ARRAY. In singular dialogs, the BEFORE ROW block is always executed when the dialog starts (and there are rows in the array).

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example the BEFORE ROW block displays a message with the current row number:

```
   DISPLAY ARRAY p_items TO s_items.*
     BEFORE ROW
       MESSAGE "We are in items, on row #", DIALOG.getCurrentRow("s_items")
```

AFTER ROW block

AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER ROW block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A NEXT FIELD instruction executed in the AFTER ROW control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the row that you are leaving.

AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

AFTER ROW is executed after the AFTER FIELD, AFTER INSERT and before AFTER DISPLAY or AFTER INPUT blocks.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the of the row that you are leaving.

For both INPUT ARRAY and DISPLAY ARRAY sub-dialogs, a NEXT FIELD executed in the AFTER ROW control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

AFTER ROW and temporary rows in INPUT ARRAY

> **Important: After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The AFTER ROW block will be executed for the temporary row, but ui.Dialog.getCurrentRow()/arr_curr() will be one row greater than ui.Dialog.getArrayLength()/ARR_COUNT(). In this case, you should ignore the AFTER ROW event. For example, you should not try to execute a NEXT FIELD or CONTINUE INPUT instruction, nor should you try to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.**

In this example, the AFTER ROW block checks the current row index and verifies a variable value to forces the focus to stay in the current row if the value is wrong:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   AFTER ROW
      LET r = DIALOG.getCurrentRow("s_items")
      IF r <= DIALOG.getArrayLength("s_items") THEN
        IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
           ERROR "Item quantity is not valid"
           NEXT FIELD item_quantity'
         END IF
      END IF
```

Another way to handle the case of temporary rows in AFTER ROW is to use a flag to know if the AFTER INSERT block was executed: The AFTER INSERT block is not executed if the temporary row is automatically removed. By setting a first value in BEFORE INSERT and changing the flag in AFTER INSERT, you can detect if the row was permanently added to the list:

```
INPUT ARRAY p_items FROM s_items.*
  ...
  BEFORE INSERT
    LET op = "T"
    ...
  AFTER INSERT
    LET op = "I"
    ...
  AFTER ROW
    IF op == "I" THEN
      IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
         ERROR "Item quantity is not valid"
```

```
         NEXT FIELD item_quantity
        END IF
      WHENEVER ERROR CONTINUE
      INSERT INTO items (item_num, item_name, item_quantity)
                         VALUES ( p_item[arr_curr()].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<0 THEN
         ERROR "Could not insert the record into database!"
         NEXT FIELD CURRENT
      ELSE
         MESSAGE "Record has been inserted successfully"
      END IF
   END IF
...
```

DISPLAY ARRAY interaction blocks

ON ACTION block

The `ON ACTION` *`action-name`* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
  ON ACTION action-name
     instruction
     ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
   ON ACTION zoom
      CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as `INPUT`, `INPUT ARRAY` and `CONSTRUCT`, if an action is specific to a field, add the `INFIELD` clause to have the action automatically enabled when the corresponding field gets the focus:

```
   ON ACTION zoom INFIELD cust_city
      CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the `ON ACTION` handler needs to define its own attributes at the program level. This can be done by adding the `ATTRIBUTES()` clause of `ON ACTION`:

```
   ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
      CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The `ON IDLE` *`seconds`* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of `ON IDLE` must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

> **Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An `ON KEY (`*`key-name`*`)` block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The `ON KEY` block is supported for backward compatibility with TUI mode applications.

An `ON KEY` block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, `ON KEY(F5,F6)` creates two actions with the names `f5` and `f6`. Each action object will get an `ACCELERATORNAME` assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON TIMER block

The `ON TIMER` *`seconds`* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

Important: The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON APPEND block

Similarly to the `ON INSERT` control block, the `ON APPEND` trigger can be used to enable row creation during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the append action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an `ON ACTION append` interaction block and the `ON APPEND` block is used, the compiler will stop with error -8408.

When the user fires the append action, the dialog first execute the user code of the `AFTER ROW` block if defined. Then the dialog moves to the end of the list, and creates a new row after the last existing row. After creating the row, the dialog executes the user code of the `ON APPEND` block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular `INPUT` statement, to let the end user enter data for the new created row. This is typically done with an `INPUT` binding explicitly array fields to the screen record fields. The new current row in the program array is identified with `arr_curr()`, and the current screen line in the form is defined by `SCR_LINE()`:

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON APPEND
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the `INPUT` instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the new row is kept in the program array, and the `BEFORE ROW` block is executed for the new created row.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the new row is removed from the program array, and the `BEFORE ROW` block is executed for the row that was existing at the current position, before the new row was created.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON APPEND` block.

The append action is disabled if the maximum number of rows is reached.

If needed, the `ON APPEND` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
  ON APPEND ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON INSERT block

Similarly to the `ON APPEND` control block, the `ON INSERT` trigger can be used to enable row creation during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the insert action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an `ON ACTION insert` interaction block and the `ON INSERT` block is used, the compiler will stop with error -8408.

When the user fires the insert action, the dialog first execute the user code of the `AFTER ROW` block if defined. Then the new row is created: The insert action creates a new row before current row in the list. After creating the row, the dialog executes the user code of the `ON INSERT` block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular INPUT statement, to let the end user enter data for the new created row. This is typically done with an INPUT binding explicitly array fields to the screen record fields. The new current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by scr_line():

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON INSERT
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the new row is kept in the program array, and the BEFORE ROW block is executed for the new created row.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the new row is removed from the program array, and the BEFORE ROW block is executed for the row that was existing at the current position, before the new row was created.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON INSERT block.

The insert action is disabled if the maximum number of rows is reached.

If needed, the ON INSERT handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
  ON INSERT ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON UPDATE block

The ON UPDATE trigger can be used to enable row modification during a DISPLAY ARRAY dialog. If this block is defined, the dialog will automatically create the update action. This action can be decorated, enabled and disabled as regular actions.

You typically configure the TABLE container in the form by defining the DOUBLECLICK attribute to "update", in order to trigger the update action when the user double-clicks on a row.

If the dialog defines an ON ACTION update interaction block and the ON UPDATE block is used, the compiler will stop with error -8408.

When the user fires the *update* action, the dialog executes the user code of the ON UPDATE block.

The dialog handles only the row modification action and navigation, you must program the record input with a regular INPUT statement, to let the end user modify the data of the current row. This is typically done with an INPUT binding explicitly array fields to the screen record fields, with the WITHOUT DEFAULTS clause. The current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by scr_line():

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON UPDATE
    INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the modified values of the current row are kept in the program array.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the old values of the current row are restored in the program array.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON UPDATE block.

If needed, the ON UPDATE handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
  ON UPDATE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON DELETE block

The ON DELETE trigger can be used to enable row deletion during a DISPLAY ARRAY dialog. If this block is defined, the dialog will automatically create the delete action. This action can be decorated, enabled and disabled as regular actions.

If the dialog defines an ON ACTION delete interaction block and the ON DELETE block is used, the compiler will stop with error -8408.

When the user fires the delete action, the dialog executes the user code of the ON DELETE block.

The dialog handles only the row deletion action and navigation, you can typically program a validation dialog box to let the user confirm the deletion. The current row in the program array is identified with arr_curr():

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON DELETE
    IF fgl_winQuestion("Delete",
          "Do you want to delete this record?",
          "yes", "no|yes", "help", 0) == "no"
    THEN
       LET int_flag = TRUE
    END IF
    ...
```

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the current row is deleted from the program array, and the BEFORE ROW block is executed for the next row in the list.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the current row is kept in the program array, and the BEFORE ROW block is executed again for the current row.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON DELETE block.

If needed, the ON DELETE handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
  ON DELETE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON SELECTION CHANGE block

The ON SELECTION CHANGE trigger can be used to enable multi-row selection and detect when rows are selected or de-selected by the end user during a DISPLAY ARRAY dialog. If this block is defined, multi-row selection is automatically enableb. However, the feature can be enabled/disabled with the setSelectionMode() dialog method.

ON SORT block

The `ON SORT` interfaction block can be used to detect when rows have to be sorted in a `DISPLAY ARRAY` or `INPUT ARRAY` dialog.

`ON SORT` is used in two different contexts:

1. In a regular `DISPLAY ARRAY` / `INPUT ARRAY` dialog (not using paged mode), the `ON SORT` trigger can be used to detect that a list sort was performed. In this case, the (visual) sort is already done by the runtime system and the `ON SORT` block is only used to execute post-sort tasks, such as displaying current row information, by using arrayToVisualIndex() dialog method. It is also possible to get the sort column and order with the `getSortKey()` and `getSortSelection()` dialog methods.
2. In a `DISPLAY ARRAY` using paged mode (`ON FILL BUFFER`), built-in row sorting is not available because data is provided by pages. Use the `ON SORT` trigger to detect a sort request and perform a new SQL query to re-order the rows. In this case, sort column and order is available with the `getSortKey()` and `getSortSelection()` dialog methods. See Populating a DISPLAY ARRAY on page 1372.

ON DRAG_START block

The `ON DRAG_START` block is executed when the end user has begun the drag operation. If this dialog trigger has not been defined, default dragging is enabled for this dialog.

In the `ON DRAG_START` block, the program typically specifies the type of drag & drop operation by calling `ui.DragDrop.setOperation()` with "move" or "copy". This call will define the default and unique drag operation. If needed, the program can allow another type of drag operation with `ui.DragDrop.addPossibleOperation()`. The end user can then choose to move or copy the dragged object, if the drag & drop target allows it.

If the dragged object can be dropped outside the program, must define the MIME type and drag/drop data with `ui.DragDrop.setMimeType()` and `ui.DragDrop.setBuffer()` methods.

Example:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    CALL dnd.setOperation("move") -- Move is the default operation
    CALL dnd.addPossibleOperation("copy") -- User can toggle to copy if
 needed
    CALL dnd.setMimeType("text/plain")
    CALL dnd.setBuffer(arr[arr_curr()].cust_name)
    ...
END DISPLAY
```

ON DRAG_FINISHED block

Execution of the `ON DRAG_FINISHED` block notifies the dialog where the drag started that the drop operation has been completed or terminated.

Call `ui.DragDrop.getOperation()` to get the final type of operation of the drop. On successful completion, the method returns "move" or "copy"; otherwise the function returns `NULL`. If `NULL` is returned, the `ON DRAG_FINISHED` trigger can be ignored.

In cases of successful moves to a target out of the current `DISPLAY ARRAY`, the application must remove the transferred data from the source model. For example, if a row was moved from dialog A to B, dialog A will get an `ON DRAG_FINISHED` execution after the row was dropped into B, and should remove the row from the list A.

The `ON DRAG_FINISHED` interaction block is optional.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    LET last_dragged_row = arr_curr()
    ...
  ON DRAG_FINISHED (dnd)
    IF dnd.getOperation() == "move" THEN
      CALL DIALOG.deleteRow(last_dragged_row)
    END IF
    ...
END DISPLAY
```

ON DRAG_ENTER block

When the `ON DROP` control block is defined, the `ON DRAG_ENTER` block will be executed when the mouse cursor enters the visual boundaries of the drop target dialog. Entering the target dialog is accepted by default if no `ON DRAG_ENTER` block is defined. However, when `ON DROP` is defined, you should also define `ON DRAG_ENTER` to deny the drop of objects with an unsupported MIME type that come from other applications.

The program can decide to deny or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a `NULL` to the method will deny drop.

To check what MIME type is available in the drag & drop buffer, the program uses the `ui.DragDrop.selectMimeType()` method. This method takes the MIME type as a parameter and returns `TRUE` if the passed MIME type is used. You can call this method several times to check the availability of different MIME types.

You may also define the visual effect when flying over the target list with `ui.DragDrop.setFeedback()`.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    IF dnd.selectMimeType("text/plain") THEN
      CALL dnd.setOperation("copy")
      CALL dnd.setFeedback("all")
    ELSE
      CALL dnd.setOperation(NULL)
    END IF
  ON DROP (dnd)
    ...
END DISPLAY
```

Once the mouse has entered the target area, subsequent mouse cursor moves can be detected with the `ON DRAG_OVER` trigger.

When using a table or tree-view as drop target, you can control the visual effect when the mouse flies over the rows, according to the type of drag & drop you want to achieve.

Basically, a dragged object can be:

1. Inserted in between two rows (visual effect must show where the object will be inserted)
2. Copied/merged to the current row (visual effect must show the row under the mouse)
3. Dropped somewhere on the target widget (the exact location inside the widget does not matter)

The visual effect can be defined with the `ui.DragDrop.setFeedback()` method, typically called in the `ON DRAG_ENTER` block.

The values to pass to the `setFeedback()` method to get the desired visual effects described are respectively:

1. `insert` (default)
2. `select`
3. `all`

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    IF canDrop() THEN
      CALL dnd.setOperation(NULL)
    ELSE
      CALL dnd.setFeedback("select")
    END IF
    ...
END DISPLAY
```

ON DRAG_OVER block

When the `ON DROP` control block is defined, the `ON DRAG_OVER` block will be executed after `ON DRAG_ENTER`, when the mouse cursor is moving over the drop target, or when the drag & drop operation has changed (toggling copy/move).

`ON DRAG_OVER` will be called only once per row, even if the mouse cursor moves over the row.

In the `ON DRAG_OVER` block, the method `ui.DragDrop.getLocationRow()` returns the index of the row in the target array, and can be used to allow or deny the drop. When using a tree-view, you must also check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and allow/deny the drop operation accordingly.

The program can change the drop operation at any execution of the `ON DRAG_OVER` block. You can deny or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a `NULL` to the method will deny the drop.

The current operation (returned by `ui.DragDrop.getOperation()`) is the value set in previous `ON DRAG_ENTER` or `ON DRAG_OVER` events, or the operation selected by the end user, if it can toggle between copy and move. Thus, `ON DRAG_OVER` can occur even if the mouse position has not changed.

If dropping has been denied with `ui.DragDrop.setOperation(NULL)` in the previous `ON DRAG_OVER` event, the program can reset the operation to allow a drop with a call to `ui.DragDrop.setOperation()` with the operation parameter "move" or "copy".

`ON DRAG_OVER` will not be called if drop has been disabled in `ON DRAG_ENTER` with `ui.DragDrop.setOperation(NULL)`

`ON DRAG_OVER` is optional, and must only be defined if the operation or the acceptance of the drag object depends on the target row of the drop target.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    ...
  ON DRAG_OVER (dnd)
    IF arr[dnd.getLocationRow()].acceptsCopy THEN
      CALL dnd.setOperation("copy")
```

```
    ELSE
      CALL dnd.setOperation(NULL)
    END IF
  ON DROP (dnd)
    ...
END DISPLAY
```

During a drag & drop process, the end user (or the target application) can decide to modify the type of the operation, to indicate whether the dragged object has to be copied or moved from the source to the target. For example, in a typical file explorer, by default files are moved when doing a drag & drop on the same disk. To make a copy of a file, you must press the Ctrl key while doing the drag & drop with the mouse.

In the drop target dialog, you can detect such operation changes in the `ON DRAG_OVER` trigger and query the `ui.DragDrop` object for the current operation with `ui.DragDrop.getOperation()`. In the drag source dialog, you typically check `ui.DragDrop.getOperation()` in the `ON DRAG_FINISHED` trigger to know what sort of operation occurred, to keep ("copy" operation) or delete ("move" operation) the original dragged object.

This example tests the current operation in the drop target list and displays a message accordingly:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    ...
  ON DRAG_OVER (dnd)
    CASE dnd.getOperation()
    WHEN "move"
      MESSAGE "The object will be moved to row ", dnd.getLocationRow()
    WHEN "copy"
      MESSAGE "The object will be copied to row ", dnd.getLocationRow()
    END CASE
    ...
  ON DROP (dnd)
    ...
END DISPLAY
```

ON DROP block

To enable drop actions on a list, you must define the `ON DROP` block; otherwise the list will not accept drop actions.

The `ON DROP` block is executed after the end user has released the mouse button to drop the dragged object. `ON DROP` will not occur if drop has been denied in the previous `ON DRAG_OVER` event or in `ON DRAG_ENTER` with a call to `ui.DragDrop.setOperation(NULL)`.

The program might also check the MIME type of the dragged object with `ui.DragDrop.getSelectedMimeType()`, and then call the `ui.DragDrop.getBuffer()` method to retrieve drag & drop data from external applications.

Ideally the drop operation should be accepted (no additional call to `ui.DragDrop.setOperation()`).

In this block, the `ui.DragDrop.getLocationRow()` method returns the index of the row in the target array, and can be used to execute the code to get the drop data / object into the row that has been chosen by the user.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DROP (dnd)
    LET arr[dnd.getLocationRow()].capacity == dnd.getBuffer()
```

```
    ...
END DISPLAY
```

If the drag & drop operations are local to the same list or tree-view controller, you can use the `ui.DragDrop.dropInternal()` method to simplify the code. This method implements the typical move of the dragged rows or tree-view node. This is especially useful in case of a tree-view, but is also the preferred way to move rows around in simple tables.

This `ON DROP` code example uses the `dropInternal()` method:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr_tree TO sr_tree.* ...
  ...
  ON DROP (dnd)
    CALL dnd.dropInternal()
    ...
END DISPLAY
```

If you want to implement by hand the code to drop a node in a tree-view, you must check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and execute the code corresponding to the drop operation: If the drop target row index returned by `getLocationRow()` is a child of the parent row index returned by `getLocationParent()` , the new row must be inserted before `getLocationRow()`; otherwise the new row must be added as a child of the parent node identified by `getLocationParent()`.

DISPLAY ARRAY control instructions
CONTINUE DISPLAY instruction

`CONTINUE DISPLAY` skips all subsequent statements in the current control block and gives the control back to the dialog.

The `CONTINUE DISPLAY` instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. If this instruction is called in a control block that is not `AFTER DISPLAY`, further control blocks might be executed according to the context.

Actually, `CONTINUE DISPLAY` just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER DISPLAY`, the focus returns to the current row in the list, giving the user another chance to browse and select a row. In this case the `BEFORE ROW` of the current row will be invoked.

EXIT DISPLAY instruction

Use the `EXIT DISPLAY` to terminate the `DISPLAY ARRAY` instruction and resume the program execution at the instruction immediately following the `DISPLAY ARRAY` block.

ACCEPT DISPLAY instruction

The `ACCEPT DISPLAY` instruction validates the `DISPLAY ARRAY` instruction and exits the dialog block.

The `AFTER DISPLAY` control block will be executed. Statements after `ACCEPT DISPLAY` will not be executed.

Examples
Various examples using the `DISPLAY ARRAY` dialog instruction.
Example 1: DISPLAY ARRAY using full list mode

Form definition file "custlist.per":

```
SCHEMA shop

LAYOUT
TABLE
```

```
{
 Id       Name         LastName
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
}
END
END

TABLES
  customer
END

ATTRIBUTES
  f001 = customer.id;
  f002 = customer.fname;
  f003 = customer.lname;
END

INSTRUCTIONS
  SCREEN RECORD srec[6] (customer.*);
END
```

Application:

```
SCHEMA shop

MAIN

  DEFINE cnt INTEGER
  DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*

  DATABASE shop

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 CURSOR FOR
    SELECT id, fname, lname FROM customer
  LET cnt = 1
  FOREACH c1 INTO arr[cnt].*
    LET cnt = cnt + 1
  END FOREACH
  CALL arr.deleteElement(cnt)

  DISPLAY ARRAY arr TO srec.*
    ON ACTION print
       DISPLAY "Print a report"
  END DISPLAY

END MAIN
```

Example 2: DISPLAY ARRAY using paged mode

Form definition file "custlist.per" (same as in Example 1)

Application:

```
SCHEMA shop
```

```
MAIN

  DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
  DEFINE cnt, ofs, len, row, i INTEGER

  DATABASE shop

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 SCROLL CURSOR FOR
         SELECT id, fname, lname FROM customer
  OPEN c1
  DISPLAY ARRAY arr TO srec.* ATTRIBUTES(COUNT=-1)
    ON FILL BUFFER
       LET ofs = fgl_dialog_getBufferStart()
       LET len = fgl_dialog_getBufferLength()
       LET row = ofs
       FOR i=1 TO len
          FETCH ABSOLUTE row c1 INTO arr[i].*
          IF SQLCA.SQLCODE!=0 THEN
            CALL DIALOG.setArrayLength("srec",row-1)
            EXIT FOR
          END IF
          LET row = row + 1
       END FOR
    AFTER DISPLAY
       IF NOT int_flag THEN
          DISPLAY "Selected customer is #"
                  || arr[arr_curr()-ofs+1].id
       END IF
  END DISPLAY
END MAIN
```

Example 3: DISPLAY ARRAY using modification triggers

Form definition file "custlist.per" (same as in Example 1)

Application:

```
SCHEMA shop

MAIN

  DEFINE arr DYNAMIC ARRAY OF RECORD LIKE customer.*
  DEFINE cnt, ofs, len, row, i INTEGER

  DATABASE shop

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 CURSOR FOR
    SELECT id, fname, lname FROM customer
  LET cnt = 1
  FOREACH c1 INTO arr[cnt].*
    LET cnt = cnt + 1
  END FOREACH
  CALL arr.deleteElement(cnt)

  DISPLAY ARRAY arr TO srec.* ATTRIBUTES(UNBUFFERED)
        ON UPDATE
            INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM
 srec[scr_line()].* ;
```

```
        ON INSERT
            INPUT arr[arr_curr()].* FROM srec[scr_line()].* ;
        ON APPEND
            INPUT arr[arr_curr()].* FROM srec[scr_line()].* ;
        ON DELETE
            MENU "Delete" ATTRIBUTES(STYLE="dialog",
                                     COMMENT="Do you want to delete the current
 row?")
              COMMAND "Yes" LET int_flag = FALSE
              COMMAND "No"  LET int_flag = TRUE
            END MENU
  END DISPLAY

END MAIN
```

Editable record list (INPUT ARRAY)

The `INPUT ARRAY` instruction provides always-editable record list handling in an application form.

- Understanding the INPUT ARRAY instruction on page 1098
- Syntax of INPUT ARRAY instruction on page 1099
- INPUT ARRAY programming steps on page 1101
- Using editable record lists on page 1101
  - Variable binding in INPUT ARRAY on page 1101
  - INPUT ARRAY instruction configuration on page 1102
  - Default actions in INPUT ARRAY on page 1105
  - INPUT ARRAY control blocks on page 1106
  - INPUT ARRAY interaction blocks on page 1117
  - INPUT ARRAY control instructions on page 1119
- Examples on page 1124
  - Example 1: INPUT ARRAY with empty record list on page 1124
  - Example 2: INPUT ARRAY using a static array on page 1124
  - Example 3: INPUT ARRAY using a dynamic array on page 1125
  - Example 4: INPUT ARRAY updating the database table on page 1126

Understanding the INPUT ARRAY instruction

The `INPUT ARRAY` is a dialog instruct designed to browse and modify a list of record, binding a static or dynamic array model to a screen array of the current displayed form.

> **Important:** This feature is not supported on mobile platforms.

An `INPUT ARRAY` instruction supports additional features, built-in sort and search, multi-row selection and list modification triggers. For a detailed description of these features, see Table views on page 1345.

Use the `INPUT ARRAY` instruction to let the end user update, delete and create new records in a list, after fetching a result set from the database. The result set is produced with a database cursor executing a `SELECT` statement. The `SELECT` SQL statement is usually completed at runtime with a `WHERE` clause produced from a `CONSTRUCT` dialog.

The `INPUT ARRAY` instruction associates a program array of records with a screen-array defined in a form so that the user can update the list of records. The `INPUT ARRAY` statement activates the current form (the form that was most recently displayed or the form in the current window.)

During the `INPUT ARRAY` execution, the user can edit or delete existing rows, insert new rows, and move inside the list of records. The program controls the behavior of the instruction with control blocks such as `BEFORE DELETE`, `BEFORE INSERT`, etc.

To terminate the INPUT ARRAY execution, the user can validate (or cancel) the dialog to commit (or invalidate) the modifications made in the list of records.

When the statement completes execution, the program must test the INT_FLAG variable to check if the dialog was validated (or canceled) and then use INSERT, DELETE, or UPDATE SQL statements to modify the appropriate database tables. The database can also be updated during the execution of theINPUT ARRAY statement.

Syntax of INPUT ARRAY instruction
The INPUT ARRAY supports data entry by users into a screen array and stores the entered data in an array of records.

Syntax

```
INPUT ARRAY array
  [ WITHOUT DEFAULTS ]
  FROM screen-array.*
  [ ATTRIBUTES ( { display-attribute
                 | control-attribute
                 } [,...] ) ]
  [ HELP help-
number ]
[ dialog-control-block
   [...]
END INPUT ]
```

where *dialog-control-block* is one of:

```
{ BEFORE INPUT
| AFTER INPUT
| AFTER DELETE
| BEFORE ROW
| AFTER ROW
| BEFORE FIELD field-spec  [,...]
| AFTER FIELD field-spec  [,...]
| ON ROW CHANGE
| ON CHANGE field-spec [,...]
| ON IDLE seconds
| ON TIMER seconds
| ON ACTION action-name
             [ INFIELD field-spec ]
             [ ATTRIBUTES ( action-attributes-input-array ) ]
| ON KEY ( key-name [,...] )
| BEFORE INSERT
| AFTER INSERT
| BEFORE DELETE
}
  dialog-statement
  [...]
```

where *action-attributes-input-array* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| VALIDATE = NO
| CONTEXTMENU = { YES | NO | AUTO }
| ROWBOUND
    [,...] }
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT INPUT
| CONTINUE INPUT
| EXIT INPUT
| NEXT FIELD { CURRENT | NEXT | PREVIOUS | field-spec }
| CANCEL DELETE
| CANCEL INSERT
}
```

where *field-spec* identifies a unique field with one of:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
}
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

where *control-attribute* is:

```
{ ACCEPT [ = boolean ]
| APPEND ROW [ = boolean ]
| AUTO APPEND [ = boolean ]
| CANCEL [ = boolean ]
| COUNT = row-count
| DELETE ROW [ = boolean ]
| FIELD ORDER FORM
| HELP = help-number
| INSERT ROW [ = boolean ]
| KEEP CURRENT ROW [ = boolean ]
| MAXCOUNT = max-row-count
| UNBUFFERED [ = boolean ]
| WITHOUT DEFAULTS [ = boolean ]
}
```

1. *array* is the array of records that will be filled by the `INPUT ARRAY` statement.
2. *help-number* is an integer that allows you to associate a help message number with the instruction.
3. *field-name* is the identifier of a field of the current form.
4. *table-name* is the identifier of a database table of the current form.
5. *screen-record* is the identifier of a screen record of the current form.
6. *screen-array* is the screen array that will be used in the form.
7. *action-name* identifies an action that can be executed by the user.
8. *seconds* is an integer literal or variable that defines a number of seconds.
9. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
10. *statement* is any instruction supported by the language.
11. *row-count* defines the initial number of rows for a static array.
12. *max-row-count* is the maximum number of rows that can be created.
13. *boolean* is a boolean expression that evaluates to `TRUE` or `FALSE`.
14. *action-attributes* are dialog-specific action attributes.

INPUT ARRAY programming steps

The following steps describe how to use the INPUT ARRAY statement:

1. Create a form specification file containing a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the rows.
2. Make sure that the program controls interruption handling with DEFER INTERRUPT, to manage the validation/cancellation of the interactive dialog.
3. Define an array of records with the DEFINE instruction. The members of the program array must correspond to the elements of the screen array, by number and data types. If you want to input data from a reduced set of columns, you must define a second screen array, containing the limited list of form fields, in the form file. You can then use the second screen array in an INPUT ARRAY a FROM sa.* instruction.
4. Open and display the form, using a OPEN WINDOW WITH FORM or the OPEN FORM / DISPLAY FORM instructions.
5. If needed, fill the program array with data, for example with a result set cursor, counting the number of program records being filled with retrieved data.
6. Set the INT_FLAG variable to FALSE.
7. Write the INPUT ARRAY statement to handle data input.
8. Inside the INPUT ARRAY statement, control the behavior of the instruction with control blocks such as BEFORE INPUT, BEFORE INSERT, BEFORE DELETE, BEFORE ROW, BEFORE FIELD,AFTER INSERT,AFTER DELETE,AFTER FIELD,AFTER ROW, AFTER INPUT and ON ACTION blocks.
9. Get the new number of rows with the ARR_COUNT() built-in function or with DIALOG.getArrayLength().
10. After the interaction statement block, test the INT_FLAG predefined variable to check if the dialog was canceled (INT_FLAG=TRUE) or validated (INT_FLAG=FALSE). If the INT_FLAG variable is TRUE, you should reset it to FALSE to not disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.

Using editable record lists

To use editable record lists, you must understand how they work and how to structure the code.

Variable binding in INPUT ARRAY

The INPUT ARRAY statement binds the members of the array of record to the screen array fields specified with the FROM keyword. Array members and screen array fields are bound by position (i.e. not by name). The number of members in the program array must match the number of fields in the screen record (that is, in a single row of the screen array).

```
SCHEMA stock
DEFINE cust_arr DYNAMIC ARRAY OF customer.*
...
INPUT ARRAY cust_arr FROM sr.*
        ATTRIBUTES(UNBUFFERED)
   ...
END INPUT
```

Keep in mind that array members are bound to screen array fields by position, so you must make sure that the members of the array are defined in the same order as the screen array fields.

Note that the array is usually defined with a flat list of members with ARRAY OF RECORD / END RECORD. However, the array can be structured with sub-records and still be used with an INPUT ARRAY dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the IMAGECOLUMN attribute):

```
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
```

```
                    item_data RECORD LIKE items.*,
                    it_image STRING,
                    it_count INTEGER
                END RECORD
...
INPUT ARRAY a_items FROM sr.*
   ...
```

When using a static array, the initial number of rows is defined by the COUNT attribute and the size of the array determines how many rows can be inserted. When using a dynamic array, the initial number of rows is defined by the number of elements in the dynamic array (the COUNT attribute is ignored), and the maximum rows is unlimited. For both static and dynamic arrays, the maximum number of rows the user can enter can be defined with the MAXCOUNT attribute.

The FROM clause binds the screen records in the screen array to the program records of the program array. The form can include other fields that are not part of the specified screen array, but the number of member variables in each record of the program array must equal the number of fields in each row of the screen array. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

The variables of the record array are the interface to display data or to get the user input through the INPUT ARRAY instruction. Always use the variables if you want to change some field values by program. When using the UNBUFFERED attribute, the instruction is sensitive to program variable changes. If you need to display new data during the INPUT ARRAY execution, use the UNBUFFERED attribute and assign the values to the program array row; the runtime system will automatically display the values to the screen:

```
INPUT ARRAY p_items FROM s_items.*
      ATTRIBUTES(UNBUFFERED)
  ON CHANGE code
    IF p_items[arr_curr()].code = "A34" THEN
      LET p_items[arr_curr()].desc = "Item A34"
    END IF
END INPUT
```

The runtime system adapts input and display rules to the data type of the array record members. If a member is declared with the DEFINE LIKE instruction and uses a column defined as SERIAL / SERIAL8 / BIGSERIAL, the runtime system will treat the field as if it was defined with the NOENTRY attribute in the form file. Since values of serial columns are automatically generated by the database server, no user input is required for such fields.

The default order in which the focus moves from field to field in the screen array is determined by the declared order of the corresponding member variables, in the array of the record definition. The program OPTIONS instruction can also change the behavior of the INPUT ARRAY instruction, with the INPUT WRAP or FIELD ORDER FORM options.

By default the INPUT ARRAY instruction clears the program array when starting, unless you specify the WITHOUT DEFAULTS keywords or option. With this option, the dialog displays the program array rows in the screen fields. Unlike the INPUT dialog, the column default values defined in the form specification file with the DEFAULT attribute or in the database schema files are always used when a new row is inserted in the list.

If the program array has the same structure as a database table (this is the case when the array is defined with a DEFINE LIKE clause), you may not want to display/use some of the columns. You can achieve this by using PHANTOM fields in the screen array definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

INPUT ARRAY instruction configuration

This section describes the options that can be specified in the ATTRIBUTES clause of the INPUT ARRAY instruction. The options of the ATTRIBUTES clause override all default attributes and temporarily override

any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. With the INPUT ARRAY statement, the INVISIBLE attribute is ignored.

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help the INPUT ARRAY dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

WITHOUT DEFAULTS option

The WITHOUT DEFAULT clause defines whether the program array elements are populated (and to be displayed) when the dialog begins. Once the dialog is started, existing rows are always handled as records to be updated in the database (i.e. WITHOUT DEFAULTS=TRUE), while newly created rows are handled as records to be inserted in the database (i.e. WITHOUT DEFAULTS=FALSE). In other words, the REQUIRED and DEFAULT attributes defined in the form are only used for new created rows.

It is unusual to implement an INPUT ARRAY with no WITHOUT DEFAULTS option, because the data of the program variables would be cleared and the list empty. So, you typically use the WITHOUT DEFAULT clause in INPUT ARRAY. In a singular INPUT ARRAY, the default is WITHOUT DEFAULTS=FALSE.

FIELD ORDER FORM option

By default, the form tabbing order is defined by the variable list in the binding specification. You can control the tabbing order by using the FIELD ORDER FORM attribute. When this attribute is used, the tabbing order is defined by the TABINDEX attribute of the form items. With FIELD ORDER FORM, if you jump from one field to a another with the mouse, the BEFORE FIELD / AFTER FIELD triggers of intermediate fields are not executed (actually, the Dialog.fieldOrder FGLPROFILE entry is ignored.)

If the form uses a TABLE container, the front-end resets the tab indexes when the user moves columns around. This way, the visual column order always corresponds to the input tabbing order. The order of the columns in an editable list can be important; you may want to freeze the table columns with the UNMOVABLECOLUMNS attribute.

UNBUFFERED option

The UNBUFFERED attribute indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the traditional "buffered" mode.

When using the traditional " buffered" mode, program variable changes are not automatically displayed to form fields; You need to execute a DISPLAY TO or DISPLAY BY NAME. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of the current field is to use GET_FLDBUF().

If the "unbuffered" mode is used, program variables and form fields are automatically synchronized. You don't need to display explicitly values with a DISPLAY TO or DISPLAY BY NAME. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable.

COUNT option

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. You can also use the SET_COUNT() built-in function, but it is supported for backward compatibility only. The COUNT option is ignored when using a dynamic array. If you specify the COUNT attribute, the WITHOUT DEFAULTS option is not required because it is implicit. If the COUNT attribute is greater than MAXCOUNT, the runtime system will take MAXCOUNT as the actual number of rows. If the value of COUNT is negative or zero, it defines an empty list.

MAXCOUNT option

The MAXCOUNT attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit of the total number of rows the user can enter, when using both static or dynamic arrays.

When binding a static array, MAXCOUNT is used as upper limit if it is lower or equal to the actual declared static array size. If MAXCOUNT is greater than the array size, the size of the static array is used as the upper limit. If MAXCOUNT is lower than the COUNT attribute (or to the SET_COUNT() parameter), the actual number of rows in the array will be reduced to MAXCOUNT.

When binding a dynamic array, the user can enter an infinite number of rows unless the MAXCOUNT attribute is used. If MAXCOUNT is lower than the actual size of the dynamic array, the number of rows in the array will be reduced to MAXCOUNT.

If MAXCOUNT is negative or equal to zero, the user cannot insert rows.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the accept default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using EXIT INPUT.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an ON ACTION close control block to create an explicit action.

APPEND ROW option

The APPEND ROW attribute can be set to FALSE to avoid the append default action, and deny the user to add rows at the end of the list. If APPEND ROW =FALSE, it is still possible to insert rows in the middle of the list. Use the INSERT ROW attribute to disallow the user from inserting rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

INSERT ROW option

The INSERT ROW attribute can be set to FALSE to avoid the insert default action, and deny the user to insert new rows in the middle of the list. However, even if INSERT ROW is FALSE, it is still possible to append rows at the end of the list. Use the APPEND ROW attribute to disallow the user from appending rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

DELETE ROW option

The DELETE ROW attribute can be set to FALSE to avoid the delete default action, and deny the user to remove rows from the list.

AUTO APPEND option

By default, an INPUT ARRAY controller creates a temporary row when needed (for example, when the user deletes the last row of the list, an new row will be automatically created). You can prevent this default behavior by setting the AUTO APPEND attribute to FALSE. When this attribute is set to FALSE, the only way to create a new temporary row is to execute the append action.

If both the APPEND ROW and INSERT ROW attributes are set to FALSE, the dialog automatically behaves as if AUTO APPEND equals FALSE.

KEEP CURRENT ROW option

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

Default actions in INPUT ARRAY

When an INPUT ARRAY instruction executes, the runtime system creates a set of default actions.

According the invoked default action, field validation occurs and different INPUT ARRAY control blocks are executed.

This table lists the default actions created for this dialog:

Table 274: Default actions for INPUT ARRAY

Default action	Description
accept	Validates the INPUT ARRAY dialog (validates fields and leaves the dialog) *Creation can be avoided with* ACCEPT *attribute.*
cancel	Cancels the INPUT ARRAY dialog (no validation, INT_FLAG is set to TRUE) *Creation can be avoided with* CANCEL *attribute.*
close	By default, cancels the INPUT ARRAY dialog (no validation, INT_FLAG is set to TRUE) Default action view is hidden. See Implementing the close action on page 1337.
insert	Inserts a new row before current row. *Creation can be avoided with* INSERT ROW = FALSE *attribute.*
append	Appends a new row at the end of the list. *Creation can be avoided with* APPEND ROW = FALSE*attribute.*
delete	Deletes the current row. *Creation can be avoided with* DELETE ROW = FALSE*attribute.*
help	Shows the help topic defined by the HELP clause. *Only created when a* HELP*clause is defined.*
nextrow	Moves to the next row in a list displayed in one row of fields.

Default action	Description
	Only created if `DISPLAY ARRAY` *used with a screen record having only one row.*
`prevrow`	Moves to the previous row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`firstrow`	Moves to the first row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`lastrow`	Moves to the last row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *used with a screen record having only one row.*
`find`	Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value. *Only created if the context allows built-in find.*
`findnext`	Seeks to the next row matching the value entered during the fglfind dialog. *Only created if the context allows built-in find.*

The insert, append, delete, accept and cancel default actions can be avoided with dialog control attributes:

```
INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, CANCEL=FALSE, ... )
   ...
```

INPUT ARRAY control blocks
INPUT ARRAY control blocks execution order

This table shows the order in which the runtime system executes the control blocks in the `INPUT ARRAY` instruction, according to the user action:

Table 275: Control block execution order for INPUT ARRAY

Context / User action	Control Block execution order
Entering the dialog	**1.** `BEFORE INPUT` **2.** `BEFORE ROW` **3.** `BEFORE FIELD`
Moving to a different row from field A to field B	**1.** `ON CHANGE` (if value has changed for field A) **2.** `AFTER FIELD` (for field A in the row you leave) **3.** `AFTER INSERT` (if the row you leave was inserted or appended) or

Context / User action	Control Block execution order
	`ON ROW CHANGE` (if values have changed in the row you leave) **4.** `AFTER ROW` (for the row you leave) **5.** `BEFORE ROW` (the new current row) **6.** `BEFORE FIELD` (for field B in the new current row)
Moving from field A to field B in the same row	**1.** `ON CHANGE` (if value has changed for field A) **2.** `AFTER FIELD` (for field A) **3.** `BEFORE FIELD` (for field B)
Deleting a row	**1.** `BEFORE DELETE` (for the row to be deleted) **2.** `AFTER DELETE` (for the deleted row) **3.** `AFTER ROW` (for the deleted row) **4.** `BEFORE ROW` (for the new current row) **5.** `BEFORE FIELD` (field in the new current row)
Inserting a new row between rows	**1.** `ON CHANGE` (if value has changed in the field you leave) **2.** `AFTER FIELD` (for the row you leave) **3.** `AFTER INSERT` (if the row you leave was inserted or appended) or `ON ROW CHANGE` (if values have changed in the row you leave) **4.** `AFTER ROW` (for the row you leave) **5.** `BEFORE INSERT` (for the new created row) **6.** `BEFORE FIELD` (for the new created row)
Appending a new row at the end	**1.** `ON CHANGE` (if value has changed in the current field) **2.** `AFTER FIELD` (for the row you leave) **3.** `AFTER INSERT` (if the row you leave was inserted or appended) or `ON ROW CHANGE` (if values have changed in the row you leave) **4.** `AFTER ROW` (for the row you leave) **5.** `BEFORE ROW` (for the new created row) **6.** `BEFORE INSERT` (for the new created row) **7.** `BEFORE FIELD` (for the new created row)
Validating the dialog	**1.** `ON CHANGE` **2.** `AFTER FIELD` **3.** `AFTER INSERT` (if the current row was inserted or appended) or

Context / User action	Control Block execution order
	`ON ROW CHANGE` (if values have changed in the current row) **4.** `AFTER ROW` **5.** `AFTER INPUT`
Canceling the dialog	**1.** `AFTER ROW` **2.** `AFTER INPUT`

BEFORE INPUT block

BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular `INPUT`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `BEFORE INPUT` is only executed once when the dialog is started.

The `BEFORE INPUT` block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
INPUT BY NAME cust_rec.* ...
   BEFORE INPUT
       MESSAGE "Input customer information"
       CALL DIALOG.setActionActive("check_info", is_super_user() )
       CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
   ...
```

The fields are initialized with the defaults values before the `BEFORE INPUT` block is executed. When the `INPUT` instruction uses the `WITHOUT DEFAULTS` option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the `DEFAULT` attributes of the form fields are used.

Use the `NEXT FIELD` control instruction in the `BEFORE INPUT` block, to jump to a specific field when the dialog starts.

BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE INPUT` block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not invoke the `BEFORE INPUT` block.

`BEFORE INPUT` is executed after the `BEFORE DIALOG` block andbefore the `BEFORE ROW`, `BEFORE FIELD` blocks.

In this example, the `BEFORE INPUT` block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
   BEFORE INPUT
       CALL DIALOG.setActionActive("validate_order", TRUE)
```

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular `INPUT`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `AFTER INPUT` is only executed once when dialog ends.

The AFTER INPUT block is executed after the user has validated or canceled the INPUT or INPUT ARRAY dialog with the accept or cancel default actions, or when the ACCEPT INPUT instruction is executed.

The AFTER INPUT block is not executed when the EXIT INPUT instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules depending from several fields. If the values entered by the user do not satisfy the constraints, use the NEXT FIELD instruction to force the dialog to continue. The CONTINUE INPUT instruction can be used instead of NEXT FIELD, when no particular field has to be select.

Before checking the validation rules, make sure that the INT_FLAG variable is FALSE: in case if the user cancels the dialog, the validation rules must be skipped.

```
INPUT BY NAME cust_rec.*
   WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
  ...

  AFTER INPUT
    IF NOT INT_FLAG THEN
       IF cust_rec.cust_address IS NOT NULL
          AND cust_rec.cust_zipcode IS NULL THEN
          ERROR "Address is incomplete, enter a zipcode."
          NEXT FIELD zipcode
       END IF
    END IF
END INPUT
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER INPUT block is executed when the focus is lost by a group of fields driven by an INPUT or INPUT ARRAY sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the AFTER INPUT block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER INPUT control block will keep the focus in the group of fields. Within an INPUT ARRAY sub-dialog, NEXT FIELD will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the AFTER INPUT block is used to validate data and disable an action that can only be used in the current group:

```
INPUT BY NAME p_order.*
   AFTER INPUT
      IF NOT check_order_data(DIALOG) THEN
        NEXT FIELD CURRENT
      END IF
      CALL DIALOG.setFieldActive("validate_order", FALSE)
```

BEFORE ROW block

BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular `DISPLAY ARRAY`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `BEFORE ROW` block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the `BEFORE ROW` block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, `BEFORE ROW` will be executed for the current row, but only if there are data rows in the array.

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the current row.

In this example, the `BEFORE ROW` block gets the new row number and displays it in a message:

```
DISPLAY ARRAY ...
   ...
   BEFORE ROW
     MESSAGE "We are on row # ", arr_curr()
   ...
```

BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE ROW` block is executed when a `DISPLAY ARRAY` or `INPUT ARRAY` list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the `BEFORE ROW` block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the `BEFORE DISPLAY` or `BEFORE INPUT` blocks instead.

In `DISPLAY ARRAY`, `BEFORE ROW` is executed after the `BEFORE DISPLAY` block. In `INPUT ARRAY`, `BEFORE ROW` is executed before the `BEFORE INSERT` and `BEFORE FIELD` blocks and after the `BEFORE INPUT` blocks.

When the procedural dialog starts, `BEFORE ROW` will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, `BEFORE ROW` will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular `DISPLAY ARRAY` or `INPUT ARRAY`. In singular dialogs, the `BEFORE ROW` block is always executed when the dialog starts (and there are rows in the array).

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the current row.

In this example the `BEFORE ROW` block displays a message with the current row number:

```
   DISPLAY ARRAY p_items TO s_items.*
     BEFORE ROW
       MESSAGE "We are in items, on row #", DIALOG.getCurrentRow("s_items")
```

ON ROW CHANGE block

The `ON ROW CHANGE` block is executed in a list controlled by an `INPUT ARRAY`, when leaving the current row and when the row has been modified since it got the focus. This is typically used to detect row modification.

The code in ON ROW CHANGE will not be executed when leaving new rows created by the user with the default append or insert action. To detect row creation, you must use the BEFORE INSERT or AFTER INSERT control blocks.

The ON ROW CHANGE block is only executed if at least one field value in the current row has changed since the row was entered, and the modification flag of the field is set. The modified field(s) might not be the current field, and several field values can be changed. Values might have been changed by the user or by the program. The modification flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

ON ROW CHANGE is executed after the AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row that has been changed.

You can, for example, code database modifications (UPDATE) in the ON ROW CHANGE block:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   ON ROW CHANGE
     LET r = DIALOG.getCurrentRow("s_items")
     UPDATE items SET
             items.item_code        = p_items[r].item_code,
             items.item_description = p_items[r].item_description,
             items.item_price       = p_items[r].item_price,
             items.item_updatedate  = TODAY
            WHERE items.item_num = p_items[r].item_num
```

AFTER ROW block

AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER ROW block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A NEXT FIELD instruction executed in the AFTER ROW control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the row that you are leaving.

AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

AFTER ROW is executed after the AFTER FIELD, AFTER INSERT and before AFTER DISPLAY or AFTER INPUT blocks.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the of the row that you are leaving.

For both INPUT ARRAY and DISPLAY ARRAY sub-dialogs, a NEXT FIELD executed in the AFTER ROW control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

AFTER ROW and temporary rows in INPUT ARRAY

> **Important: After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The `AFTER ROW` block will be executed for the temporary row, but `ui.Dialog.getCurrentRow()`/`arr_curr()` will be one row greater than `ui.Dialog.getArrayLength()`/`ARR_COUNT()`. In this case, you should ignore the `AFTER ROW` event. For example, you should not try to execute a `NEXT FIELD` or `CONTINUE INPUT` instruction, nor should you try to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.**

In this example, the `AFTER ROW` block checks the current row index and verifies a variable value to forces the focus to stay in the current row if the value is wrong:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   AFTER ROW
      LET r = DIALOG.getCurrentRow("s_items")
      IF r <= DIALOG.getArrayLength("s_items") THEN
        IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
           ERROR "Item quantity is not valid"
           NEXT FIELD item_quantity'
         END IF
      END IF
```

Another way to handle the case of temporary rows in `AFTER ROW` is to use a flag to know if the `AFTER INSERT` block was executed: The `AFTER INSERT` block is not executed if the temporary row is automatically removed. By setting a first value in `BEFORE INSERT` and changing the flag in `AFTER INSERT`, you can detect if the row was permanently added to the list:

```
INPUT ARRAY p_items FROM s_items.*
  ...
  BEFORE INSERT
    LET op = "T"
    ...
  AFTER INSERT
    LET op = "I"
    ...
  AFTER ROW
    IF op == "I" THEN
      IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
         ERROR "Item quantity is not valid"
         NEXT FIELD item_quantity
        END IF
      WHENEVER ERROR CONTINUE
      INSERT INTO items (item_num, item_name, item_quantity)
                        VALUES ( p_item[arr_curr()].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<0 THEN
         ERROR "Could not insert the record into database!"
         NEXT FIELD CURRENT
      ELSE
         MESSAGE "Record has been inserted successfully"
      END IF
   END IF
...
```

BEFORE INSERT block

The BEFORE INSERT block is executed when a new row ins created in an INPUT ARRAY. You typically use this trigger to set some default values in the new created row. A new row can be created by moving down after the last row, by executing a insert action, or by executing an append action.

The BEFORE INSERT block is executed after the BEFORE ROW block and before the BEFORE FIELD block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the new created row.

To distinguish row insertion from an appended row, compare the current row (DIALOG.getCurrentRow("*screen-array*")) with the total number of rows (DIALOG.getArrayLength("*screen-array*")). If the current row index and the total number of rows correspond, the BEFORE INSERT concerns a temporary row, otherwise it concerns an inserted row.

Row creation can be stopped by using the CANCEL INSERT instruction inside BEFORE INSERT. If possible, it is however better to disable the insert and append actions to prevent the user to execute the actions with DIALOG.setActionActive().

In this example, the BEFORE INSERT block checks if the user can create rows and denies new row creation if needed; otherwise, it sets some default values:

```
INPUT ARRAY p_items FROM s_items.*
    ...
    BEFORE INSERT
      IF NOT user_can_append THEN
         ERROR "You are not allowed to append rows"
         CANCEL INSERT
      END IF
      LET r = DIALOG.getCurrentRow("s_items")
      LET p_items[r].item_num = get_new_serial("items")
      LET p_items[r].item_name = "undefined"
```

AFTER INSERT block

The AFTER INSERT block of INPUT ARRAY is executed when the creation of a new row is validated. In this block, you can for example implement SQL to insert a new row in the database table.

The AFTER INSERT block is executed afterthe AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the new created row.

When the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the AFTER INSERT block is only executed if at least one field was edited. If no data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the AFTER INSERT block.

When executing a NEXT FIELD in the AFTER INSERT block, the dialog will keep the focus in the list and stay in the current row. Use this behavior to implement row input validation and prevent the user from leaving the list or moving to another row. However, this will not cancel the row insertion and will not invoke the BEFORE INSERT / AFTER INSERT triggers again. The only way to keep the focus in the current row after the row was inserted is to execute a NEXT FIELD in the AFTER ROW block.

In this example, the AFTER INSERT block inserts a new row in the database and cancels the operation if the SQL command fails:

```
INPUT ARRAY p_items FROM s_items.*
    ...
    AFTER INSERT
      WHENEVER ERROR CONTINUE
```

```
      INSERT INTO items VALUES
 ( p_items[DIALOG.getCurrentRow("s_items")].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<>0 THEN
         ERROR SQLERRMESSAGE
         CANCEL INSERT
      END IF
```

BEFORE DELETE block

The BEFORE DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, before the row is removed from the list.

You typically code the database table synchronization in the BEFORE DELETE block, by executing a DELETE SQL statement using the primary key of the current row. In the BEFORE DELETE block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.

The BEFORE DELETE block is executed before the AFTER DELETE block.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the row that will be deleted.

The next example uses the BEFORE DELETE block to remove the row from the database table and cancels the deletion operation if an SQL error occurs:

```
INPUT ARRAY p_items FROM s_items.*
  BEFORE DELETE
    LET r = DIALOG.getCurrentRow("s_items")
    WHENEVER ERROR CONTINUE
    DELETE FROM items
       WHERE item_num = p_items[r].item_num
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<>0 VALUES
      ERROR SQLERRMESSAGE
      CANCEL DELETE
    END IF
...
```

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The AFTER DELETE block is executed after the BEFORE DELETE block and before the AFTER ROW block for the deleted row and the BEFORE ROW block of the new current row.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array (except in the special case when deleting the last row). The arr_curr() function or the ui.Dialog.getCurrentRow() method returns the same index as in BEFORE ROW, but it is the index of the new current row. The AFTER ROW block is also executed just after the AFTER DELETE block.

> **Important: When deleting the last row of the list, AFTER DELETE is executed for the delete row, and DIALOG.getCurrentRow() / arr_curr() will be one higher as DIALOG.getArrayLength() / ARR_COUNT(). You should not access a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array. When using a static array, you must ignore the values in the rows after ARR_COUNT().**

Here the AFTER DELETE block is used to re-number the rows with a new item line number (note that DIALOG.getArrayLength() / ARR_COUNT() may return zero):

```
INPUT ARRAY p_items FROM s_items.*
   AFTER DELETE
      LET r = DIALOG.getCurrentRow("s_items")
      FOR i=r TO DIALOG.getArrayLength("s_items")
          LET p_items[i].item_lineno = i
      END FOR
...
```

It is not possible to use the CANCEL DELETE instruction in an AFTER DELETE block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

BEFORE FIELD block

For fields controlled by an INPUT, INPUT ARRAY or by a CONSTRUCT instructions, the BEFORE FIELD block is executed every time the cursor enters into the specified field.

For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
   BEFORE FIELD cust_status
     LET p_cust.cust_comment = NULL
     MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing the first time with a specific form, the BEFORE FIELD block might be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column would have been done in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

ON CHANGE block

The ON CHANGE block can be used to detect that a field changed by user input. The ON CHANGE block is executed if the value has changed since the field got the focus and if the modification flag is set. The ON CHANGE block can only be used for fields controlled by an INPUT or INPUT ARRAY dialog, it is not available in CONSTRUCT.

For editable fields defined as EDIT, TEXTEDIT or BUTTONEDIT, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and if the modification flag is set for the field. You leave the field when you validate the dialog, when you move to another field, or when you move to another row in an INPUT ARRAY. However, if the text edit field is

defined with the COMPLETER attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

```
   ON CHANGE order_checked -- Defined as CHECKBOX
      CALL setup_dialog(DIALOG)
```

If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing the value of the current field by program in an ON ACTION block, the ON CHANGE block will be executed when leaving the field if the value is different from the reference value and if the modification flag is set (after previous user input or when the touched flag has been changed by program).

When using the NEXT FIELD instruction, the comparison value is reassigned as if the user had leaved and reentered the field. Therefore, when using NEXT FIELD in ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be invoked again if the value is different from the reference value. This denies to do field validation in ON CHANGE blocks: you must do validations in AFTER FIELD blocks and/or AFTER INPUT blocks.

AFTER FIELD block

In dialog parts driven by a simple INPUT, INPUT ARRAY or by a CONSTRUCT sub-dialog, the AFTER FIELD block is executed every time the focus leaves the specified field. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, based on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT or ACCEPT CONTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
   AFTER FIELD item_quantity
     IF p_item.item_quantity <= 0 THEN
        ERROR "Item quantity cannot be negative or zero"
        LET p_item.item_quantity = 0
        NEXT FIELD item_quantity
     END IF
```

INPUT ARRAY interaction blocks

ON ACTION block

The `ON ACTION` *`action-name`* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
  ON ACTION action-name
     instruction
     ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
   ON ACTION zoom
      CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as `INPUT`, `INPUT ARRAY` and `CONSTRUCT`, if an action is specific to a field, add the `INFIELD` clause to have the action automatically enabled when the corresponding field gets the focus:

```
   ON ACTION zoom INFIELD cust_city
      CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the `ON ACTION` handler needs to define its own attributes at the program level. This can be done by adding the `ATTRIBUTES()` clause of `ON ACTION`:

```
   ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
      CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The `ON IDLE` *`seconds`* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of `ON IDLE` must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

Important: The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable

will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An `ON KEY (key-name)` block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The `ON KEY` block is supported for backward compatibility with TUI mode applications.

An `ON KEY` block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, `ON KEY(F5,F6)` creates two actions with the names `f5` and `f6`. Each action object will get an `ACCELERATORNAME` assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON SORT block

The `ON SORT` interfaction block can be used to detect when rows have to be sorted in a `DISPLAY ARRAY` or `INPUT ARRAY` dialog.

`ON SORT` is used in two different contexts:

1. In a regular `DISPLAY ARRAY` / `INPUT ARRAY` dialog (not using paged mode), the `ON SORT` trigger can be used to detect that a list sort was performed. In this case, the (visual) sort is already done by the runtime system and the `ON SORT` block is only used to execute post-sort tasks, such as displaying current row information, by using arrayToVisualIndex() dialog method. It is also possible to get the sort column and order with the `getSortKey()` and `getSortSelection()` dialog methods.
2. In a `DISPLAY ARRAY` using paged mode (`ON FILL BUFFER`), built-in row sorting is not available because data is provided by pages. Use the `ON SORT` trigger to detect a sort request and perform a new SQL query to re-order the rows. In this case, sort column and order is available with the `getSortKey()` and `getSortSelection()` dialog methods. See Populating a DISPLAY ARRAY on page 1372.

ON TIMER block

The `ON TIMER seconds` clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

INPUT ARRAY control instructions

ACCEPT INPUT instruction

The `ACCEPT INPUT` instruction validates the `INPUT`instruction and exits the `INPUT ARRAY` instruction if no error is raised. The `AFTER FIELD`, `ON CHANGE`, etc. control blocks will be executed. Statements after the `ACCEPT INPUT` will not be executed.

Input field validation is a process that does several successive validation tasks, as listed here:

1. The current field value is checked, according to the program variable data type (for example, the user must input a valid date in a DATE field).
2. `NOT NULL` field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.
3. `INCLUDE` field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.
4. `REQUIRED` field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be "touched" by the user or by program. If the field was not edited during the dialog, the constraint is not satisfied.

If a field does not satisfy one of these constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

EXIT INPUT instruction

Use the `EXIT INPUT` to terminate the `INPUT ARRAY` instruction and resume the program execution at the instruction following the `INPUT ARRAY` block.

```
ON ACTION leave_now
   EXIT INPUT
```

When leaving the `INPUT ARRAY` instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

CANCEL DELETE instruction

In a list controlled by an `INPUT ARRAY`, row deletion can be canceled by using the `CANCEL DELETE` instruction in the `BEFORE DELETE` block. Using this instruction in a different place will generate a compilation error.

When the `CANCEL DELETE` instruction is executed, the current `BEFORE DELETE` block is terminated without any other trigger execution (no `BEFORE ROW` or `BEFORE FIELD` is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```
   BEFORE DELETE
      IF user_can_delete() == FALSE THEN
```

```
         ERROR "You are not allowed to delete rows"
         CANCEL DELETE
      END IF
```

The instructions that appear after `CANCEL DELETE` will be skipped.

If the row deletion condition is known before the delete action occurs, disable the delete action to prevent the user from performing a delete row action with the `DIALOG.setActionActive()` method:

```
   CALL DIALOG.setActionActive("delete", FALSE)
```

It is also possible to prevent the user from deleting rows with the `DELETE ROW = FALSE` option in the `ATTRIBUTE` clause.

CANCEL INSERT instruction

In a list controlled by an `INPUT ARRAY`, row creation can be canceled by the program with the `CANCEL INSERT` instruction. This instruction can only be used in the `BEFORE INSERT` and `AFTER INSERT` control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after `CANCEL INSERT` will be skipped.

If the row creation condition is known before the insert/append action occurs, disable the insert and/or append actions to prevent the user from creating new rows, with `DIALOG.setActionActive()`:

```
   CALL DIALOG.setActionActive("insert", FALSE)
   CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list, when moving down after the last row. To prevent row creation completely, use the `INSERT ROW = FALSE` and `APPEND ROW =FALSE` options in the `ATTRIBUTE` clause of `INPUT ARRAY`, or combine with the `AUTO APPEND = FALSE` attribute.

CANCEL INSERT in BEFORE INSERT

A `CANCEL INSERT` executed inside a `BEFORE INSERT` block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The `BEFORE INSERT` block is terminated (further instructions are skipped).
3. The `BEFORE ROW` and `BEFORE FIELD` triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```
   BEFORE INSERT
      IF NOT user_can_insert THEN
         ERROR "You are not allowed to insert rows"
         CANCEL INSERT
      END IF
```

Executing `CANCEL INSERT` in `BEFORE INSERT` will also cancel a temporary row creation, except when there are no more rows in the list. In this case, `CANCEL INSERT` will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). You can prevent automatic temporary row creation with the `AUTO APPEND=FALSE` attribute. If `AUTO APPEND=FALSE` and a `CANCEL INSERT` is executed in `BEFORE INSERT` (user has invoked an append action), the temporary row will be deleted and list will remain empty if it was the last row.

CANCEL INSERT in AFTER INSERT

A `CANCEL INSERT` executed inside an `AFTER INSERT` block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The `AFTER INSERT` block is terminated (further instructions are skipped).
3. The `BEFORE ROW` and `BEFORE FIELD` triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```
   AFTER INSERT
      WHENEVER ERROR CONTINUE
      LET r = DIALOG.getCurrentRow("s_items")
      INSERT INTO items VALUES ( p_items[r].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<>0 THEN
         ERROR SQLERRMESSAGE
         CANCEL INSERT
      END IF
```

CONTINUE INPUT instruction

`CONTINUE INPUT` skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. If this instruction is called in a control block that is not `AFTER INPUT`, further control blocks might be executed according to the context. Actually, `CONTINUEINPUT` just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of `GOTO end_of_control_block`). However, when executed in `AFTER INPUT`, the focus returns to the current row and current field in the list, giving the user another chance to enter data in that field. In this case the `BEFORE ROW` and `BEFORE FIELD` triggers will be invoked.

In this example, an `ON ACTION` block gives control back to the dialog, skipping all instructions after line 04:

```
ON ACTION zoom
   IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
      ERROR "Zoom window cannot be opened."
      CONTINUE INPUT
   END IF
   IF p_cust.cust_address IS NULL THEN
      ...
```

You can also use the `NEXT FIELD` control instruction to give the focus to a specific field and force the dialog to continue. However, unlike `CONTINUE INPUT`, the `NEXT FIELD` instruction will also skip the further control blocks that are normally executed.

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The `NEXT FIELD` *`field-name`* instruction gives the focus to the specified field and forces the dialog to stay in that field.

This instruction can be used to control field input, in `BEFORE FIELD`, `ON CHANGE` or `AFTER FIELD` blocks, it can also force a `DISPLAY ARRAY` or `INPUT ARRAY` to stay in the current row when `NEXT FIELD` is used in the `AFTER ROW` block.

If it exists, the `BEFORE FIELD` block of the corresponding field is executed.

The purpose of the NEXT FIELD instruction is give the focus to an editable field. Make sure that the field specified in NEXT FIELD is active, or use NEXT FIELD CURRENT. Non-editable fields are fields defined with the NOENTRY attribute, fields disabled at runtime with DIALOG.setFieldActive(), or fields using a widget that does not allow input, such as a LABEL.

Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("*field-name*") method to register a field, for example when the name is not known at compile time. However, this method only registers the field: It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.

Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog. Form fields are bound to program variables with the binding clause of dialog instruction (INPUT *variable-list* FROM *field-list*, INPUT BY NAME *variable-list*, CONSTRUCT BY NAME *sql* ON *column-list*,CONSTRUCT *sql* ON *column-list* FROM *field-list*, INPUT ARRAY *array-name* FROM *screen-array*.*).

The field name specification can be any of the following:

- *field-name*
- *table-name.field-name*
- *screen-record-name.field-name*
- FORMONLY.*field-name*

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog according to the field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using INPUT BY NAME *variable-list*), the field prefix must be the database table name (or FORMONLY) used in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT *variable-list* FROM *screen-record.* / field-list-with-screen-record-prefix* or INPUT ARRAY *array-name* FROM *screen-array*.*), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD PREVIOUS will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable INPUT / INPUT ARRAY / CONSTRUCT field

Non-editable fields are fields defined with the NOENTRY attribute, fields disabled with ui.Dialog.setFieldActive("*field-name*", FALSE), or fields using a widget that does not allow input, such as a LABEL.

If a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog. Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; Use NEXT FIELD CURRENT instead.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD *field-name* instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW and AFTER INPUT/DISPLAY/CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT/DISPLAY/CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user. This behavior allows you to implement data input rules:

```
AFTER ROW
  IF NOT int_flag AND arr_count()<=arr_curr() THEN
     IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
        ERROR "Amount of line exceeds max value."
        NEXT FIELD item_count
     END IF
  END IF
```

CLEAR instruction in dialogs

The CLEAR *field-list* and CLEAR SCREEN ARRAY *screen-array.** instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to

the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFEFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated to the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

Examples

Example 1: INPUT ARRAY with empty record list

Form definition file (custlist.per):

```
SCHEMA shop
LAYOUT
TABLE
{
 Id       First name   Last name
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
[f001    |f002        |f003        ]
}
END
END
TABLES
  customer
END
ATTRIBUTES
  f001 = customer.id ;
  f002 = customer.fname ;
  f003 = customer.lname, NOT NULL, REQUIRED ;
END
INSTRUCTIONS
  SCREEN RECORD sr_cust( customer.* );
END
```

Program source code:

```
SCHEMA shop

MAIN
  DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*

  OPEN FORM f FROM "custlist"
  DISPLAY FORM f

  INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*

END MAIN
```

Example 2: INPUT ARRAY using a static array

The form definition file is the same as in Example 1.

```
SCHEMA shop

MAIN

  DEFINE custarr ARRAY[100] OF RECORD LIKE customer.*
  DEFINE allow_insert, size INTEGER
```

```
  LET custarr[1].id = 1
  LET custarr[1].fname = "John"
  LET custarr[1].lname = "SMITH"
  LET custarr[2].id = 2
  LET custarr[2].fname = "Mike"
  LET custarr[2].lname = "STONE"
  LET size = 2
  LET allow_insert = TRUE

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
      ATTRIBUTES (COUNT=size, MAXCOUNT=50, UNBUFFERED, INSERT
 ROW=allow_insert)
      BEFORE INPUT
        MESSAGE "Editing the customer table"
      BEFORE INSERT
        IF arr_curr()=4 THEN
           CANCEL INSERT
        END IF
      BEFORE FIELD fname
        MESSAGE "Enter First Name"
      BEFORE FIELD lname
        MESSAGE "Enter Last Name"
      AFTER FIELD lname
        IF custarr[arr_curr()].lname IS NULL THEN
           LET custarr[arr_curr()].fname = NULL
        END IF
  END INPUT

END MAIN
```

Example 3: INPUT ARRAY using a dynamic array

The form definition file is the same as in .Example 1

```
SCHEMA shop

MAIN

  DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
  DEFINE counter INTEGER

  FOR counter = 1 TO 500
    LET custarr[counter].id = counter
    LET custarr[counter].fname = "ff"||counter
    LET custarr[counter].lname = "NNN"||counter
  END FOR

  OPEN FORM f FROM "custlist"
  DISPLAY FORM f

  INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
                ATTRIBUTES ( UNBUFFERED )
      ON ROW CHANGE
        MESSAGE "Row #"||arr_curr()||" has been updated."
  END INPUT

END MAIN
```

Example 4: INPUT ARRAY updating the database table

The form definition file is the same as in Example 1.

```
SCHEMA shop

MAIN

 DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*

 DEFINE op CHAR(1)
 DEFINE i INTEGER

 DATABASE shop

 OPEN FORM f1 FROM "custlist"
 DISPLAY FORM f1

 DECLARE c1 CURSOR FOR
   SELECT id, fname, lname FROM customer ORDER BY id
 LET i = 1
 FOREACH c1 INTO custarr[i].*
    LET i = i + 1
 END FOREACH
 CALL custarr.deleteElement(custarr.getLength())

 INPUT ARRAY custarr FROM sr_cust.*
                 ATTRIBUTES(WITHOUT DEFAULTS, UNBUFFERED)

      BEFORE DELETE
        IF op == "N" THEN -- No real SQL delete for new inserted rows
           IF NOT mbox_yn("List",
                "Are you sure you want to delete this record?",
                "question") THEN
             CANCEL DELETE -- Keeps row in list
           END IF
           WHENEVER ERROR CONTINUE
           DELETE FROM customer
                  WHERE ID = custarr[arr_curr()].id
           WHENEVER ERROR STOP
           IF SQLCA.SQLCODE<0 THEN
             ERROR "Could not delete the record from database!"
             CANCEL DELETE -- Keeps row in list
           END IF
        END IF

      AFTER DELETE
        IF op == "N" THEN
           MESSAGE "Record has been deleted successfully"
        ELSE
           LET op = "N"
        END IF

      AFTER FIELD fname
        IF custarr[arr_curr()].fname MATCHES "*@#$%^&()*" THEN
           ERROR "This field contains invalid characters"
           NEXT FIELD CURRENT
        END IF

      ON ROW CHANGE
        -- Warning: ON ROW CHANGE can occur if the SQL INSERT failed...
        IF op != "I" THEN LET op = "M" END IF

      BEFORE INSERT
```

```
        LET op = "T"
        -- (not the best way to get a unique sequence number!)
        SELECT MAX(ID)+1 INTO custarr[arr_curr()].id FROM customer
        IF custarr[arr_curr()].id IS NULL THEN
          LET custarr[arr_curr()].id = 1
        END IF

      AFTER INSERT
        LET op = "I"

      BEFORE ROW
        LET op = "N"

      AFTER ROW
        IF int_flag THEN EXIT INPUT END IF
        IF op == "M" OR op == "I" THEN
          IF custarr[arr_curr()].fname IS NULL
            OR custarr[arr_curr()].lname IS NULL
            OR custarr[arr_curr()].fname ==
             custarr[arr_curr()].lname THEN
              ERROR "First name and last name are equal..."
              NEXT FIELD fname
           END IF
        END IF
        IF op == "I" THEN
          WHENEVER ERROR CONTINUE
          INSERT INTO customer (id, fname, lname)
                      VALUES ( custarr[arr_curr()].* )
          WHENEVER ERROR STOP
          IF SQLCA.SQLCODE<0 THEN
            ERROR "Could not insert the record into database!"
            NEXT FIELD CURRENT
          ELSE
            MESSAGE "Record has been inserted successfully"
          END IF
        END IF
        IF op == "M" THEN
          WHENEVER ERROR CONTINUE
            UPDATE customer SET
                 fname = custarr[arr_curr()].fname,
                 lname = custarr[arr_curr()].lname
              WHERE id = custarr[arr_curr()].id
            WHENEVER ERROR STOP
              IF SQLCA.SQLCODE<0 THEN
                ERROR "Could not update the record in database!"
                NEXT FIELD CURRENT
              ELSE
                MESSAGE "Record has been updated successfully"
              END IF
        END IF

 END INPUT

END MAIN

FUNCTION mbox_yn(title,message,icon)
  DEFINE title, message, icon STRING
  DEFINE r SMALLINT
  MENU title ATTRIBUTES(STYLE='dialog',IMAGE=icon,COMMENT=message)
    COMMAND "Yes" LET r=TRUE
    COMMAND "No"  LET r=FALSE
  END MENU
  RETURN r
```

```
END FUNCTION
```

Query by example (CONSTRUCT)

The CONSTRUCT instruction implements database query criteria input in an application form.

- Understanding the CONSTRUCT instruction on page 1128
- Syntax of CONSTRUCT instruction on page 1128
- CONSTRUCT programming steps on page 1130
- Using query by example on page 1131
  - Query operators in CONSTRUCT on page 1131
  - CONSTRUCT instruction configuration on page 1133
  - Default actions IN CONSTRUCT on page 1134
  - CONSTRUCT control blocks on page 1134
  - CONSTRUCT interaction blocks on page 1138
  - CONSTRUCT control instructions on page 1139
- Examples on page 1142
  - Example 1: CONSTRUCT with binding by field position on page 1142
  - Example 2: CONSTRUCT with binding by field name on page 1143

Understanding the CONSTRUCT instruction

The CONSTRUCT instruction provides database query by example. Query by example enables a user to query a database by specifying values (or ranges of values) for screen fields that correspond to the database columns. The runtime system converts the query values entered by the user into a boolean SQL condition that can be used in the WHERE clause of a prepared SELECT statement.

The CONSTRUCT statement produces an SQL condition corresponding to all search criteria that a user specifies in the fields. The instruction fills a character variable with that SQL condition, and you can use the content of this variable to create the WHERE clause of a SELECT statement. The SELECT statement must be executed with the dynamic SQL management instructions PREPARE or DECLARE FROM:

The CONSTRUCT instruction uses the data types of the form field to verify user input and to produce the SQL condition.

> **Important:** The SQL condition is generated according to the current database session, which defines the type of the database server. Therefore, the program must be connected to a database server before entering the CONSTRUCT block. The generated SQL condition is specific to the database server and may not be used with other types of database servers.

If no criteria were entered, the string '1=1' is assigned to the string variable. This is a boolean SQL expression that always evaluates to true so that all rows are returned.

The CONSTRUCT dialog activates the current form. This is the form most recently displayed or, if you are using more than one window, the form currently displayed in the current window. When the CONSTRUCT statement completes execution, the form is cleared and deactivated.

By default the screen field tabbing order is defined by the order of the field names in the FROM clause; by default this is the list of column names in the ON clause when no FROM clause is specified. If needed, change the field tabbing order with the FIELD ORDER FORM option and TABINDEX field attributes.

Syntax of CONSTRUCT instruction

The CONSTRUCT instruction provides database query by example, producing a WHERE condition for SELECT.

Syntax

```
CONSTRUCT { BY NAME variable ON column-list
```

```
          | variable ON column-list FROM field-list
          }
    [ ATTRIBUTES ( { display-attribute
                   | control-attribute }
                   [,...] ) ]
    [ HELP help-number ]
[ dialog-control-block
  [...]
END CONSTRUCT ]
```

where *column-list* defines a list of database columns as:

```
{ column-name
| table-name.*
|  table-name. column-name
} [,...]
```

where *field-list* defines a list of fields with one or more of:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *dialog-control-block* is one of:

```
{ BEFORE CONSTRUCT
| AFTER CONSTRUCT
| BEFORE FIELD field-spec [,...]
| AFTER FIELD field-spec [,...]
| ON IDLE seconds
| ON TIMER seconds
| ON ACTION action-name
             [ INFIELD field-spec ]
             [ ATTRIBUTES ( action-attributes-construct ) ]
| ON KEY ( key-name [,...] )
}
    dialog-statement
    [...]
```

where *action-attributes-construct* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *dialog-statement* is one of:

```
{ statement
| NEXT FIELD { NEXT | PREVIOUS | field-spec}
| CONTINUE CONSTRUCT
| EXIT CONSTRUCT
}
```

where *field-spec* identifies a unique field with one of:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
}
```

where *display-attribute* is:

```
{ BLACK | BLUE | CYAN | GREEN
| MAGENTA | RED | WHITE | YELLOW
| BOLD | DIM | INVISIBLE | NORMAL
| REVERSE | BLINK | UNDERLINE
}
```

where *control-attribute* is:

```
{ ACCEPT [ = boolean ]
| CANCEL [ = boolean ]
| FIELD ORDER FORM
| HELP = help-number
| NAME = "dialog-name"
}
```

1. *variable* is the variable that will contain the SQL condition built by the `CONSTRUCT` instruction.
2. *column-name* is the identifier of a database column of the current form.
3. *table-name* is the identifier of a database table of the current form.
4. *field-name* is the identifier of a field of the current form.
5. *screen-array* is the screen array that will be used in the current form.
6. *line* is a screen array line in the form.
7. *screen-record* is the identifier of a screen record of the current form.
8. *help-number* is an integer that allows you to associate a help message number with the instruction.
9. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
10. *dialog-name* is the identifier of the dialog.
11. *action-name* identifies an action that can be executed by the user.
12. *seconds* is an integer literal or variable that defines a number of seconds.
13. *statement* is any instruction supported by the language.
14. *action-attributes* are dialog-specific action attributes.

CONSTRUCT programming steps

The following steps describe how to implement the `CONSTRUCT` statement:

1. Declare a variable with the `DEFINE` statement, it can be `CHAR`, `VARCHAR` or `STRING`. Prefer `STRING` to avoid any size limitation.
2. Open and display the form, using an `OPEN WINDOW WITH FORM` or the `OPEN FORM` / `DISPLAY FORM` instructions.
3. Set the `INT_FLAG` variable to `FALSE`.
4. Define the `CONSTRUCT` block with the list of form fields to be used for the query by example. If needed, define dialog control blocks to implement rules for the query by example.
5. Inside the `CONSTRUCT` statement, control the behavior of the instruction with `BEFORE CONSTRUCT`, `BEFORE FIELD`, `AFTER FIELD`, `AFTER CONSTRUCT` and `ON ACTION` blocks.
6. After the interaction statement block, test the `INT_FLAG` predefined variable to check if the dialog was canceled (`INT_FLAG=TRUE`) or validated (`INT_FLAG=FALSE`).

If the INT_FLAG variable is TRUE, you should reset it to FALSE to not disturb code that relies on this variable to detect interruption events from the GUI front-end or TUI console.

7. To build the complete SQL statement, concatenate "SELECT ... WHERE" to the string variable that contains the boolean SQL expression produced by CONSTRUCT.
8. Define a database cursor with the DECLARE FROM instruction, by using the SELECT statement.
9. Execute the cursor and fetch the rows found by the database server. You can for example implement a FOREACH loop to fill a program array, to be shown by a DISPLAY ARRAY statement.

Using query by example

To use query by example, you must understand how they work and how to structure the code.

Form field specification in CONSTRUCT

In order to produce an SQL condition, the CONSTRUCT instruction uses a list of database columns that must match form fields for user input. Unlike INPUT, DISPLAY ARRAY and INPUT ARRAY, the CONSTRUCT dialog does not use a program variable for each form field: Only one string variable is required, to hold the SQL condition. Individual field criteria is available in the input buffers (GET_FLDBUF()).

The list of database columns specified in the CONSTRUCT statement will appear in the SQL condition produced.

Binding columns and fields by name

The CONSTRUCT BY NAME *variable* ON *column-list* syntax maps the field names to database column names by name. Form fields are typically defined in the form by following a database schema, specifying the column name and data type.

```
SCHEMA stock
DEFINE where_part STRING
...
CONSTRUCT BY NAME where_part ON cust_name, cust_address
  ...
END CONSTRUCT
```

Binding columns and fields by position

The CONSTRUCT *variable* ON *column-list* FROM *field-list* clause explicitly maps database columns to form fields by position. The form can include other fields that are not part of the specified column list, but the number of variables or record members must equal the number of form fields listed in the FROM clause. Each database column must be of the same (or a compatible) data type as the corresponding form field. When the user enters data, the runtime system checks the entered value against the data type of the form field.

```
DEFINE where_part STRING
...
CONSTRUCT where_part ON cust_name, cust_address
                     FROM field_02, field_04
  ...
END CONSTRUCT
```

Query operators in CONSTRUCT

The CONSTRUCT instruction supports a specific query syntax, using wildcard characters and comparison operators.

The table below lists CONSTRUCT wildcard characters that can be used during a query by example input:

Table 276: CONSTRUCT relational operators

Symbol	Meaning	Data type
value	Use value as is to filter	Any simple type
=	Is NULL	Any simple type
==	Equal to	Any simple type
>	Greater than	Any simple type
>=	Greater than or equal to	Any simple type
<	Less than	Any simple type
<=	Less than or equal to	Any simple type
<> *or* !=	Not equal to	Any simple type
!=	Not NULL	Any simple type
: *or* ..	Range of values	Any simple type
|	List of values	Any simple type
*	Wildcard for any string	Char string types
?	Single-character wildcard	Char string types
[*c1-c2*]	Range of characters	Char string types
[*c1c2...*]	Set of characters	Char string types

Queries based on character types are case sensitive, because SQL is case sensitive, except if the database server is configured to be case-insensitive.

The * (star) and ? (question mark) wildcards are specific to character string type queries, and will generate a MATCHES expression or a LIKE expression, according to the type of database used. When entering a * or ?, the pattern can also contain a character range specification with the square brackets notation [a-z] or [xyz]. A caret (^.) as the first character within the square brackets specifies the logical complement of the set, and matches any character that is not listed. For example, the search value [^AB]* specifies all strings beginning with characters other than A or B.

Some syntaxes can produce an **Error in field** dialog error if the feature is supported by the pattern matching operator of the database server. For example, not all db servers support the [a-z] character range specification in the LIKE pattern.

If you want to search for rows with values containing a * star, a ? question mark or a \ backslash, you must escape the wildcard character with a backslash. Specifying a backslash before another character will have no effect.

Table 277: CONSTRUCT input examples with matching and non matching values

QBE input example	Matching values	Non matching values
100	100	99, 101, NULL
>=100	100, 101, 200	10, 99, NULL
!=100	98, 98, 101, 102	100, NULL
!=	98, 99, 100, 101	NULL
1:100	1, 2 ... 99, 100	0, 101, NULL

QBE input example	Matching values	Non matching values
aaa:yyy	aaa, aab, ab, yy, yyy	zaa, NULL
abc	abc	bc, abcd, Abc, NULL
ABC	ABC	abc, aBC, NULL
abc*	abc, abcd, abcdef	bc, ABC, NULL
*bc	abc, bc	acd, aBC, NULL
?bc	abc, xbc, zbc	aabc, aBC, NULL
*bc?	aaaabc, abcd, bcd	abcdef, bcdef, NULL
[a-z]bc	abc, ebc, zbc	2bc, +bc, Abc, NULL
[^abc]*	deee, feee, zyx, z	azzz, byy, d, NULL
a[bxy]c	abc, axc, ayc	a2c, azc, aBc, NULL
*[xyz]	abcx, eeeez	abcd, eeee, NULL
1|2|35	1, 2, 35	4, 5, 6, NULL
aa|bb|cc	aa, bb, cc	ab, dd, NULL
\\abc*	\abc, \abcdef	abc, NULL
*bc	*bc	abc, bc, NULL
\[?\]	[a], a[b]c, xx[y]zz	a[bb]c, a[]c, NULL

CONSTRUCT instruction configuration

This section describes the options that can be specified in the `ATTRIBUTES` clause of the `CONSTRUCT` instruction. The options of the `ATTRIBUTES` clause override all default attributes and temporarily override any display attributes that the `OPTIONS` or the `OPEN WINDOW` statement specified for these fields. With the `CONSTRUCT` statement, the `INVISIBLE` attribute is ignored.

NAME option

The `NAME` attribute can be used to name the `CONSTRUCT` dialog. This is especially used to identify actions of the dialog.

The clause specifies the number of a to display if the user invokes the help the dialog. The predefined 'help' action is automatically created by the runtime system. You can bind to the 'help' action.

HELP option

`HELP`help message`CONSTRUCT`action views

The `HELP` clause overrides the `HELP` attribute.

FIELD ORDER FORM option

By default, the tabbing order is defined by the variable binding list in the instruction description. You can control the tabbing order by using the `FIELD ORDER FORM` attribute: When this attribute is used, the tabbing order is defined by the `TABINDEX` attribute of the form fields. If this attribute is used, the Dialog.fieldOrder FGLPROFILE entry is ignored.

The `OPTIONS` instruction can also change the behavior of the `INPUT` instruction, with the `INPUT WRAP` or `FIELD ORDER FORM` options.

ACCEPT option

The ACCEPT attribute can be set to FALSE to avoid the automatic creation of the accept default action. This option can be used for example when you want to write a specific validation procedure, by using ACCEPT INPUT.

CANCEL option

The CANCEL attribute can be set to FALSE to avoid the automatic creation of the cancel default action. This is useful for example when you only need a validation action (accept), or when you want to write a specific cancellation procedure, by using `EXIT INPUT`.

If the CANCEL=FALSE option is set, no close action will be created, and you must write an `ON ACTION close` control block to create an explicit action.

Default actions IN CONSTRUCT

When an CONSTRUCT instruction executes, the runtime system creates a set of default actions.

According the invoked default action, field validation occurs and different CONSTRUCT control blocks are executed.

This table lists the default actions created for this dialog:

Table 278: Default actions created for the CONSTRUCT dialog

Default action	Description
`accept`	Validates the CONSTRUCT dialog (validates field criteria) *Creation can be avoided with* ACCEPT *attribute.*
`cancel`	Cancels the CONSTRUCT dialog (no validation, INT_FLAG is set) *Creation can be avoided with* CANCEL *attribute.*
`close`	By default, cancels the CONSTRUCT dialog (no validation, INT_FLAG is set) Default action view is hidden. See Implementing the close action on page 1337.
`help`	Shows the help topic defined by the HELP clause. *Only created when a* HELP *clause is defined.*

The accept and cancel default actions can be avoided with the ACCEPT and CANCEL dialog control attributes:

```
CONSTRUCT BY NAME cond ON field1 ATTRIBUTES (CANCEL=FALSE)
    ...
```

CONSTRUCT control blocks

CONSTRUCT control blocks execution order

This table shows the order in which the runtime system executes the control blocks in the CONSTRUCT instruction, according to the user action:

Table 279: Control block execution order for CONSTRUCT

Context / User action	Control Block execution order
Entering the dialog	**1.** `BEFORE CONSTRUCT` **2.** `BEFORE FIELD` (first field)
Moving from field A to field B	**1.** `AFTER FIELD` (for field A) **2.** `BEFORE FIELD` (for field B)
Validating the dialog	**1.** `AFTER FIELD` **2.** `AFTER CONSTRUCT`
Canceling the dialog	**1.** `AFTER CONSTRUCT`

BEFORE CONSTRUCT block

BEFORE CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular `CONSTRUCT` instruction, or when used as parallel dialog, the `BEFORE CONSTRUCT` is only executed once when dialog is started.

The `BEFORE CONSTRUCT` block is executed once at dialog startup, before the runtime system gives control to the user for criteria input. This block can be used to display messages to the user, initialize form fields with default search criteria values, and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
CONSTRUCT BY NAME where_part ON ...
   BEFORE CONSTRUCT
       MESSAGE "Enter customer search filter"
       CALL DIALOG.setActionActive("clean", FALSE )
   ...
```

The fields are cleared before the `BEFORE CONSTRUCT` block is executed.

You can use the `NEXT FIELD` control instruction in the `BEFORE CONSTRUCT` block, to jump to a specific field when the dialog starts.

BEFORE CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a `CONSTRUCT` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE CONSTRUCT` block is executed when the focus goes to a group of fields driven by a `CONSTRUCT` sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

`BEFORE CONSTRUCT` is executed after the `BEFORE DIALOG` block and before the `BEFORE FIELD` blocks.

In this example, the `BEFORE CONSTRUCT` block is used to display a message:

```
CONSTRUCT BY NAME sql ON customer.*
  BEFORE CONSTRUCT
    MESSAGE "Enter customer search filter"
```

AFTER CONSTRUCT block

AFTER CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular `CONSTRUCT` instruction, or when used as parallel dialog, the `AFTER CONSTRUCT` is only executed once when dialog is ended.

Use an AFTER CONSTRUCT block to execute instructions after the user has finished search criteria input.

AFTER CONSTRUCT is not executed if an EXIT CONSTRUCT is performed.

The code in AFTER CONSTRUCT can for example check if a criteria combination of different fields is required or denied, and force the end use to enter all

Before checking the content of the fields used in the CONSTRUCT, make sure that the INT_FLAG variable is FALSE: in case if the user cancels the dialog, the validation rules must be skipped.

Since no program variables are associated to the form fields, you must query the input buffers of the fields to get the values entered by the user.

```
CONSTRUCT BY NAME where_part ON ...
  ...
  AFTER CONSTRUCT
    IF NOT INT_FLAG THEN
       IF length(DIALOG.getFieldBuffer(cust_name))==0
       OR length(DIALOG.getFieldBuffer(cust_addr))==0 THEN
          ERROR "Enter a search criteria for customer name and address
 fields."
          NEXT FIELD CURRENT
       END IF
    END IF
END CONSTRUCT
```

To limit the validation to fields that have been modified by the end user, you can call the FIELD_TOUCHED() function or the DIALOG.getFieldTouched() method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the AFTER CONSTRUCT block is executed when the focus is lost by a group of fields driven by a CONSTRUCT sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER CONSTRUCT is executed after the AFTER FIELD and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER CONSTRUCT control block will keep the focus in the group of fields.

In this example, the AFTER CONSTRUCT block is used to build the SELECT statement:

```
CONSTRUCT BY NAME sql ON customer.*
   AFTER CONSTRUCT
       LET sql = "SELECT * FROM customers WHERE " || sql
```

BEFORE FIELD block

For fields controlled by an INPUT, INPUT ARRAY or by a CONSTRUCT instructions, the BEFORE FIELD block is executed every time the cursor enters into the specified field.

For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
   BEFORE FIELD cust_status
     LET p_cust.cust_comment = NULL
     MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing the first time with a specific form, the BEFORE FIELD block might be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column would have been done in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

AFTER FIELD block

In dialog parts driven by a simple INPUT, INPUT ARRAY or by a CONSTRUCT sub-dialog, the AFTER FIELD block is executed every time the focus leaves the specified field. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, based on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT or ACCEPT CONTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
   AFTER FIELD item_quantity
     IF p_item.item_quantity <= 0 THEN
        ERROR "Item quantity cannot be negative or zero"
        LET p_item.item_quantity = 0
        NEXT FIELD item_quantity
     END IF
```

CONSTRUCT interaction blocks

ON ACTION block

The `ON ACTION` *`action-name`* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
  ON ACTION action-name
     instruction
     ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
   ON ACTION zoom
      CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as `INPUT`, `INPUT ARRAY` and `CONSTRUCT`, if an action is specific to a field, add the `INFIELD` clause to have the action automatically enabled when the corresponding field gets the focus:

```
   ON ACTION zoom INFIELD cust_city
      CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the `ON ACTION` handler needs to define its own attributes at the program level. This can be done by adding the `ATTRIBUTES()` clause of `ON ACTION`:

```
   ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
      CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The `ON IDLE` *`seconds`* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of `ON IDLE` must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

Important: The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable

will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An `ON KEY (key-name)` block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The `ON KEY` block is supported for backward compatibility with TUI mode applications.

An `ON KEY` block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, `ON KEY(F5,F6)` creates two actions with the names `f5` and `f6`. Each action object will get an `ACCELERATORNAME` assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON TIMER block

The `ON TIMER` *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

CONSTRUCT control instructions

ACCEPT CONSTRUCT instruction

The `ACCEPT CONSTRUCT` instruction validates the `CONSTRUCT` instruction and exits the dialog block if no error is raised.

The `AFTER FIELD` and `AFTER CONSTRUCT` control blocks will be executed.

The statements after the `ACCEPT CONSTRUCT` will not be executed.

```
CONSTRUCT BY NAME where_part ON ...
```

```
  ...
  ON ACTION default_query
    CALL set_default_filter()
    ACCEPT CONSTRUCT
  ...
END CONSTRUCT
```

The CONSTRUCT instruction creates the default accept action to let the user validate the dialog. The ACCEPT CONSTRUCT instruction should only be used in specific cases when the default accept action is not appropriated.

CONTINUE CONSTRUCT instruction

CONTINUE CONSTRUCT skips all subsequent statements in the current control block and gives the control back to the dialog. This instruction is useful when program control is nested within multiple conditional statements, and you want to return the control to the dialog. If this instruction is called in a control block that is not AFTER CONSTRUCT, further control blocks might be executed according to the context. Actually, CONTINUE CONSTRUCT just instructs the dialog to continue as if the code in the control block was terminated (i.e. it's a kind of GOTO end_of_control_block). However, when executed in AFTER CONSTRUCT, the focus returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field. In this case the BEFORE FIELD of the current field will be invoked.

As alternative, use the NEXT FIELD control instruction to give the focus to a specific field and force the dialog to continue. However, unlike CONTINUE CONSTRUCT, the NEXT FIELD instruction will skip the further control blocks that are normally executed.

EXIT CONSTRUCT instruction

The EXIT CONSTRUCT instruction terminates the CONSTRUCT instruction and resumes the program execution at the instruction following the INPUT block.

Performing an EXIT CONSTRUCT instruction during a dialog is equivalent to cancel the dialog: No field validation will occur, and the AFTER FIELD or AFTER CONSTRUCT blocks will not be executed. The dialog is exited immediately. However, INT_FLAG will not be set to TRUE as when the cancel action is fired.

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The NEXT FIELD *field-name* instruction gives the focus to the specified field and forces the dialog to stay in that field.

This instruction can be used to control field input, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, it can also force a DISPLAY ARRAY or INPUT ARRAY to stay in the current row when NEXT FIELD is used in the AFTER ROW block.

If it exists, the BEFORE FIELD block of the corresponding field is executed.

The purpose of the NEXT FIELD instruction is give the focus to an editable field. Make sure that the field specified in NEXT FIELD is active, or use NEXT FIELD CURRENT. Non-editable fields are fields defined with the NOENTRY attribute, fields disabled at runtime with DIALOG.setFieldActive(), or fields using a widget that does not allow input, such as a LABEL.

Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("*field-name*") method to register a field, for example when the name is not known at compile time. However, this method only registers the field: It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.

Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog. Form fields are bound to program variables with the binding clause of

dialog instruction (`INPUT variable-list FROM field-list`, `INPUT BY NAME variable-list`, `CONSTRUCT BY NAME sql ON column-list`,`CONSTRUCT sql ON column-list FROM field-list`, `INPUT ARRAY array-name FROM screen-array.*`).

The field name specification can be any of the following:

- *field-name*
- *table-name.field-name*
- *screen-record-name.field-name*
- `FORMONLY.field-name`

Here are some examples:

- `"cust_name"`
- `"customer.cust_name"`
- `"cust_screen_record.cust_name"`
- `"item_screen_array.item_label"`
- `"formonly.total"`

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog according to the field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using `INPUT BY NAME variable-list`), the field prefix must be the database table name (or `FORMONLY`) used in the form file, or any valid screen-record using that field. When the `FROM` clause of the dialog specifies an explicit screen-record (for example, in `INPUT variable-list FROM screen-record.* / field-list-with-screen-record-prefix` or `INPUT ARRAY array-name FROM screen-array.*`), the field prefix must be the screen-record name used in the `FROM` clause.

Abstract field identification is supported with the `CURRENT`, `NEXT` and `PREVIOUS` keywords. These keywords represent the current, next and previous fields respectively. When using `FIELD ORDER FORM`, the `NEXT` and `PREVIOUS` options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the `FROM` or `BY NAME` clause).

In a procedural dialog, if the focus is in the first field of an `INPUT` or `CONSTRUCT` sub-dialog, `NEXT FIELD PREVIOUS` will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an `INPUT` or `CONSTRUCT` sub-dialog, `NEXT FIELD NEXT` will jump out of the current sub-dialog and set the focus to the next sub-dialog. `NEXT FIELD NEXT` or `NEXT FIELD PREVIOUS` also jumps to another sub-dialog when the focus is in a `DISPLAY ARRAY` sub-dialog. However, when using an `INPUT ARRAY` sub-dialog, `NEXT FIELD NEXT` from within the last column will loop to the first column of the current row, and `NEXT FIELD PREVIOUS` from within the first column will jump to the last column of the current row - the focus stays in the current `INPUT ARRAY` sub-dialog. When another sub-dialog gets the focus because of a `NEXT FIELD NEXT/PREVIOUS`, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable INPUT / INPUT ARRAY / CONSTRUCT field

Non-editable fields are fields defined with the `NOENTRY` attribute, fields disabled with `ui.Dialog.setFieldActive("field-name", FALSE)`, or fields using a widget that does not allow input, such as a `LABEL`.

If a `NEXT FIELD` instruction specifies a non-editable field, the `BEFORE FIELD` block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the `AFTER FIELD` block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the `FIELD ORDER` mode used by the dialog. Doing a `NEXT FIELD` to a non-editable field can lead to infinite loops in the dialog; Use `NEXT FIELD CURRENT` instead.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD *field-name* instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW and AFTER INPUT/DISPLAY/CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT/DISPLAY/CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user. This behavior allows you to implement data input rules:

```
AFTER ROW
  IF NOT int_flag AND arr_count()<=arr_curr() THEN
     IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
        ERROR "Amount of line exceeds max value."
        NEXT FIELD item_count
     END IF
  END IF
```

CLEAR instruction in dialogs

The CLEAR *field-list* and CLEAR SCREEN ARRAY *screen-array.** instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFEFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated to the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

Examples

Example 1: CONSTRUCT with binding by field position

Form definition in the *form1.per* file:

```
SCHEMA office
```

```
LAYOUT
GRID
{
  Customer id: [f001    ]
  First Name : [f002                  ]
  Last Name  : [f003                  ]
}
END
END

TABLES
  customer
END

ATTRIBUTES
  f001 = customer.id;
  f002 = customer.fname;
  f003 = customer.lname, UPSHIFT;
END

INSTRUCTIONS
  SCREEN RECORD sr_cust(customer.*);
END
```

Program:

```
MAIN
  DEFINE condition STRING
  DATABASE office
  OPEN FORM f1 FROM "form1"
  DISPLAY FORM f1
  CONSTRUCT condition
        ON id, fname, lname
        FROM sr_cust.*
  DISPLAY condition
END MAIN
```

Example 2: CONSTRUCT with binding by field name

Form definition file "form1.per" (same as in Example 1)

Program:

```
SCHEMA office
MAIN
  DEFINE condition STRING
  DEFINE statement STRING
  DEFINE cust RECORD LIKE customer.*

  DATABASE office

  OPEN FORM f1 FROM "form1"
  DISPLAY FORM f1

  CONSTRUCT BY NAME condition ON customer.*
    BEFORE CONSTRUCT
      DISPLAY "A*" TO fname
      DISPLAY "B*" TO lname
  END CONSTRUCT

  LET statement =
    "SELECT fname, lname FROM customer WHERE " || condition
```

```
  DISPLAY "SQL: " || statement

  DECLARE c1 CURSOR FROM statement
  FOREACH c1 INTO cust.*
    DISPLAY cust.*
  END FOREACH

END MAIN
```

Multiple dialogs (DIALOG)

The procedural `DIALOG` instruction allows to combine record list, record input, query criteria input in the same application form.

- Understanding multiple dialogs on page 1144
- Syntax of the procedural DIALOG instruction on page 1147
- Procedural dialog programming steps on page 1152
- Using multiple dialogs on page 1152
  - Identifying sub-dialogs in procedural DIALOG on page 1152
  - Structure of a procedural DIALOG block on page 1153
  - Procedural DIALOG block configuration on page 1158
  - Default actions created by a DIALOG block on page 1162
  - DIALOG data blocks on page 1163
  - DIALOG control blocks on page 1164
  - DIALOG interaction blocks on page 1179
  - DIALOG control instructions on page 1189
- Examples on page 1195
  - Example 1: DIALOG controlling two lists on page 1195
  - Example 2: DIALOG with CONSTRUCT and DISPLAY ARRAY on page 1196
  - Example 3: DIALOG with SUBDIALOG on page 1198

Understanding multiple dialogs

The concept of *multiple dialogs* refers to the usage of a procedural `DIALOG` block, to control several elements of a form. During the execution of a procedural dialog, no other window/form can be accessed: multiple dialogs are in the category of modal dialogs.

The `DIALOG` procedural instruction handles different parts of a form simultaneously, including simple display fields, simple input fields, read-only list of records, editable list of records, query by example fields, and action views. The `DIALOG` instruction acts as a collection of singular dialogs working in parallel.

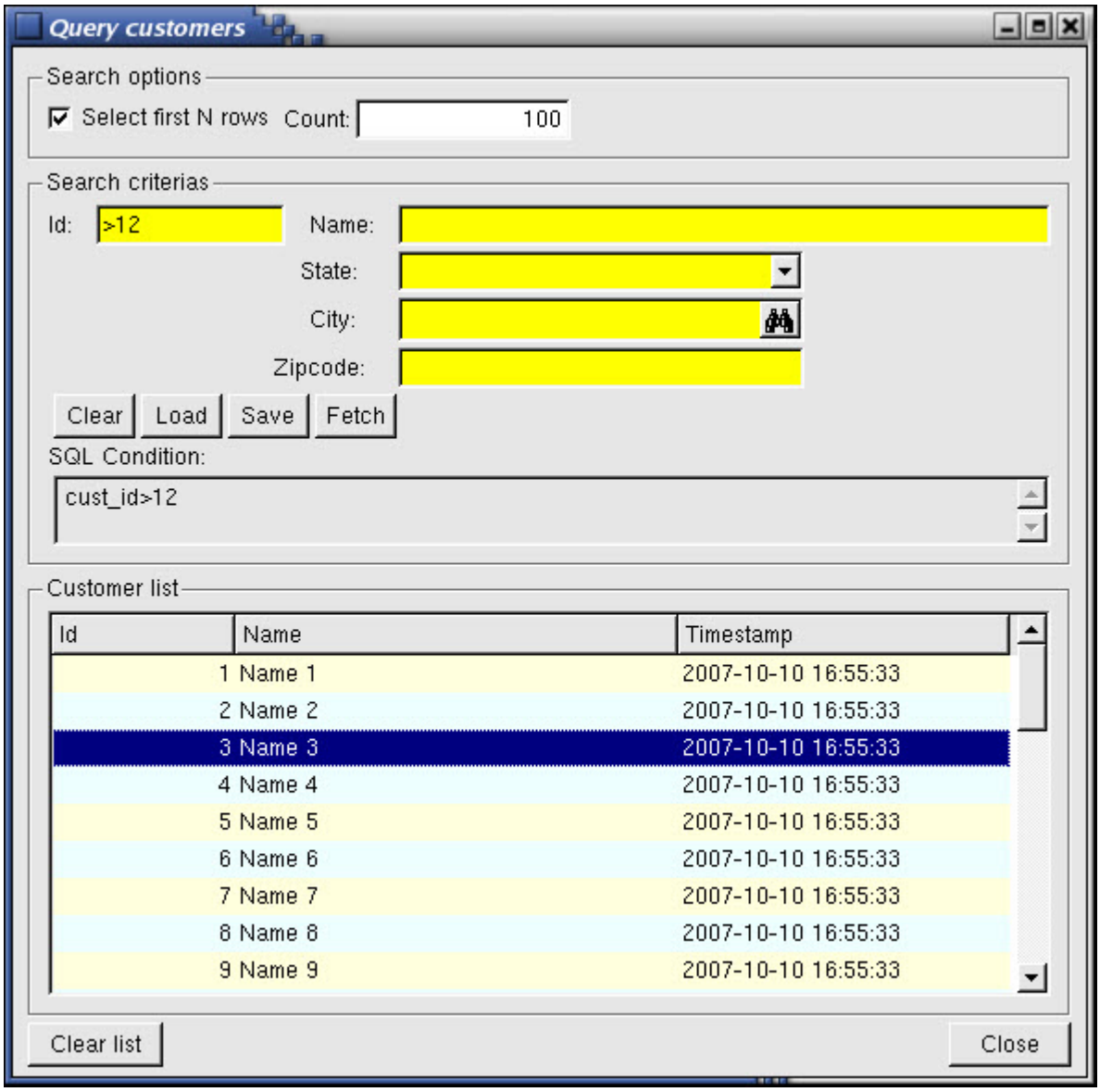


Figure 76: Query customers screenshot with multiple dialogs

"Singular interactive instructions" refer to `INPUT`, `CONSTRUCT`, `DISPLAY ARRAY` and `INPUT ARRAY` independent blocks not surrounded by the `DIALOG` / `END DIALOG` keywords. While the `DIALOG` instruction reuses some of the semantics and behaviors of singular interactive instructions, there are some differences.

Like the singular interactive instructions, `DIALOG` is an interactive instruction. You can execute a `DIALOG` instruction from one of the singular dialogs, or execute a singular dialog from a `DIALOG` block. The parent dialog will be disabled until the child dialog returns.

A `DIALOG` procedural instruction consist of several sub-dialog blocks declared inside the `DIALOG` instruction, or external dialog blocks declared in scope outside of the current function. The external dialogs are attached to the current dialog with the `SUBDIALOG` clause.

The `DIALOG` instruction binds program variables (such as simple records or arrays of records) with a screen-record or screen-array defined in a form, allowing the user to view and update application data.

When a `DIALOG` block executes, it activates the current form (the form most recently displayed or the form in the current window). When the statement completes execution, the form is deactivated.

This screen shot is from a demo program called "Query customers" that you can find in FGLDIR/demo/MultipleDialogs. This demo involves a DIALOG block that contains a simple INPUT block, a CONSTRUCT block and a DISPLAY ARRAY block:

The syntax of the DIALOG instruction is very close to singular dialogs, using common triggers such as BEFORE FIELD, ON ACTION, and so on. Despite the similarities, the behavior and semantics of DIALOG are a bit different from singular dialogs.

Understand that the DIALOG instruction is not provided to replace singular dialogs. Singular dialogs are still supported. It is recommended that you use singular dialogs if no multiple dialog is required.

Unlike singular dialogs, the DIALOG instruction does not use the INT_FLAG variable. You must implement ON ACTION accept or ON ACTION cancel to handle dialog validation or cancellation. These actions do not exist by default in DIALOG.

Unlike singular dialogs creating implicit accept and cancel actions, by default there is no way to quit the DIALOG instruction. You must implement your own action handler and execute EXIT DIALOG or ACCEPT DIALOG.

A good practice is to write a setup dialog function to centralize all field and action activations according to the context. Call that setup function at any place in the DIALOG code where the field and action activation rules may change.

While static arrays are supported by the DIALOG instruction, it is strongly recommended that you use dynamic arrays instead. With a dynamic array, the actual number of rows is automatically defined by the array variable, while static arrays need an additional step to define the total number of rows.

When needed, use the UNBUFFERED mode with multiple dialogs to force model/view synchronization, and use the FIELD ORDER FORM option to follow the TABINDEX definitions in the form file.

This example is of a DIALOG procedural instruction that includes both an INPUT and a DISPLAY ARRAY sub-dialog, plus a sub-dialog defined externally and included with the SUBDIALOG keyword:

```
SCHEMA stores
DEFINE p_customer RECORD LIKE customer.*
DEFINE p_orders DYNAMIC ARRAY OF RECORD LIKE order.*
FUNCTION customer_dialog()
  DIALOG ATTRIBUTES(UNBUFFERED, FIELD ORDER FORM)
    INPUT BY NAME p_customer.*
      AFTER FIELD cust_name
        CALL setup_dialog(DIALOG)
    END INPUT
    DISPLAY ARRAY p_orders TO s_orders.*
      BEFORE ROW
        CALL setup_dialog(DIALOG)
    END DISPLAY
    SUBDIALOG common_footer
    ON ACTION close
      EXIT DIALOG
  END DIALOG
END FUNCTION
```

All elements of the dialog are active at the same time, so you must handle tabbing order properly. By default - as in singular dialogs - the tabbing order is driven by the binding list (order of program variables). It is strongly recommended that you use the FIELD ORDER FORM option and the TABINDEX field attributes instead.

Like the singular INPUT ARRAY instruction, DIALOG creates implicit insert, append and delete actions. These actions are only active when the focus is in the list.

Syntax of the procedural DIALOG instruction

The DIALOG block is an interactive instruction that executes several sub-dialogs simultaneously.

Syntax

```
DIALOG
   [ ATTRIBUTES ( dialog-control-attribute [,...] ) ]

   { record-input-block
   | construct-block
   | display-array-block
   | input-array-block
   | SUBDIALOG dialog-name
   }
   [...]

   [
   dialog-control-block
   [...]
   ]

END DIALOG
```

where *dialog-control-attribute* is:

```
{ FIELD ORDER FORM
| UNBUFFERED [ = boolean ]
}
```

where *dialog-name* in the SUBDIALOG clause is the name of a declarative dialog block defined outside the scope of the current function.

where *dialog-control-block* is one of:

```
{ BEFORE DIALOG
| ON ACTION action-name
             [ ATTRIBUTES ( action-attributes-dialog ) ]
| ON KEY ( key-name [,...] )
| ON IDLE seconds
| ON TIMER seconds
| COMMAND option-name
        [ option-comment ]
        [ HELP help-number ]
| COMMAND KEY ( key-name  [,...] ) option-name
        [ option-comment ]
        [ HELP help-number ]
| AFTER DIALOG
}
    dialog-statement
    [...]
```

where *action-attributes-dialog* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *record-input-block* is:

```
INPUT { BY NAME { variable | record.* } [,...]
      | variable | record.* } [,...] FROM field-list
      }
   [ ATTRIBUTES ( input-control-attribute [,...] ) ]
   [ input-control-block
       [...]
   ]
END INPUT
```

where *input-control-attribute* is:

```
{ HELP = help-number
| NAME = "sub-dialog-name"
| WITHOUT DEFAULTS [ = boolean ]
}
```

where *input-control-block* is one of:

```
{ BEFORE INPUT
| BEFORE FIELD field-spec [,...]
| ON CHANGE field-spec [,...]
| AFTER FIELD field-spec [,...]
| AFTER INPUT
| ON ACTION action-name
             [ INFIELD field-spec ]
             [ ATTRIBUTES ( action-attributes-input ) ]
| ON KEY ( key-name [,...] )}
    dialog-statement
    [...]
```

where *action-attributes-input* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| VALIDATE = NO
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *construct-block* is:

```
CONSTRUCT { BY NAME variable ON column-list
          | variable ON column-list FROM field-list
          }
   [ ATTRIBUTES ( construct-control-attribute [,...] ) ]
   [ construct-control-block
       [...]
   ]
END CONSTRUCT
```

where *construct-control-attribute* is:

```
{ HELP = help-number
| NAME = "sub-dialog-name"
}
```

where *construct-control-block* is one of:

```
{ BEFORE CONSTRUCT
| BEFORE FIELD field-spec [,...]
| AFTER FIELD field-spec [,...]
| AFTER CONSTRUCT
| ON ACTION action-name
             [INFIELD field-spec]
             [ ATTRIBUTES ( action-attributes-construct ) ]
| ON KEY ( key-name [,...] )}
    dialog-statement
    [...]
```

where *action-attributes-construct* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *display-array-block* is:

```
DISPLAY ARRAY array TO screen-array.*
   [ ATTRIBUTES ( display-array-control-attribute [,...] ) ]
   [ display-array-control-block
       [...]
   ]
END DISPLAY
```

where *display-array-control-attribute* is:

```
{ HELP = help-number
| COUNT = row-count
| KEEP CURRENT ROW = [ = boolean ]
| DETAILACTION = action-name
| DOUBLECLICK = action-name
| ACCESSORYTYPE = { DETAIBUTTON | DISCLOSUREINDICATOR | CHECKMARK }
}
```

where *display-array-control-block* is one of:

```
{ BEFORE DISPLAY
| BEFORE ROW
| AFTER ROW
| AFTER DISPLAY
| ON ACTION action-name
             [ ATTRIBUTES ( action-attributes-display-array ) ]
| ON KEY ( key-name [,...] )
| ON FILL BUFFER
| ON SELECTION CHANGE
| ON SORT
| ON APPEND [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON INSERT [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON UPDATE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON DELETE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON EXPAND ( row-index )
| ON COLLAPSE ( row-index )
| ON DRAG_START ( dnd-object )
| ON DRAG_FINISH ( dnd-object )
```

```
| ON DRAG_ENTER( dnd-object )
| ON DRAG_OVER ( dnd-object )
| ON DROP ( dnd-object ) }
    dialog-statement
    [...]
```

where *action-attributes-display-array* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
| ROWBOUND
    [,...] }
```

where *action-attributes-listmod-triggers* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *input-array-block* is:

```
INPUT ARRAY array FROM screen-array.*
   [ ATTRIBUTES ( input-array-control-attribute [,...] ) ]
   [ input-array-control-block
       [...]
   ]
END INPUT
```

where *input-array-control-attribute* is:

```
{ APPEND ROW [ = boolean ]
| AUTO APPEND [ = boolean ]
| COUNT = row-count
| DELETE ROW [ = boolean ]
| HELP = help-number
| INSERT ROW [ = boolean ]
| KEEP CURRENT ROW [ = boolean ]
| MAXCOUNT = max-row-count
| WITHOUT DEFAULTS [ = boolean ]
}
```

where *input-array-control-block* is one of:

```
{ BEFORE INPUT
| BEFORE ROW
| BEFORE FIELD [,...]
| ON CHANGE field-spec [,...]
| AFTER FIELD field-spec [,...]
| ON ROW CHANGE
| ON SORT
| AFTER ROW
| BEFORE DELETE
| AFTER DELETE
| BEFORE INSERT
```

```
| AFTER INSERT
| AFTER INPUT
| ON ACTION action-name
             [INFIELD field-spec]
             [ ATTRIBUTES ( action-attributes-input-array ) ]
| ON KEY ( key-name [,...] ) }
    dialog-statement
    [...]
```

where *action-attributes-input-array* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| VALIDATE = NO
| CONTEXTMENU = { YES | NO | AUTO }
| ROWBOUND
    [,...] }
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT DIALOG
| CONTINUE DIALOG
| EXIT DIALOG
| NEXT FIELD
   { CURRENT
   | NEXT
   | PREVIOUS
   | field-spec
   }
}
```

where *field-list* defines a list of fields with one or more of:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *field-spec* identifies a unique field with one of:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
}
```

where *column-list* defines a list of database columns as:

```
{ column-name
| table-name.*
| table-name.column-name
} [,...]
```

1. *variable-definition* is a variable declaration with data type as in a regular `DEFINE` statement.
2. *array* is the array of records used by the `DIALOG` statement.
3. *help-number* is an integer that allows you to associate a help message number with the command.
4. *field-name* is the identifier of a field of the current form.
5. *option-name* is a string expression defining the label of the action and identifying the action that can be executed by the user.
6. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
7. *column-name* is the identifier of a database column of the current form.
8. *table-name* is the identifier of a database table of the current form.
9. *variable* is a simple program variable (not a record).
10. *record* is a program record (structured variable).
11. *screen-array* is the screen array that will be used in the current form.
12. *line* is a screen array line in the form.
13. *screen-record* is the identifier of a screen record of the current form.
14. *action-name* identifies an action that can be executed by the user.
15. *seconds* is an integer literal or variable that defines a number of seconds.
16. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
17. *row-index* identifies the program variable which holds the row index corresponding to the tree node that has been expanded or collapsed.
18. *dnd-object* references a `ui.DragDrop` variable defined in the scope of the dialog.
19. *statement* is any instruction supported by the language.
20. *action-attributes* are dialog-specific action attributes for the action.

Procedural dialog programming steps

The following steps describe how to implement a procedural `DIALOG` block:

1. Create a form specification file containing screen record(s) and/or screen array(s). The screen records and screen arrays identify the presentation elements to be used by the runtime system to display the data models (i.e. the content of program variables bound to the `DIALOG` blocks).
2. With the `DEFINE` instruction, declare program variables (i.e. records and arrays) that will be used as data models. For record lists (`DISPLAY ARRAY` or `INPUT ARRAY`), the members of the program array must correspond to the elements of the screen array, by number and data types. To handle record lists, use dynamic arrays instead of static arrays.
3. Open and display the form, using the `OPEN WINDOW WITH FORM` clause or the `OPEN FORM` / `DISPLAY FORM` instructions.
4. Fill the program variables (.i.e the model) with data. For lists, you typically use a result set cursor.
5. Implement the `DIALOG` instruction block to handle interaction. Define each sub-dialog with program variables to be used as data models. The sub-dialogs will define how variables will be used (display or input).
   a) Inside each sub-dialog instruction, define the behavior with control blocks such as `BEFORE DIALOG`, `AFTER ROW`, `BEFORE FIELD`, and interaction blocks such as `ON ACTION`.
   b) To end the `DIALOG` instruction, implement an `ON ACTION close` or `ON ACTION accept` / `ON ACTION cancel` to handle dialog validation and cancellation, with the `ACCEPT DIALOG` and `EXIT DIALOG` control instructions. The `INT_FLAG` variable will not be set as in singular dialogs.

Using multiple dialogs

To use multiple dialogs, you must understand how they work and how to structure the code.

Identifying sub-dialogs in procedural DIALOG

Sub-dialogs need to be identified by a name to distinguish the different contexts.

A procedural `DIALOG` block is a collection of sub-dialogs that act as controllers for different parts of a form. In order to program a procedural `DIALOG` block, there must be a unique identifier for each sub-dialog.

For example, to set the current row of a screen array with the `DIALOG.setCurrentRow()` method, you pass the name of the screen array to specify the sub-dialog to be affected. Sub-dialog identifiers are also used as a prefix to specify actions for the sub-dialog.

The following topics describe how to specify the names of the different types of `DIALOG` sub-dialogs:

- Identifying an INPUT sub-dialog on page 1154
- Identifying a DISPLAY ARRAY sub-dialog on page 1156
- Identifying an INPUT ARRAY sub-dialog on page 1157
- Identifying a CONSTRUCT sub-dialog on page 1155
- The SUBDIALOG clause on page 1158.

Structure of a procedural DIALOG block

A procedural `DIALOG` instruction is made of several *sub-dialogs*, plus global control blocks such as `BEFORE DIALOG` and action handlers such as `ON ACTION` or `COMMAND`.

Sub-dialogs can be defined inside the `DIALOG` instruction, or can be declared externally in another module and attached to the current `DIALOG` block with the `SUBDIALOG` clause. A dialog defined in the scope of a function is know as a *procedural dialog block*, while a dialog declared in the scope of a module is named a *declarative dialog block*.

The sub-dialogs bind program variables to form fields and define the type of interaction that will take place for the data model (simple input, list input or query). The sub-dialogs implement individual control blocks which let you control the behavior of the interactive instruction. Sub-dialogs can also hold action handlers, which will define local sub-dialog actions.

The `DIALOG` procedural instruction can hold the following type of sub-dialogs:

1. Simple record input with the `INPUT` sub-dialog block.
2. Query by example input with the `CONSTRUCT` sub-dialog block.
3. Read-only record list navigation with the `DISPLAY ARRAY` sub-dialog block.
4. Editable record list handling with the `INPUT ARRAY` sub-dialog block.
5. A `SUBDIALOG` clause referencing a declarative sub-dialog by name.

The INPUT sub-dialog
The `INPUT` sub-dialog implement single record input in fields of the current form.

Program variable to form field binding

Each record member variable is bound to the corresponding field of a screen record, in order to manipulate the values that the user enters in the form fields.

The `INPUT` clause can be used in two forms:

1. `INPUT BY NAME` *`variable-list`*
2. `INPUT` *`variable-list`* `FROM` *`field-list`*

The `BY NAME` clause implicitly binds the fields to the variables that have the same identifiers as the field names. The variables must be declared with the same names as the fields from which they accept input. The runtime system ignores any record name prefix when making the match. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exceptions, and sets the `STATUS` variable to a negative value.

```
DEFINE p_cust RECORD
       cust_num INTEGER,
       cust_name VARCHAR(50),
       cust_address VARCHAR(100)
   END RECORD
   ...
DIALOG
   INPUT BY NAME p_cust.*
```

```
      BEFORE FIELD cust_name
        ...
   END INPUT
   ...
END DIALOG
```

The FROM clause explicitly binds the fields in the screen record to a list of program variables by position. The number of variables or record members must equal the number of fields listed in the FROM clause. Each variable must be of the same (or a compatible) data types as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

```
DEFINE c_name VARCHAR(50)
       c_addr VARCHAR(100)
   ...
DIALOG
  INPUT c_name,
        c_addr
   FROM FORMONLY.field01,
        FORMONLY.field02
    BEFORE FIELD cust_name
        ...
  END INPUT
   ...
END DIALOG
```

Identifying an INPUT sub-dialog

The name of an INPUT sub-dialog can be used to qualify sub-dialog actions with a prefix.

In order to identify the INPUT sub-dialog with a specific name, you can use the ATTRIBUTES clause to set the NAME attribute:

```
INPUT BY NAME p_cust.*
      ATTRIBUTES (NAME = "cust")
  ...
```

Control blocks in INPUT

Simple record input declared with the INPUT sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE FIELD
- ON CHANGE
- AFTER FIELD
- AFTER INPUT

In the singular INPUT instruction, BEFORE INPUT and AFTER INPUT blocks are typically used as initialization and finalization blocks. In an INPUT sub-dialog of a DIALOG block, BEFORE INPUT and AFTER INPUT blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

The CONSTRUCT sub-dialog

The CONSTRUCT sub-dialog provides database query by example feature, converting search criteria entered by the user into an SQL WHERE condition that can be use to execute a SELECT statement.

Defining query by example fields

The CONSTRUCT sub-dialog requires a character string variable to hold the WHERE clause, and a list of screen fields where the user can enter search criteria.

```
DEFINE sql_condition STRING
  ...
DIALOG
  CONSTRUCT BY NAME sql_condition
      ON customer.cust_name, customer.cust_address
    BEFORE FIELD cust_name
    ...
  END CONSTRUCT
  ...
END DIALOG
```

Make sure the character string variable is large enough to store all possible SQL conditions. It is better to use a STRING data type to avoid any size problems.

CONSTRUCT uses the field data types defined in the current form file to produce the SQL conditions. This is different from other interactive instructions, where the data types of the program variables define the way to handle input/display. It is *strongly* recommended (but not mandatory) that the form field data types correspond to the data types of the program variables used for input. This is implicit if both form fields and program variables are based on the database schema file.

The CONSTRUCT clause can be used in two forms:

1. `CONSTRUCT BY NAME` *`string-variable`* `ON` *`column-list`*
2. `CONSTRUCT` *`string-variable`* `ON` *`column-list`* `FROM` *`field-list`*

The BY NAME clause implicitly binds the form fields to the columns, where the form field identifiers match the column names specified in the column-list after the ON keyword. You can specify the individual column names (separated by commas) or use the `tablename.*` shortcut to include all columns defined for a table in the database schema file.

The FROM clause explicitly binds the form fields listed after the FROM keyword with the column definitions listed after the ON keyword.

In both cases, the name of the columns in *column-list* will be used to produce the SQL condition in *string-variable*.

Identifying a CONSTRUCT sub-dialog

The name of a CONSTRUCT sub-dialog can be used to qualify sub-dialog actions with a prefix. In order to identify the CONSTRUCT sub-dialog with a specific name, use the ATTRIBUTES clause to set the NAME attribute:

```
CONSTRUCT BY NAME sql_condition ON customer.*
  ATTRIBUTES (NAME = "q_cust")
  ...
```

Control blocks in CONSTRUCT

A Query By Example declared with the CONSTRUCT clause can raise the following triggers:

- BEFORE CONSTRUCT
- BEFORE FIELD

- AFTER FIELD
- AFTER CONSTRUCT

In the singular CONSTRUCT instruction, BEFORE CONSTRUCT and AFTER CONSTRUCT blocks are typically used as initialization and finalization blocks. In DIALOG block, BEFORE CONSTRUCT and AFTER CONSTRUCT blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the group of fields defined by this sub-dialog.

The DISPLAY ARRAY sub-dialog
The DISPLAY ARRAY sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

Program array to screen array binding

The DISPLAY ARRAY sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the TO keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the DIALOG instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```
DEFINE p_items DYNAMIC ARRAY OF RECORD
       item_num INTEGER,
       item_name VARCHAR(50),
       item_price DECIMAL(6,2)
   END RECORD
   ...
DIALOG
   DISPLAY ARRAY p_items TO sa.*
      BEFORE ROW
      ...
   END DISPLAY
   ...
END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
   ...
DIALOG
   DISPLAY ARRAY p_names TO sa.*
      BEFORE DELETE
      ...
   END DISPLAY
   ...
END DIALOG
```

Identifying a DISPLAY ARRAY sub-dialog

The name of the screen array specified with the TO clause identifies the list. The dialog class method such as takes the name of the screen array as the parameter, identifying the list. For example, you would use `DIALOG.getCurrentRow("screen-array")` to query for the current row in the list identified by 'screen-array'. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in DISPLAY ARRAY

Read-only record lists declared with the DISPLAY ARRAY sub-dialog can raise the following triggers:

- BEFORE DISPLAY
- BEFORE ROW
- AFTER ROW
- AFTER DISPLAY

In the singular `DISPLAY ARRAY`instruction, `BEFORE DISPLAY` and `AFTER DISPLAY` blocks are typically used as initialization and finalization blocks. In a `DISPLAY ARRAY` sub-dialog of a `DIALOG` block, `BEFORE DISPLAY` and `AFTER DISPLAY` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The INPUT ARRAY sub-dialog
The `INPUT ARRAY` sub-dialog is the controller to implement the navigation and edition in a list of records.

> **Important:** This feature is not supported on mobile platforms.

Program array to screen array binding

The `INPUT ARRAY` sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the `FROM` keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the `DIALOG` instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```
DEFINE p_items DYNAMIC ARRAY OF RECORD
       item_num INTEGER,
       item_name VARCHAR(50),
       item_price DECIMAL(6,2)
   END RECORD
   ...
DIALOG
   INPUT ARRAY p_items FROM sa.*
      BEFORE INSERT
      ...
   END INPUT
   ...
END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
   ...
DIALOG
   INPUT ARRAY p_names FROM sa.*
      BEFORE DELETE
      ...
   END INPUT
   ...
END DIALOG
```

Identifying an INPUT ARRAY sub-dialog

The name of the screen array specified with the `FROM` clause will be used to identify the list. For example, the dialog class method such as `DIALOG.getCurrentRow("screen-array")` takes the name of the screen array as the parameter, to identify the list you want to query for the current row. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in INPUT ARRAY

Editable record lists declared with the `INPUT ARRAY` sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE ROW
- BEFORE FIELD
- ON CHANGE
- AFTER FIELD
- ON ROW CHANGE
- AFTER ROW
- BEFORE DELETE
- AFTER DELETE
- BEFORE INSERT
- AFTER INSERT
- AFTER INPUT

In the singular `INPUT ARRAY` instruction, `BEFORE INPUT` and `AFTER INPUT` blocks are typically used as initialization and finalization blocks. In the `INPUT ARRAY` sub-dialog of a `DIALOG` block, `BEFORE INPUT` and `AFTER INPUT` blocks will be executed each time the focus goes to (`BEFORE`)or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The SUBDIALOG clause

The `SUBDIALOG` clause attaches a declarative dialog to the current procedural `DIALOG` block. The declarative dialog will be implemented outside of the scope of the current dialog, at the same level as a function. The declarative dialog can be defined in a different module.

In terms of semantics, behavior and control block execution, a declarative dialog attached to a procedural dialog behaves like a sub-dialog that is defined inside the procedural `DIALOG` block. For example, the `BEFORE INPUT` control block will be executed for a declarative dialog when the focus goes to one of the fields of that sub-dialog.

Other sub-dialogs can reference the attached declarative dialog in the current scope, for example to execute a `NEXT FIELD` instruction referencing a field in another sub-dialog.

When using the `DIALOG` keyword inside a declarative dialog block to use `ui.Dialog` class methods, it references the current procedural dialog object.

Like other module elements such as functions and reports, the name specification is mandatory when defining a declarative dialog. The name of the declarative dialog will be referenced in a `SUBDIALOG` clause of a procedural dialog instruction.

Implementing a sub-dialog as a declarative dialog in a separate module can be used in conjunction with the form inclusion directive in the `LAYOUT` section of form specification files. With form inclusion and declarative dialogs, you enforce code reusability in your application sources.

Note that declarative dialog blocks can also be used to implement parallel dialogs.

Procedural DIALOG block configuration

This sections describes the `ATTRIBUTES` clause attributes that can be used to configure a procedural `DIALOG` instruction and its sub-dialogs.

The `ATTRIBUTES` clause of dialogs overrides all default attributes and temporarily override any display attributes that the `OPTIONS` or the `OPEN WINDOW` statement specified for these fields.

DIALOG ATTRIBUTES clause

FIELD ORDER FORM option

By default, the form tabbing order is defined by the variable list in the binding specification. You can control the tabbing order by using the `FIELD ORDER FORM` attribute; when this attribute is used, the tabbing order is defined by the `TABINDEX` attribute of the form items.

The field order mode can also be specified globally with the `OPTIONS FIELD ORDER` instruction.

With `FIELD ORDER FORM`, if the user changes the focus from field A to a distant field B with the mouse, the dialog does not execute the `BEFORE FIELD` / `AFTER FIELD` triggers of intermediate fields which appear in the binding specification between field A and field B. Unlike singular dialogs, if the default `FIELD ORDER CONSTRAINT` mode is used in a multiple dialog instruction, intermediate triggers are never executed (i.e. the `Dialog.fieldOrder` FGLPROFILE entry is ignored by `DIALOG`.)

See also Defining the tabbing order on page 1271.

UNBUFFERED option

The `UNBUFFERED` attribute indicates that the dialog must be sensitive to program variable changes. When using this option, you bypass the compatible "buffered" mode.

The unbuffered mode can be set globally for all `DIALOG` instructions with the `ui.Dialog.setDefaultUnbuffered()` class method:

```
CALL ui.Dialog.setDefaultUnbuffered(TRUE)
DIALOG -- Will work in UNBUFFERED mode     ...
END DIALOG
```

INPUT ATTRIBUTES clause
Attributes of the INPUT clause of a DIALOG block.

NAME option

The `NAME` attribute can be used to identify the `INPUT` sub-dialog, especially useful to qualify sub-dialog actions.

HELP option

The `HELP` attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the `INPUT` sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The `HELP` clause overrides the `HELP` attribute.

WITHOUT DEFAULTS option

By default, sub-dialogs use the default values defined in the form files. If you want to use the values stored in the program variables bound to the dialog, you must use the `WITHOUT DEFAULTS` attribute. For more details see WITHOUT DEFAULTS option.

DISPLAY ARRAY ATTRIBUTES clause
Attributes of the DISPLAY ARRAY clause of a DIALOG block.

HELP option

The `HELP` attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the `DISPLAY ARRAY` sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The `HELP` clause overrides the `HELP` attribute.

COUNT option

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array, unless page mode is used. In this case, the COUNT attribute must be used to define the total number of rows, because the dynamic array will only hold a page of the entire row set. If the value of COUNT is negative or zero, it defines an empty list.

See also Controlling the total number of rows on page 1350.

DOUBLECLICK option

The DOUBLICKCLICK option can be used to define the action that will be fired when the user chooses a row from the list. On front-end platforms using a mouse-device, this corresponds to a physical double-click on a row with the mouse. On mobile front-ends, this corresponds to a tap on the row with a finger. Note that this attribute can also be defined for the TABLE/TREE containers in form files; DOUBLECLICK in DISPLAY ARRAY attributes has a higher precedence as DOUBLECLICK in the form file. For more details, see Defining the action for a row choice on page 1360.

ACCESSORTYPE option

Important: This feature is only for mobile platforms.

The ACCESSORYTYPE attribute can be used to define the decoration of rows, typically used on a iOS device. Values can be DETAILBUTTON, DISCLOSUREINDICATOR, CHECKMARK to respectively get a (i), > or checkmark icon. For more details, see Row configuration on iOS devices on page 1369.

DETAILACTION option

Important: This feature is only for mobile platforms.

The DETAILACTION attribute can be used to define the action that will be fired when the user selects the detail button of a row. The detail button is typically shown with a (i) icon on iOS devices. Note that the DOUBLECLICK attribute can be used to distinguish the action when the user selects the row instead of the detail button in the row. For more details, see Row configuration on iOS devices on page 1369.

INPUT ARRAY ATTRIBUTES clause
Attributes of the INPUT ARRAY clause of a DIALOG block.

INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header:

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help the INPUT ARRAY dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

WITHOUT DEFAULTS option

You typically use the INPUT ARRAY sub-dialog with the WITHOUT DEFAULTS attribute. If this attribute is not set when using an INPUT ARRAY sub-dialog, the list is empty even if the array holds data. For more details see WITHOUT DEFAULTS option.

COUNT option

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array. If you specify

the COUNT attribute, the WITHOUT DEFAULTS option is not required because it is implicit. If the COUNT attribute is greater than MAXCOUNT, the runtime system will take MAXCOUNT as the actual number of rows. If the value of COUNT is negative or zero, it defines an empty list.

MAXCOUNT option

The MAXCOUNT attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit of the total number of rows the user can enter. It can be used with static or dynamic arrays.

When binding a static array, MAXCOUNT is used as upper limit if it is lower or equal to the actual declared static array size. If MAXCOUNT is greater than the array size, the size of the static array is used as the upper limit. If MAXCOUNT is lower than the COUNT attribute (or to the SET_COUNT() parameter when using a singular INPUT ARRAY), the actual number of rows in the array will be reduced to MAXCOUNT.

When binding a dynamic array, the user can enter an infinite number of rows unless the MAXCOUNT attribute is used. If MAXCOUNT is lower than the actual size of the dynamic array, the number of rows in the array will be reduced to MAXCOUNT.

If MAXCOUNT is negative or equal to zero, the user cannot insert rows.

APPEND ROW option

The APPEND ROW attribute can be set to FALSE to avoid the append default action, and deny the user to add rows at the end of the list. If APPEND ROW =FALSE, it is still possible to insert rows in the middle of the list. Use the INSERT ROW attribute to disallow the user from inserting rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

INSERT ROW option

The INSERT ROW attribute can be set to FALSE to avoid the insert default action, and deny the user to insert new rows in the middle of the list. However, even if INSERT ROW is FALSE, it is still possible to append rows at the end of the list. Use the APPEND ROW attribute to disallow the user from appending rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

DELETE ROW option

The DELETE ROW attribute can be set to FALSE to avoid the delete default action, and deny the user to remove rows from the list.

AUTO APPEND option

By default, an INPUT ARRAY controller creates a temporary row when needed (for example, when the user deletes the last row of the list, an new row will be automatically created). You can prevent this default behavior by setting the AUTO APPEND attribute to FALSE. When this attribute is set to FALSE, the only way to create a new temporary row is to execute the append action.

If both the APPEND ROW and INSERT ROW attributes are set to FALSE, the dialog automatically behaves as if AUTO APPEND equals FALSE.

KEEP CURRENT ROW option

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

CONSTRUCT ATTRIBUTES clause
Attributes of the CONSTRUCT clause of a DIALOG block.

HELP option

The `HELP` attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the `CONSTRUCT` sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The `HELP` clause overrides the `HELP` attribute.

NAME option

The `NAME` attribute can be used to identify the `CONSTRUCT` sub-dialog; this is especially useful to qualify sub-dialog actions.

Default actions created by a DIALOG block
Default actions ease the implementation of the controller by providing expected actions.

According to the sub-dialogs defined in a (declarative or procedural) `DIALOG` block, the runtime system creates a set of default actions. These actions are provided to ease the implementation of the controller. For example, when using an `INPUT ARRAY` sub-dialog, the dialog instruction will automatically create the insert, append and delete default actions.

Table 280: Default actions created for the DIALOG block on page 1162 lists the default actions created for the `DIALOG` interactive instruction, according to the sub-dialogs defined:

Table 280: Default actions created for the DIALOG block

Default action	Control Block execution order
`help`	Shows the help topic defined by the `HELP` clause. *Only created when a* `HELP` *clause or option is defined for the sub-dialog.*
`insert`	Inserts a new row before current row. *Only created if* `INPUT ARRAY` *is used; action creation can be avoided with* `INSERT ROW = FALSE` *attribute.*
`append`	Appends a new row at the end of the list. *Only created if* `INPUT ARRAY` *is used; action creation can be avoided with* `APPEND ROW = FALSE` *attribute.*
`delete`	Deletes the current row. *Only created if* `INPUT ARRAY` *is used;* action *creation can be avoided with* `DELETE ROW = FALSE` *attribute.*
`nextrow`	Moves to the next row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*

Default action	Control Block execution order
`prevrow`	Moves to the previous row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`firstrow`	Moves to the first row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`lastrow`	Moves to the last row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`find`	Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value. *Only created if the context allows built-in find.*
`findnext`	Seeks to the next row matching the value entered during the fglfind dialog. *Only created if the context allows built-in find.*

The `insert`, `append` and `delete` default actions can be avoided with dialog control attributes:

```
INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, APPEND
 ROW=FALSE, ... )
     ...
```

DIALOG data blocks

Dialog data blocks are dialog triggers invoked when the dialog controller needs data to feed the view with values.

Such blocks are typically used when record list data is provided dynamically, with the paged mode or when implementing dynamic tree-views.

- ON FILL BUFFER block on page 1082
- ON EXPAND block on page 1082
- ON COLLAPSE block on page 1082

ON FILL BUFFER block

The `ON FILL BUFFER` block is used to fill a page of rows into the dynamic array, according to an offset and a number of rows.

This data block is used in the `DISPLAY ARRAY` blocks.

The offset can be retrieved with the `FGL_DIALOG_GETBUFFERSTART()` built-in function and the number of rows to provide is defined by the `FGL_DIALOG_GETBUFFERLENGTH()` built-in function.

ON EXPAND block
The `ON EXPAND` block is executed when a tree view node is expanded (i.e. opened).

This data block is used to implement dynamic trees in a `DISPLAY ARRAY`, where nodes are added according to the nodes opened by the end user.

ON COLLAPSE block
The `ON COLLAPSE` block is executed when a tree view node is collapsed (i.e. closed).

This data block is used to implement dynamic trees in a `DISPLAY ARRAY`, where nodes are removed according to the nodes closed by the end user.

DIALOG control blocks
Dialog control blocks are predefined dialog triggers where you can implement specific code to control the interactive instruction.

The code could involve using `ui.Dialog` class methods or dialog specific instructions such as `NEXT FIELD` or `CONTINUE DIALOG`.

- Control block execution order in parallel dialogs on page 1220
- BEFORE FIELD block on page 1069
- AFTER FIELD block on page 1070
- ON CHANGE block on page 1069
- BEFORE INPUT block on page 1067
- AFTER INPUT block on page 1068
- BEFORE CONSTRUCT block on page 1135
- AFTER CONSTRUCT block on page 1135
- BEFORE DISPLAY block on page 1082
- AFTER DISPLAY block on page 1083
- BEFORE ROW block on page 1083
- ON ROW CHANGE block on page 1110
- AFTER ROW block on page 1084
- BEFORE INSERT block on page 1113
- AFTER INSERT block on page 1113
- BEFORE DELETE block on page 1114
- AFTER DELETE block on page 1114
- BEFORE MENU block on page 1054

Control block execution order in multiple dialogs

This table shows the order in which control blocks are executed in a procedural `DIALOG` instruction, according to the context and user action:

Table 281: Control block execution order for a procedural dialog

Context / User action	Control Block execution order
Entering the dialog	**1.** `BEFORE DIALOG` **2.** `BEFORE INPUT`, `BEFORE CONSTRUCT` or `BEFORE DISPLAY` (first sub-dialog getting focus) **3.** `BEFORE ROW` (if focus goes to a list) **4.** `BEFORE FIELD` (if focus goes to a field)
When the focus goes to an `INPUT`or to a `CONSTRUCT` from a different sub-dialog	**1.** *Triggers raised by the context of the sub-dialog you leave* **2.** `BEFORE INPUT` or `BEFORE CONSTRUCT` (new sub-dialog getting focus)

Context / User action	Control Block execution order
	3. `BEFORE FIELD`
When the focus leaves an `INPUT` or a `CONSTRUCT` to a different sub-dialog	**1.** `ON CHANGE` (if INPUT and value of current field has changed) **2.** `AFTER FIELD` (for the current field) **3.** `AFTER INPUT` or `AFTER CONSTRUCT` (current sub-dialog losing focus) **4.** *Triggers raised by the context of the sub-dialog you enter*
When the focus goes to a `DISPLAY ARRAY` list or to an `INPUT ARRAY` list from a different sub-dialog	**1.** *Triggers raised by the context of the sub-dialog you leave* **2.** `BEFORE INPUT` or `BEFORE DISPLAY` (new sub-dialog getting focus) **3.** `BEFORE ROW` (the row that was selected in the list) **4.** `BEFORE FIELD` (if it's an INPUT ARRAY)
When the focus leaves a `DISPLAY ARRAY` or `INPUT ARRAY` list to a different sub-dialog	**1.** `ON CHANGE` (if INPUT ARRAY and value of current field has changed) **2.** `AFTER FIELD` (if it's an INPUT ARRAY) **3.** `AFTER INSERT` (if INPUT ARRAY and current row was just created) or `ON ROW CHANGE` (if INPUT ARRAY and a value in the row has changed) **4.** `AFTER ROW` (the current row in the list you leave) **5.** `AFTER INPUT` or `AFTER DISPLAY` (current sub-dialog losing focus) **6.** *Triggers raised by the context of the sub-dialog you enter*
Moving from field A to field B in an `INPUT` or `CONSTRUCT` sub-dialog or in the same row of an `INPUT ARRAY` list	**1.** `ON CHANGE` (if value of current field has changed) **2.** `AFTER FIELD A` **3.** `BEFORE FIELD B`
Moving from field A of an `INPUT` or `CONSTRUCT` sub-dialog to field B in another `INPUT` or `CONSTRUCT` sub-dialog	**1.** `ON CHANGE` (if value of current field has changed) **2.** `AFTER FIELD A` **3.** `AFTER INPUT` or `AFTER CONSTRUCT` (for sub-dialog of field A) **4.** `BEFORE INPUT` or `BEFORE CONSTRUCT` (for sub-dialog of field B) **5.** `BEFORE FIELD B`
Moving to a different row in a `DISPLAY ARRAY` list	**1.** `AFTER ROW` (the row you leave) **2.** `BEFORE ROW` (the new current row)

Context / User action	**Control Block execution order**
Moving to a different row in an `INPUT ARRAY` list when current row was newly created	**1.** `ON CHANGE` (if value of current field has changed) **2.** `AFTER FIELD` (for field A in the row you leave) **3.** `AFTER INSERT` (the newly created row) **4.** `AFTER ROW` (the newly created row) **5.** `BEFORE ROW` (the new current row) **6.** `BEFORE FIELD` (field in the new current row)
Moving to a different row in an `INPUT ARRAY` list when current row was modified	**1.** `ON CHANGE` (if value of current field has changed) **2.** `AFTER FIELD` (for field A in the row you leave) **3.** `ON ROW CHANGE` (the values in current row have changed) **4.** `AFTER ROW` (for the row that was modified) **5.** `BEFORE ROW` (the new current row) **6.** `BEFORE FIELD` (field in the new current row)
Inserting or appending a new row in an `INPUT ARRAY` list	**1.** *Triggers raised by the context of the sub-dialog you leave* **2.** `BEFORE INSERT` (for the new current row) **3.** `BEFORE ROW` (the new current row) **4.** `BEFORE FIELD` (field in the new current row)
Deleting a row in an `INPUT ARRAY` list	**1.** `BEFORE DELETE` (for the current row to be deleted) **2.** `AFTER DELETE` (now the deleted row is removed) **3.** `AFTER ROW` (for the deleted row) **4.** `BEFORE ROW` (the new current row)
Firing the *insert* or *append* action for the `ON INSERT` block in a `DISPLAY ARRAY` list	**1.** `AFTER ROW` **2.** `ON INSERT` **3.** `BEFORE ROW`
Firing the *delete* action for the `ON DELETE` block in a `DISPLAY ARRAY` list	**1.** `AFTER ROW` **2.** `ON DELETE` **3.** `BEFORE ROW`
Validating the dialog with `ACCEPT DIALOG`	**1.** `ON CHANGE` (if focus is in input field and value has changed) **2.** `AFTER FIELD` (if focus is in input field) **3.** `AFTER INSERT` (if INPUT ARRAY and current row was just created) or `ON ROW CHANGE` (if INPUT ARRAY and a value in the row has changed) **4.** `AFTER ROW` (if focus is in a list) **5.** `AFTER INPUT`, `AFTER CONSTRUCT` or `AFTER CONSTRUCT` (current sub-dialog)

Context / User action	Control Block execution order
	6. AFTER DIALOG
Canceling the dialog with EXIT DIALOG	None of the control blocks will be executed; we just leave the dialog instruction.

BEFORE DIALOG block

The BEFORE DIALOG block is executed one time as the first trigger when the DIALOG instruction starts, before the runtime system gives control to the user. You can implement variable initialization and dialog configuration in this block.

In this example, the BEFORE DIALOG block performs some dialog setup and gives the focus to a specific field:

```
BEFORE DIALOG
  CALL DIALOG.setActionActive("save",FALSE)
  CALL DIALOG.setFieldActive("cust_status", is_admin())
  IF cust_is_new() THEN
     NEXT FIELD cust_name
  END IF
```

A DIALOG instruction can include no more than one BEFORE DIALOG control block.

AFTER DIALOG block

The AFTER DIALOG block is executed one time as the last trigger when the DIALOG instruction terminates, when performing an ACCEPT DIALOG instruction. Dialog finalization code can be implemented in this block.

The dialog terminates when an ACCEPT DIALOG or EXIT DIALOG control instruction is executed. However, the AFTER DIALOG block is not executed if an EXIT DIALOG is performed.

If you execute one of the following control instructions in an AFTER DIALOG block, the dialog will not terminate and it will give control back to the user:

1. NEXT FIELD
2. NEXT OPTION
3. CONTINUE DIALOG

In the next example, the AFTER DIALOG block checks whether a field value is correct and gives control back to the dialog if the value is wrong:

```
ON ACTION accept
   ACCEPT DIALOG
...
AFTER DIALOG
  IF NOT cust_is_valid_status(p_cust.cust_status) THEN
     ERROR "Customer state is not valid"
     NEXT FIELD cust_status
  END IF
```

BEFORE FIELD block

For fields controlled by an INPUT, INPUT ARRAY or by a CONSTRUCT instructions, the BEFORE FIELD block is executed every time the cursor enters into the specified field.

For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The BEFORE FIELD block is also executed when performing a NEXT FIELD instruction.

The BEFORE FIELD keywords must be followed by a list of form field specification. The screen-record name can be omitted.

BEFORE FIELD is executed after BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW and BEFORE INSERT.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
   BEFORE FIELD cust_status
     LET p_cust.cust_comment = NULL
     MESSAGE "Enter customer status"
```

When using the default FIELD ORDER CONSTRAINT mode, the dialog executes the BEFORE FIELD block of the field corresponding to the first variable of an INPUT or INPUT ARRAY, even if that field is not editable (NOENTRY, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of INPUT ARRAY. This behavior is supported for backward compatibility. The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing the first time with a specific form, the BEFORE FIELD block might be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column would have been done in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

AFTER FIELD block

In dialog parts driven by a simple INPUT, INPUT ARRAY or by a CONSTRUCT sub-dialog, the AFTER FIELD block is executed every time the focus leaves the specified field. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, based on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT or ACCEPT CONTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
   AFTER FIELD item_quantity
     IF p_item.item_quantity <= 0 THEN
        ERROR "Item quantity cannot be negative or zero"
        LET p_item.item_quantity = 0
        NEXT FIELD item_quantity
     END IF
```

ON CHANGE block

The ON CHANGE block can be used to detect that a field changed by user input. The ON CHANGE block is executed if the value has changed since the field got the focus and if the modification flag is set. The ON CHANGE block can only be used for fields controlled by an INPUT or INPUT ARRAY dialog, it is not available in CONSTRUCT.

For editable fields defined as EDIT, TEXTEDIT or BUTTONEDIT, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and if the modification flag is set for the field. You leave the field when you validate the dialog, when you move to another field, or when you move to another row in an INPUT ARRAY. However, if the text edit field is defined with the COMPLETER attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

```
   ON CHANGE order_checked -- Defined as CHECKBOX
      CALL setup_dialog(DIALOG)
```

If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing the value of the current field by program in an ON ACTION block, the ON CHANGE block will be executed when leaving the field if the value is different from the reference value and if the modification flag is set (after previous user input or when the touched flag has been changed by program).

When using the NEXT FIELD instruction, the comparison value is reassigned as if the user had leaved and reentered the field. Therefore, when using NEXT FIELD in ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be invoked again if the value is different from the reference value. This denies to do field validation in ON CHANGE blocks: you must do validations in AFTER FIELD blocks and/or AFTER INPUT blocks.

BEFORE INPUT block

BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE INPUT is only executed once when the dialog is started.

The BEFORE INPUT block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
INPUT BY NAME cust_rec.* ...
   BEFORE INPUT
       MESSAGE "Input customer information"
       CALL DIALOG.setActionActive("check_info", is_super_user() )
       CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
   ...
```

The fields are initialized with the defaults values before the BEFORE INPUT block is executed. When the INPUT instruction uses the WITHOUT DEFAULTS option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the DEFAULT attributes of the form fields are used.

Use the NEXT FIELD control instruction in the BEFORE INPUT block, to jump to a specific field when the dialog starts.

BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE INPUT` block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not invoke the `BEFORE INPUT` block.

`BEFORE INPUT` is executed after the `BEFORE DIALOG` block andbefore the `BEFORE ROW`, `BEFORE FIELD` blocks.

In this example, the `BEFORE INPUT` block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
   BEFORE INPUT
       CALL DIALOG.setActionActive("validate_order", TRUE)
```

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular `INPUT`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `AFTER INPUT` is only executed once when dialog ends.

The `AFTER INPUT` block is executed after the user has validated or canceled the `INPUT` or `INPUT ARRAY` dialog with the accept or cancel default actions, or when the `ACCEPT INPUT` instruction is executed.

The `AFTER INPUT` block is not executed when the `EXIT INPUT` instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules depending from several fields. If the values entered by the user do not satisfy the constraints, use the `NEXT FIELD` instruction to force the dialog to continue. The `CONTINUE INPUT` instruction can be used instead of `NEXT FIELD`, when no particular field has to be select.

Before checking the validation rules, make sure that the `INT_FLAG` variable is `FALSE`: in case if the user cancels the dialog, the validation rules must be skipped.

```
INPUT BY NAME cust_rec.*
   WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
  ...

  AFTER INPUT
    IF NOT INT_FLAG THEN
       IF cust_rec.cust_address IS NOT NULL
          AND cust_rec.cust_zipcode IS NULL THEN
          ERROR "Address is incomplete, enter a zipcode."
          NEXT FIELD zipcode
       END IF
    END IF
END INPUT
```

To limit the validation to fields that have been modified by the end user, you can call the `FIELD_TOUCHED()` function or the `DIALOG.getFieldTouched()` method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER INPUT` block is executed when the focus is lost by a group of fields driven by an `INPUT` or `INPUT ARRAY` sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the

focus. When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the AFTER INPUT block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

AFTER INPUT is executed after the AFTER FIELD, AFTER ROW blocks and before the AFTER DIALOG block.

Executing a NEXT FIELD in the AFTER INPUT control block will keep the focus in the group of fields. Within an INPUT ARRAY sub-dialog, NEXT FIELD will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the AFTER INPUT block is used to validate data and disable an action that can only be used in the current group:

```
INPUT BY NAME p_order.*
   AFTER INPUT
      IF NOT check_order_data(DIALOG) THEN
        NEXT FIELD CURRENT
      END IF
      CALL DIALOG.setFieldActive("validate_order", FALSE)
```

BEFORE CONSTRUCT block

BEFORE CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular CONSTRUCT instruction, or when used as parallel dialog, the BEFORE CONSTRUCT is only executed once when dialog is started.

The BEFORE CONSTRUCT block is executed once at dialog startup, before the runtime system gives control to the user for criteria input. This block can be used to display messages to the user, initialize form fields with default search criteria values, and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
CONSTRUCT BY NAME where_part ON ...
   BEFORE CONSTRUCT
       MESSAGE "Enter customer search filter"
       CALL DIALOG.setActionActive("clean", FALSE )
   ...
```

The fields are cleared before the BEFORE CONSTRUCT block is executed.

You can use the NEXT FIELD control instruction in the BEFORE CONSTRUCT block, to jump to a specific field when the dialog starts.

BEFORE CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a CONSTRUCT sub-dialog of a procedural DIALOG instruction, the BEFORE CONSTRUCT block is executed when the focus goes to a group of fields driven by a CONSTRUCT sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

BEFORE CONSTRUCT is executed after the BEFORE DIALOG block and before the BEFORE FIELD blocks.

In this example, the BEFORE CONSTRUCT block is used to display a message:

```
CONSTRUCT BY NAME sql ON customer.*
  BEFORE CONSTRUCT
    MESSAGE "Enter customer search filter"
```

AFTER CONSTRUCT block

AFTER CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular `CONSTRUCT` instruction, or when used as parallel dialog, the `AFTER CONSTRUCT` is only executed once when dialog is ended.

Use an `AFTER CONSTRUCT` block to execute instructions after the user has finished search criteria input.

`AFTER CONSTRUCT` is not executed if an `EXIT CONSTRUCT` is performed.

The code in `AFTER CONSTRUCT` can for example check if a criteria combination of different fields is required or denied, and force the end use to enter all

Before checking the content of the fields used in the `CONSTRUCT`, make sure that the `INT_FLAG` variable is `FALSE`: in case if the user cancels the dialog, the validation rules must be skipped.

Since no program variables are associated to the form fields, you must query the input buffers of the fields to get the values entered by the user.

```
CONSTRUCT BY NAME where_part ON ...
  ...
  AFTER CONSTRUCT
    IF NOT INT_FLAG THEN
       IF length(DIALOG.getFieldBuffer(cust_name))==0
       OR length(DIALOG.getFieldBuffer(cust_addr))==0 THEN
          ERROR "Enter a search criteria for customer name and address
 fields."
          NEXT FIELD CURRENT
       END IF
    END IF
END CONSTRUCT
```

To limit the validation to fields that have been modified by the end user, you can call the `FIELD_TOUCHED()` function or the `DIALOG.getFieldTouched()` method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a `CONSTRUCT` sub-dialog of a procedural `DIALOG` instruction, the `AFTER CONSTRUCT` block is executed when the focus is lost by a group of fields driven by a `CONSTRUCT` sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

`AFTER CONSTRUCT` is executed after the `AFTER FIELD` and before the `AFTER DIALOG` block.

Executing a `NEXT FIELD` in the `AFTER CONSTRUCT` control block will keep the focus in the group of fields.

In this example, the `AFTER CONSTRUCT` block is used to build the SELECT statement:

```
CONSTRUCT BY NAME sql ON customer.*
   AFTER CONSTRUCT
       LET sql = "SELECT * FROM customers WHERE " || sql
```

BEFORE DISPLAY block

BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a singular `DISPLAY ARRAY` instruction, or when used as parallel dialog, the `BEFORE DISPLAY` is only executed once when the dialog is started.

The `BEFORE DISPLAY` block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating actions the user is not allowed to execute.

```
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", is_super_user())
```

BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a `DISPLAY ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE DISPLAY` block is executed when a `DISPLAY ARRAY` list gets the focus.

`BEFORE DISPLAY` is executed before the `BEFORE ROW` block.

In this example the `BEFORE DISPLAY` block enables an action and displays a message:

```
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("print_list", TRUE)
       MESSAGE "You are now in the list of items"
```

AFTER DISPLAY block

AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a singular `DISPLAY ARRAY` instruction, or when used as parallel dialog, the `AFTER DISPLAY` is only executed once when dialog is ended.

You typically implement dialog finalization in this block.

```
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       DISPLAY "Current row is: ", arr_curr()
```

AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a `DISPLAY ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER DISPLAY` block is executed when a `DISPLAY ARRAY` list loses the focus and goes to another sub-dialog.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

`AFTER DISPLAY` is executed after the `AFTER ROW` block.

In this example, the `AFTER DISPLAY` block disables an action that is specific to the current list:

```
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", FALSE)
```

BEFORE ROW block

BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE ROW block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, BEFORE ROW will be executed for the current row, but only if there are data rows in the array.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example, the BEFORE ROW block gets the new row number and displays it in a message:

```
DISPLAY ARRAY ...
   ...
   BEFORE ROW
     MESSAGE "We are on row # ", arr_curr()
   ...
```

BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list gets the focus, or when the user moves to another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the BEFORE ROW block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the BEFORE DISPLAY or BEFORE INPUT blocks instead.

In DISPLAY ARRAY, BEFORE ROW is executed after the BEFORE DISPLAY block. In INPUT ARRAY, BEFORE ROW is executed before the BEFORE INSERT and BEFORE FIELD blocks and after the BEFORE INPUT blocks.

When the procedural dialog starts, BEFORE ROW will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, BEFORE ROW will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular DISPLAY ARRAY or INPUT ARRAY. In singular dialogs, the BEFORE ROW block is always executed when the dialog starts (and there are rows in the array).

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row.

In this example the BEFORE ROW block displays a message with the current row number:

```
   DISPLAY ARRAY p_items TO s_items.*
     BEFORE ROW
       MESSAGE "We are in items, on row #", DIALOG.getCurrentRow("s_items")
```

ON ROW CHANGE block

The ON ROW CHANGE block is executed in a list controlled by an INPUT ARRAY, when leaving the current row and when the row has been modified since it got the focus. This is typically used to detect row modification.

The code in ON ROW CHANGE will not be executed when leaving new rows created by the user with the default append or insert action. To detect row creation, you must use the BEFORE INSERT or AFTER INSERT control blocks.

The ON ROW CHANGE block is only executed if at least one field value in the current row has changed since the row was entered, and the modification flag of the field is set. The modified field(s) might not be the current field, and several field values can be changed. Values might have been changed by the user or by the program. The modification flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

ON ROW CHANGE is executed after the AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the current row that has been changed.

You can, for example, code database modifications (UPDATE) in the ON ROW CHANGE block:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   ON ROW CHANGE
     LET r = DIALOG.getCurrentRow("s_items")
     UPDATE items SET
             items.item_code        = p_items[r].item_code,
             items.item_description = p_items[r].item_description,
             items.item_price       = p_items[r].item_price,
             items.item_updatedate  = TODAY
            WHERE items.item_num = p_items[r].item_num
```

AFTER ROW block

AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular DISPLAY ARRAY, INPUT ARRAY instruction, or when used as parallel dialog, the AFTER ROW block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A NEXT FIELD instruction executed in the AFTER ROW control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the row that you are leaving.

AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the AFTER ROW block is executed when a DISPLAY ARRAY or INPUT ARRAY list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

AFTER ROW is executed after the AFTER FIELD, AFTER INSERT and before AFTER DISPLAY or AFTER INPUT blocks.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the of the row that you are leaving.

For both INPUT ARRAY and DISPLAY ARRAY sub-dialogs, a NEXT FIELD executed in the AFTER ROW control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

AFTER ROW and temporary rows in INPUT ARRAY

> **Important: After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The `AFTER ROW` block will be executed for the temporary row, but `ui.Dialog.getCurrentRow()`/`arr_curr()` will be one row greater than `ui.Dialog.getArrayLength()`/`ARR_COUNT()`. In this case, you should ignore the `AFTER ROW` event. For example, you should not try to execute a `NEXT FIELD` or `CONTINUE INPUT` instruction, nor should you try to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.**

In this example, the `AFTER ROW` block checks the current row index and verifies a variable value to forces the focus to stay in the current row if the value is wrong:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   AFTER ROW
      LET r = DIALOG.getCurrentRow("s_items")
      IF r <= DIALOG.getArrayLength("s_items") THEN
        IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
           ERROR "Item quantity is not valid"
           NEXT FIELD item_quantity'
         END IF
      END IF
```

Another way to handle the case of temporary rows in `AFTER ROW` is to use a flag to know if the `AFTER INSERT` block was executed: The `AFTER INSERT` block is not executed if the temporary row is automatically removed. By setting a first value in `BEFORE INSERT` and changing the flag in `AFTER INSERT`, you can detect if the row was permanently added to the list:

```
INPUT ARRAY p_items FROM s_items.*
  ...
  BEFORE INSERT
    LET op = "T"
    ...
  AFTER INSERT
    LET op = "I"
    ...
  AFTER ROW
    IF op == "I" THEN
      IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
         ERROR "Item quantity is not valid"
         NEXT FIELD item_quantity
        END IF
      WHENEVER ERROR CONTINUE
      INSERT INTO items (item_num, item_name, item_quantity)
                        VALUES ( p_item[arr_curr()].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<0 THEN
         ERROR "Could not insert the record into database!"
         NEXT FIELD CURRENT
      ELSE
         MESSAGE "Record has been inserted successfully"
      END IF
   END IF
...
```

BEFORE INSERT block

The BEFORE INSERT block is executed when a new row ins created in an INPUT ARRAY. You typically use this trigger to set some default values in the new created row. A new row can be created by moving down after the last row, by executing a insert action, or by executing an append action.

The BEFORE INSERT block is executed after the BEFORE ROW block and before the BEFORE FIELD block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the new created row.

To distinguish row insertion from an appended row, compare the current row (DIALOG.getCurrentRow("*screen-array*")) with the total number of rows (DIALOG.getArrayLength("*screen-array*")). If the current row index and the total number of rows correspond, the BEFORE INSERT concerns a temporary row, otherwise it concerns an inserted row.

Row creation can be stopped by using the CANCEL INSERT instruction inside BEFORE INSERT. If possible, it is however better to disable the insert and append actions to prevent the user to execute the actions with DIALOG.setActionActive().

In this example, the BEFORE INSERT block checks if the user can create rows and denies new row creation if needed; otherwise, it sets some default values:

```
INPUT ARRAY p_items FROM s_items.*
    ...
    BEFORE INSERT
      IF NOT user_can_append THEN
         ERROR "You are not allowed to append rows"
         CANCEL INSERT
      END IF
      LET r = DIALOG.getCurrentRow("s_items")
      LET p_items[r].item_num = get_new_serial("items")
      LET p_items[r].item_name = "undefined"
```

AFTER INSERT block

The AFTER INSERT block of INPUT ARRAY is executed when the creation of a new row is validated. In this block, you can for example implement SQL to insert a new row in the database table.

The AFTER INSERT block is executed afterthe AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the new created row.

When the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the AFTER INSERT block is only executed if at least one field was edited. If no data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the AFTER INSERT block.

When executing a NEXT FIELD in the AFTER INSERT block, the dialog will keep the focus in the list and stay in the current row. Use this behavior to implement row input validation and prevent the user from leaving the list or moving to another row. However, this will not cancel the row insertion and will not invoke the BEFORE INSERT / AFTER INSERT triggers again. The only way to keep the focus in the current row after the row was inserted is to execute a NEXT FIELD in the AFTER ROW block.

In this example, the AFTER INSERT block inserts a new row in the database and cancels the operation if the SQL command fails:

```
INPUT ARRAY p_items FROM s_items.*
    ...
    AFTER INSERT
      WHENEVER ERROR CONTINUE
```

```
      INSERT INTO items VALUES
 ( p_items[DIALOG.getCurrentRow("s_items")].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<>0 THEN
         ERROR SQLERRMESSAGE
         CANCEL INSERT
      END IF
```

BEFORE DELETE block

The BEFORE DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, before the row is removed from the list.

You typically code the database table synchronization in the BEFORE DELETE block, by executing a DELETE SQL statement using the primary key of the current row. In the BEFORE DELETE block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.

The BEFORE DELETE block is executed before the AFTER DELETE block.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the row that will be deleted.

The next example uses the BEFORE DELETE block to remove the row from the database table and cancels the deletion operation if an SQL error occurs:

```
INPUT ARRAY p_items FROM s_items.*
  BEFORE DELETE
    LET r = DIALOG.getCurrentRow("s_items")
    WHENEVER ERROR CONTINUE
    DELETE FROM items
       WHERE item_num = p_items[r].item_num
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<>0 VALUES
      ERROR SQLERRMESSAGE
      CANCEL DELETE
    END IF
...
```

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The AFTER DELETE block is executed after the BEFORE DELETE block and before the AFTER ROW block for the deleted row and the BEFORE ROW block of the new current row.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array (except in the special case when deleting the last row). The arr_curr() function or the ui.Dialog.getCurrentRow() method returns the same index as in BEFORE ROW, but it is the index of the new current row. The AFTER ROW block is also executed just after the AFTER DELETE block.

> **Important: When deleting the last row of the list, AFTER DELETE is executed for the delete row, and DIALOG.getCurrentRow() / arr_curr() will be one higher as DIALOG.getArrayLength() / ARR_COUNT(). You should not access a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array. When using a static array, you must ignore the values in the rows after ARR_COUNT().**

Here the AFTER DELETE block is used to re-number the rows with a new item line number (note that DIALOG.getArrayLength() / ARR_COUNT() may return zero):

```
INPUT ARRAY p_items FROM s_items.*
   AFTER DELETE
      LET r = DIALOG.getCurrentRow("s_items")
      FOR i=r TO DIALOG.getArrayLength("s_items")
          LET p_items[i].item_lineno = i
      END FOR
...
```

It is not possible to use the CANCEL DELETE instruction in an AFTER DELETE block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

DIALOG interaction blocks

Dialog interaction blocks are dialog triggers that can be used to execute specific code when the user executes an action in the dialog. For example, when pressing a button in the form, the corresponding ON ACTION interaction block will be executed.

Interaction blocks also include special handlers such as timeout event handler, drag & drop handlers, and modification triggers for DISPLAY ARRAY sub-dialogs.

- ON ACTION block on page 1056
- ON IDLE block on page 1046
- ON KEY block on page 1046
- ON APPEND block on page 1088
- ON INSERT block on page 1088
- ON UPDATE block on page 1089
- ON DELETE block on page 1090
- ON SELECTION CHANGE block on page 1090
- ON DRAG_START block on page 1091
- ON DRAG_FINISHED block on page 1091
- ON DRAG_ENTER block on page 1092
- ON DRAG_OVER block on page 1093
- ON DROP block on page 1094

ON ACTION block

The ON ACTION *action-name* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
  ON ACTION action-name
     instruction
     ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
   ON ACTION zoom
      CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as INPUT, INPUT ARRAY and CONSTRUCT, if an action is specific to a field, add the INFIELD clause to have the action automatically enabled when the corresponding field gets the focus:

```
   ON ACTION zoom INFIELD cust_city
      CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the ON ACTION handler needs to define its own attributes at the program level. This can be done by adding the ATTRIBUTES() clause of ON ACTION:

```
   ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
      CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The ON IDLE *seconds* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of ON IDLE must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the ON IDLE trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

> **Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An ON KEY (*key-name*) block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The ON KEY block is supported for backward compatibility with TUI mode applications.

An ON KEY block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, ON KEY(F5,F6) creates two actions with the names f5 and f6. Each action object will get an ACCELERATORNAME assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for ON KEY actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the ON KEY CONTROL-? statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, ON KEY statements used with ESC, TAB, UP, DOWN, LEFT, RIGHT, HELP, NEXT, PREVIOUS, INSERT, CONTROL-M, CONTROL-X, CONTROL-V, CONTROL-C and CONTROL-A should be avoided for use in GUI programs,

because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON TIMER block

The `ON TIMER` *seconds* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of `ON TIMER` must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON TIMER` trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

COMMAND [KEY] block

Use `COMMAND [KEY]` blocks as global procedural `DIALOG` action handler to execute a sequence of instructions when the user clicks on a button or presses a specific key. `COMMAND` defines the text and comment decoration attributes as well as accelerator keys for a specific action. `COMMAND` is especially useful when writing TUI programs, however, it's legal to use such handler when programming new GUI dialogs, especially when the action view (`BUTTON` in form) must take the focus.

Declaring a `COMMAND` block in `DIALOG` is similar to an `ON ACTION` block, except that `COMMAND` defines an implicit text and comment decoration attribute. The name of the action will be the command text converted to lowercase letters. For example, with the following code:

```
COMMAND "Open" "Opens a new file"
```

The name of the action will be "`open`", and the default decoration text will be "`Open`" with a capital letter.

Note that if you use an ampersand (&) in the command name, some front-ends consider the letter following & as an Alt-key accelerator, and the letter will be underscored. However the ampersand forms part of the action name. For example, `COMMAND "&Save"` will create an action with the name "`&save`". It is not recommended to use & ampersand characters in action names.

Unlike `ON KEY` actions, if no explicit action view is defined in the form, the default action view will be visible for a `COMMAND` hander (i.e. the automatic button will appear for this action on the front-end).

action defaults will be applied by using the action name. For explicit action views such as a `BUTTON` in the form layout, the text/comment defined in the corresponding action default entry will overwrite the values used in the `COMMAND` handler. When no explicit action view is defined in the form, the text/comment defined in the program `COMMAND` clause take precedence over action defaults, to display the default action view (button on action frame).

Inside `DIALOG` instruction, `COMMAND` blocks can only be defined as global dialog actions; Sub-dialog specific `COMMAND` handlers cannot be defined. When binding a form `BUTTON` to a `COMMAND` handler, the

button can get the focus and will be managed in the tabbing list, using preferably the `FIELD ORDER FORM` option.

When using the optional `KEY` clause, `COMMAND` defines also an implicit accelerator key. The key name must be specified between parentheses with `COMMAND KEY`:

```
COMMAND KEY (F5) "Open" "Opens a new file"
```

The `COMMAND KEY` syntax allows multiple key names in the syntax. When using multiple keys in an `COMMAND KEY` clause, the `DIALOG` instruction will assign the specified keys as accelerators:

```
COMMAND KEY (F5, CONTROL-P, CONTROL-Z) "Open" "Opens a new file"
```

With the above code example, the action name will be "`open`" and accelerators will be F5, CONTROL-P and CONTROL-Z.

The keys defined by program will take precedence over the accelerators defined in the action default entry corresponding to the action.

The `COMMAND [KEY]` block specification can also define a help number with the `HELP` clause, to display the corresponding text of the current help file.

```
COMMAND "Open" "Opens a new file" HELP 34
```

ON APPEND block

Similarly to the `ON INSERT` control block, the `ON APPEND` trigger can be used to enable row creation during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the append action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an `ON ACTION append` interaction block and the `ON APPEND` block is used, the compiler will stop with error -8408.

When the user fires the append action, the dialog first execute the user code of the `AFTER ROW` block if defined. Then the dialog moves to the end of the list, and creates a new row after the last existing row. After creating the row, the dialog executes the user code of the `ON APPEND` block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular `INPUT` statement, to let the end user enter data for the new created row. This is typically done with an `INPUT` binding explicitly array fields to the screen record fields. The new current row in the program array is identified with `arr_curr()`, and the current screen line in the form is defined by `SCR_LINE()`:

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON APPEND
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the `INPUT` instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the new row is kept in the program array, and the `BEFORE ROW` block is executed for the new created row.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the new row is removed from the program array, and the `BEFORE ROW` block is executed for the row that was existing at the current position, before the new row was created.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON APPEND` block.

The append action is disabled if the maximum number of rows is reached.

If needed, the `ON APPEND` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
  ON APPEND ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON INSERT block

Similarly to the `ON APPEND` control block, the `ON INSERT` trigger can be used to enable row creation during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the insert action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an `ON ACTION insert` interaction block and the `ON INSERT` block is used, the compiler will stop with error -8408.

When the user fires the insert action, the dialog first execute the user code of the `AFTER ROW` block if defined. Then the new row is created: The insert action creates a new row before current row in the list. After creating the row, the dialog executes the user code of the `ON INSERT` block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular `INPUT` statement, to let the end user enter data for the new created row. This is typically done with an `INPUT` binding explicitly array fields to the screen record fields. The new current row in the program array is identified with `arr_curr()`, and the current screen line in the form is defined by `scr_line()`:

```
DISPLAY ARRAY arr TO sr.*
 ...
  ON INSERT
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the `INPUT` instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the new row is kept in the program array, and the `BEFORE ROW` block is executed for the new created row.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the new row is removed from the program array, and the `BEFORE ROW` block is executed for the row that was existing at the current position, before the new row was created.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON INSERT` block.

The insert action is disabled if the maximum number of rows is reached.

If needed, the `ON INSERT` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
  ON INSERT ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON UPDATE block

The `ON UPDATE` trigger can be used to enable row modification during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the update action. This action can be decorated, enabled and disabled as regular actions.

You typically configure the `TABLE` container in the form by defining the `DOUBLECLICK` attribute to "update", in order to trigger the update action when the user double-clicks on a row.

If the dialog defines an `ON ACTION update` interaction block and the `ON UPDATE` block is used, the compiler will stop with error -8408.

When the user fires the *update* action, the dialog executes the user code of the ON UPDATE block.

The dialog handles only the row modification action and navigation, you must program the record input with a regular INPUT statement, to let the end user modify the data of the current row. This is typically done with an INPUT binding explicitly array fields to the screen record fields, with the WITHOUT DEFAULTS clause. The current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by scr_line():

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON UPDATE
    INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the modified values of the current row are kept in the program array.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the old values of the current row are restored in the program array.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON UPDATE block.

If needed, the ON UPDATE handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
  ON UPDATE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON DELETE block

The ON DELETE trigger can be used to enable row deletion during a DISPLAY ARRAY dialog. If this block is defined, the dialog will automatically create the delete action. This action can be decorated, enabled and disabled as regular actions.

If the dialog defines an ON ACTION delete interaction block and the ON DELETE block is used, the compiler will stop with error -8408.

When the user fires the delete action, the dialog executes the user code of the ON DELETE block.

The dialog handles only the row deletion action and navigation, you can typically program a validation dialog box to let the user confirm the deletion. The current row in the program array is identified with arr_curr():

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON DELETE
    IF fgl_winQuestion("Delete",
          "Do you want to delete this record?",
          "yes", "no|yes", "help", 0) == "no"
    THEN
       LET int_flag = TRUE
    END IF
    ...
```

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the current row is deleted from the program array, and the BEFORE ROW block is executed for the next row in the list.

- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the current row is kept in the program array, and the BEFORE ROW block is executed again for the current row.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON DELETE block.

If needed, the ON DELETE handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
  ON DELETE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON SELECTION CHANGE block

The ON SELECTION CHANGE trigger can be used to enable multi-row selection and detect when rows are selected or de-selected by the end user during a DISPLAY ARRAY dialog. If this block is defined, multi-row selection is automatically enableb. However, the feature can be enabled/disabled with the setSelectionMode() dialog method.

ON SORT block

The ON SORT interfaction block can be used to detect when rows have to be sorted in a DISPLAY ARRAY or INPUT ARRAY dialog.

ON SORT is used in two different contexts:

1. In a regular DISPLAY ARRAY / INPUT ARRAY dialog (not using paged mode), the ON SORT trigger can be used to detect that a list sort was performed. In this case, the (visual) sort is already done by the runtime system and the ON SORT block is only used to execute post-sort tasks, such as displaying current row information, by using arrayToVisualIndex() dialog method. It is also possible to get the sort column and order with the getSortKey() and getSortSelection() dialog methods.
2. In a DISPLAY ARRAY using paged mode (ON FILL BUFFER), built-in row sorting is not available because data is provided by pages. Use the ON SORT trigger to detect a sort request and perform a new SQL query to re-order the rows. In this case, sort column and order is available with the getSortKey() and getSortSelection() dialog methods. See Populating a DISPLAY ARRAY on page 1372.

ON DRAG_START block

The ON DRAG_START block is executed when the end user has begun the drag operation. If this dialog trigger has not been defined, default dragging is enabled for this dialog.

In the ON DRAG_START block, the program typically specifies the type of drag & drop operation by calling ui.DragDrop.setOperation() with "move" or "copy". This call will define the default and unique drag operation. If needed, the program can allow another type of drag operation with ui.DragDrop.addPossibleOperation(). The end user can then choose to move or copy the dragged object, if the drag & drop target allows it.

If the dragged object can be dropped outside the program, must define the MIME type and drag/drop data with ui.DragDrop.setMimeType() and ui.DragDrop.setBuffer() methods.

Example:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    CALL dnd.setOperation("move") -- Move is the default operation
    CALL dnd.addPossibleOperation("copy") -- User can toggle to copy if
 needed
    CALL dnd.setMimeType("text/plain")
    CALL dnd.setBuffer(arr[arr_curr()].cust_name)
    ...
```

```
END DISPLAY
```

ON DRAG_FINISHED block

Execution of the `ON DRAG_FINISHED` block notifies the dialog where the drag started that the drop operation has been completed or terminated.

Call `ui.DragDrop.getOperation()` to get the final type of operation of the drop. On successful completion, the method returns "move" or "copy"; otherwise the function returns `NULL`. If `NULL` is returned, the `ON DRAG_FINISHED` trigger can be ignored.

In cases of successful moves to a target out of the current `DISPLAY ARRAY`, the application must remove the transferred data from the source model. For example, if a row was moved from dialog A to B, dialog A will get an `ON DRAG_FINISHED` execution after the row was dropped into B, and should remove the row from the list A.

The `ON DRAG_FINISHED` interaction block is optional.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    LET last_dragged_row = arr_curr()
    ...
  ON DRAG_FINISHED (dnd)
    IF dnd.getOperation() == "move" THEN
      CALL DIALOG.deleteRow(last_dragged_row)
    END IF
    ...
END DISPLAY
```

ON DRAG_ENTER block

When the `ON DROP` control block is defined, the `ON DRAG_ENTER` block will be executed when the mouse cursor enters the visual boundaries of the drop target dialog. Entering the target dialog is accepted by default if no `ON DRAG_ENTER` block is defined. However, when `ON DROP` is defined, you should also define `ON DRAG_ENTER` to deny the drop of objects with an unsupported MIME type that come from other applications.

The program can decide to deny or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a `NULL` to the method will deny drop.

To check what MIME type is available in the drag & drop buffer, the program uses the `ui.DragDrop.selectMimeType()` method. This method takes the MIME type as a parameter and returns `TRUE` if the passed MIME type is used. You can call this method several times to check the availability of different MIME types.

You may also define the visual effect when flying over the target list with `ui.DragDrop.setFeedback()`.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    IF dnd.selectMimeType("text/plain") THEN
      CALL dnd.setOperation("copy")
      CALL dnd.setFeedback("all")
    ELSE
      CALL dnd.setOperation(NULL)
    END IF
  ON DROP (dnd)
```

```
    ...
END DISPLAY
```

Once the mouse has entered the target area, subsequent mouse cursor moves can be detected with the `ON DRAG_OVER` trigger.

When using a table or tree-view as drop target, you can control the visual effect when the mouse flies over the rows, according to the type of drag & drop you want to achieve.

Basically, a dragged object can be:

1. Inserted in between two rows (visual effect must show where the object will be inserted)
2. Copied/merged to the current row (visual effect must show the row under the mouse)
3. Dropped somewhere on the target widget (the exact location inside the widget does not matter)

The visual effect can be defined with the `ui.DragDrop.setFeedback()` method, typically called in the `ON DRAG_ENTER` block.

The values to pass to the `setFeedback()` method to get the desired visual effects described are respectively:

1. `insert` (default)
2. `select`
3. `all`

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    IF canDrop() THEN
      CALL dnd.setOperation(NULL)
    ELSE
      CALL dnd.setFeedback("select")
    END IF
    ...
END DISPLAY
```

ON DRAG_OVER block

When the `ON DROP` control block is defined, the `ON DRAG_OVER` block will be executed after `ON DRAG_ENTER`, when the mouse cursor is moving over the drop target, or when the drag & drop operation has changed (toggling copy/move).

`ON DRAG_OVER` will be called only once per row, even if the mouse cursor moves over the row.

In the `ON DRAG_OVER` block, the method `ui.DragDrop.getLocationRow()` returns the index of the row in the target array, and can be used to allow or deny the drop. When using a tree-view, you must also check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and allow/deny the drop operation accordingly.

The program can change the drop operation at any execution of the `ON DRAG_OVER` block. You can deny or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a `NULL` to the method will deny the drop.

The current operation (returned by `ui.DragDrop.getOperation()`) is the value set in previous `ON DRAG_ENTER` or `ON DRAG_OVER` events, or the operation selected by the end user, if it can toggle between copy and move. Thus, `ON DRAG_OVER` can occur even if the mouse position has not changed.

If dropping has been denied with `ui.DragDrop.setOperation(NULL)` in the previous `ON DRAG_OVER` event, the program can reset the operation to allow a drop with a call to `ui.DragDrop.setOperation()` with the operation parameter "move" or "copy".

`ON DRAG_OVER` will not be called if drop has been disabled in `ON DRAG_ENTER` with `ui.DragDrop.setOperation(NULL)`

`ON DRAG_OVER` is optional, and must only be defined if the operation or the acceptance of the drag object depends on the target row of the drop target.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    ...
  ON DRAG_OVER (dnd)
    IF arr[dnd.getLocationRow()].acceptsCopy THEN
      CALL dnd.setOperation("copy")
    ELSE
      CALL dnd.setOperation(NULL)
    END IF
  ON DROP (dnd)
    ...
END DISPLAY
```

During a drag & drop process, the end user (or the target application) can decide to modify the type of the operation, to indicate whether the dragged object has to be copied or moved from the source to the target. For example, in a typical file explorer, by default files are moved when doing a drag & drop on the same disk. To make a copy of a file, you must press the Ctrl key while doing the drag & drop with the mouse.

In the drop target dialog, you can detect such operation changes in the `ON DRAG_OVER` trigger and query the `ui.DragDrop` object for the current operation with `ui.DragDrop.getOperation()`. In the drag source dialog, you typically check `ui.DragDrop.getOperation()` in the `ON DRAG_FINISHED` trigger to know what sort of operation occurred, to keep ("copy" operation) or delete ("move" operation) the original dragged object.

This example tests the current operation in the drop target list and displays a message accordingly:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    ...
  ON DRAG_OVER (dnd)
    CASE dnd.getOperation()
    WHEN "move"
      MESSAGE "The object will be moved to row ", dnd.getLocationRow()
    WHEN "copy"
      MESSAGE "The object will be copied to row ", dnd.getLocationRow()
    END CASE
    ...
  ON DROP (dnd)
    ...
END DISPLAY
```

ON DROP block

To enable drop actions on a list, you must define the `ON DROP` block; otherwise the list will not accept drop actions.

The `ON DROP` block is executed after the end user has released the mouse button to drop the dragged object. `ON DROP` will not occur if drop has been denied in the previous `ON DRAG_OVER` event or in `ON DRAG_ENTER` with a call to `ui.DragDrop.setOperation(NULL)`.

The program might also check the MIME type of the dragged object with `ui.DragDrop.getSelectedMimeType()`, and then call the `ui.DragDrop.getBuffer()` method to retrieve drag & drop data from external applications.

Ideally the drop operation should be accepted (no additional call to `ui.DragDrop.setOperation()`).

In this block, the `ui.DragDrop.getLocationRow()` method returns the index of the row in the target array, and can be used to execute the code to get the drop data / object into the row that has been chosen by the user.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DROP (dnd)
    LET arr[dnd.getLocationRow()].capacity == dnd.getBuffer()
    ...
END DISPLAY
```

If the drag & drop operations are local to the same list or tree-view controller, you can use the `ui.DragDrop.dropInternal()` method to simplify the code. This method implements the typical move of the dragged rows or tree-view node. This is especially useful in case of a tree-view, but is also the preferred way to move rows around in simple tables.

This `ON DROP` code example uses the `dropInternal()` method:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr_tree TO sr_tree.* ...
  ...
  ON DROP (dnd)
    CALL dnd.dropInternal()
    ...
END DISPLAY
```

If you want to implement by hand the code to drop a node in a tree-view, you must check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and execute the code corresponding to the drop operation: If the drop target row index returned by `getLocationRow()` is a child of the parent row index returned by `getLocationParent()` , the new row must be inserted before `getLocationRow()`; otherwise the new row must be added as a child of the parent node identified by `getLocationParent()`.

DIALOG control instructions

Dialog control instructions are language instructions dedicated to dialog control, to programmatically force the dialog to behave in a given way.

For example the `NEXT FIELD` instruction forces the focus to a specific form field.

- NEXT FIELD instruction on page 1121
- CLEAR instruction in dialogs on page 1073
- DISPLAY TO / BY NAME instruction on page 1192
- CONTINUE DIALOG instruction on page 1192
- EXIT DIALOG instruction on page 1193
- ACCEPT DIALOG instruction on page 1193
- CANCEL DELETE instruction on page 1119
- CANCEL INSERT instruction on page 1120

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The `NEXT FIELD` *`field-name`* instruction gives the focus to the specified field and forces the dialog to stay in that field.

This instruction can be used to control field input, in `BEFORE FIELD`, `ON CHANGE` or `AFTER FIELD` blocks, it can also force a `DISPLAY ARRAY` or `INPUT ARRAY` to stay in the current row when `NEXT FIELD` is used in the `AFTER ROW` block.

If it exists, the `BEFORE FIELD` block of the corresponding field is executed.

The purpose of the `NEXT FIELD` instruction is give the focus to an editable field. Make sure that the field specified in `NEXT FIELD` is active, or use `NEXT FIELD CURRENT`. Non-editable fields are fields defined with the `NOENTRY` attribute, fields disabled at runtime with `DIALOG.setFieldActive()`, or fields using a widget that does not allow input, such as a `LABEL`.

Instead of the `NEXT FIELD` instruction, you can use the `DIALOG.nextField("`*`field-name`*`")` method to register a field, for example when the name is not known at compile time. However, this method only registers the field: It does not stop code execution, like the `NEXT FIELD` instruction does. You must execute a `CONTINUE DIALOG` to get the same behavior as `NEXT FIELD`.

Form field identification with NEXT FIELD

With the `NEXT FIELD` instruction, fields are identified by the form field name specification, not the program variable name used by the dialog. Form fields are bound to program variables with the binding clause of dialog instruction (`INPUT` *`variable-list`* `FROM` *`field-list`*, `INPUT BY NAME` *`variable-list`*, `CONSTRUCT BY NAME` *`sql`* `ON` *`column-list`*,`CONSTRUCT` *`sql`* `ON` *`column-list`* `FROM` *`field-list`*, `INPUT ARRAY` *`array-name`* `FROM` *`screen-array`*`.*`).

The field name specification can be any of the following:

- *field-name*
- *table-name.field-name*
- *screen-record-name.field-name*
- `FORMONLY.`*`field-name`*

Here are some examples:

- `"cust_name"`
- `"customer.cust_name"`
- `"cust_screen_record.cust_name"`
- `"item_screen_array.item_label"`
- `"formonly.total"`

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog according to the field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using `INPUT BY NAME` *`variable-list`*), the field prefix must be the database table name (or `FORMONLY`) used in the form file, or any valid screen-record using that field. When the `FROM` clause of the dialog specifies an explicit screen-record (for example, in `INPUT` *`variable-list`* `FROM` *`screen-record.* / field-list-with-screen-record-prefix`* or `INPUT ARRAY` *`array-name`* `FROM` *`screen-array`*`.*`), the field prefix must be the screen-record name used in the `FROM` clause.

Abstract field identification is supported with the `CURRENT`, `NEXT` and `PREVIOUS` keywords. These keywords represent the current, next and previous fields respectively. When using `FIELD ORDER FORM`, the `NEXT` and `PREVIOUS` options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the `FROM` or `BY NAME` clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD PREVIOUS will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable INPUT / INPUT ARRAY / CONSTRUCT field

Non-editable fields are fields defined with the NOENTRY attribute, fields disabled with ui.Dialog.setFieldActive("*field-name*", FALSE), or fields using a widget that does not allow input, such as a LABEL.

If a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog. Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; Use NEXT FIELD CURRENT instead.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD *field-name* instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW and AFTER INPUT/DISPLAY/CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT/DISPLAY/CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user. This behavior allows you to implement data input rules:

```
AFTER ROW
  IF NOT int_flag AND arr_count()<=arr_curr() THEN
     IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
        ERROR "Amount of line exceeds max value."
        NEXT FIELD item_count
```

```
       END IF
    END IF
```

CLEAR instruction in dialogs

The `CLEAR` *`field-list`* and `CLEAR SCREEN ARRAY` *`screen-array.*`* instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program variables bound to the dialog are left unchanged. `CLEAR` can be used outside any dialog instruction, such as the `DISPLAY BY NAME / TO` instructions.

When a dialog is configured with the `UNBUFFERED` mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with `DISPLAY` or `CLEAR` instruction in an `UNBUFFERED` dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an `UNBUFEFERED` dialog, just set to `NULL` the variables bound to the dialog. However, when using a `CONSTRUCT`, no program variables are associated to the dialog and no `UNBUFFERED` concept exits, and the `CLEAR` or `DISPLAY TO / BY NAME` instructions are the only way to modify the `CONSTRUCT` fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using `TABLE` containers, you can therefore use the `CLEAR SCREEN ARRAY` instruction to clear all rows of a list.

DISPLAY TO / BY NAME instruction

The `DISPLAY` *`variable-list`* `TO` *`field-list`* or `DISPLAY BY NAME` *`variable-list`* instruction fills the value buffers of specified form fields with the values contained in the specified program variables. The `DISPLAY` instruction changes the buffers directly in the current form, not the program variables bound to the dialog. `DISPLAY` can be used outside any dialog instruction, in the same way as the `CLEAR` instruction. `DISPLAY` also sets the modification flag of fields.

As `DIALOG` is typically used with the `UNBUFFERED` mode, there is no reason to set field buffers in a `DIALOG` block since any variable assignment will synchronize field buffers. Actually, changing the field buffers with the `DISPLAY` or `CLEAR` instruction will have no visual effect if the fields are used by a dialog working in `UNBUFFERED` mode, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. So if you want to set field values, just assign the variables and the fields will be synchronized. However, when using a `CONSTRUCT` binding, you may want to set field buffers with this `DISPLAY` instruction, as there are no program variables bound to fields (with `CONSTRUCT`, only one string variable is bound to hold the SQL condition).

Instead of using a `DISPLAY` instruction to set the modification flag of fields to simulate user input, use the `DIALOG.setFieldTouched()` method instead.

CONTINUE DIALOG instruction

The `CONTINUE DIALOG` statement continues the execution of a `DIALOG` instruction, skipping all statements appearing after this instruction.

Control returns to the dialog instruction, which executes remaining control blocks as if the program reached the end of the current control block. Then the control goes back to the user and the dialog waits for a new event.

The `CONTINUE DIALOG` statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user by skipping the rest of the statements.

In the following code example, an `ON ACTION` block gives control back to the dialog, skipping all instructions below line 04:

```
ON ACTION zoom
  IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
    ERROR "Zoom window cannot be opened if no info to identify customer"
    CONTINUE DIALOG
  END IF
  IF p_cust.cust_address IS NULL THEN
     ...
```

If CONTINUE DIALOG is called in a control block that is not AFTER DIALOG, further control blocks might be executed according to the context. Actually, CONTINUE DIALOG just instructs the dialog to continue as if the code in the control block was terminated (it is a kind of GOTO end_of_control_block). However, when executed in AFTER DIALOG, the focus returns to the current field or read-only list. In this case the BEFORE ROW and BEFORE FIELD triggers will be invoked.

A CONTINUE DIALOG in AFTER FIELD, AFTER INPUT, AFTER DISPLAY or AFTER CONSTRUCT will only stop the program flow of the current block of statements; instructions after CONTINUE DIALOG will not be executed. If the user has selected a field in a different sub-dialog, this new field will get the focus and all necessary AFTER / BEFORE control blocks will be executed.

In case of input error in a field, the best practice is to use a NEXT FIELD instruction to stay in the dialog and set the focus to the field that the user has to correct.

EXIT DIALOG instruction

The EXIT DIALOG statement terminates a procedural DIALOG block without any further control block execution.

> **Note:** When used in a declarative DIALOG block, the EXIT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

Program flow resumes at the instruction following the END DIALOG keywords. Blocks such as AFTER DIALOG will not be executed.

```
   ON ACTION quit
       EXIT DIALOG
```

When leaving the DIALOG instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

ACCEPT DIALOG instruction

The ACCEPT DIALOG statement validates all input fields bound to the DIALOG instruction and leaves the block if no error is raised.

> **Note:** When used in a declarative DIALOG block, the ACCEPT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

When defined in the dialog block, ON CHANGE, AFTER FIELD, AFTER ROW, AFTER INPUT/DISPLAY/CONSTRUCT control blocks will be executed when ACCEPT DIALOG is performed.

The statements appearing after the ACCEPT DIALOG instruction will be skipped.

You typically code an ACCEPT DIALOG in an ON ACTION accept block:

```
ON ACTION accept ACCEPT DIALOG
```

Note that any usage of ACCEPT DIALOG outside an ON ACTION accept block is not intended and its behavior is undocumented.

Input field validation is a process that does several successive validation tasks:

1. The current field value is checked, according to the program variable data type (for example, the user must input a valid date in a DATE field).
2. NOT NULL field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.
3. REQUIRED field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be modified by the user or by program with a DISPLAY TO / BY NAME or

`DIALOG.setFieldTouched()` call. If the field was not modified during the dialog, the `REQUIRED` constraint is not satisfied.

4. `INCLUDE` field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.

If a field does not satisfy one of these constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

After input field validation has succeeded, different types of control blocks will be executed, such as `AFTER FIELD`, `AFTER ROW`, `AFTER INPUT` and `AFTER DIALOG`.

In order to validate some parts of the dialog without leaving the block, use the `DIALOG.validate()` method.

CANCEL DELETE instruction

In a list controlled by an `INPUT ARRAY`, row deletion can be canceled by using the `CANCEL DELETE` instruction in the `BEFORE DELETE` block. Using this instruction in a different place will generate a compilation error.

When the `CANCEL DELETE` instruction is executed, the current `BEFORE DELETE` block is terminated without any other trigger execution (no `BEFORE ROW` or `BEFORE FIELD` is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```
BEFORE DELETE
   IF user_can_delete() == FALSE THEN
      ERROR "You are not allowed to delete rows"
      CANCEL DELETE
   END IF
```

The instructions that appear after `CANCEL DELETE` will be skipped.

If the row deletion condition is known before the delete action occurs, disable the delete action to prevent the user from performing a delete row action with the `DIALOG.setActionActive()` method:

```
CALL DIALOG.setActionActive("delete", FALSE)
```

It is also possible to prevent the user from deleting rows with the `DELETE ROW = FALSE` option in the `ATTRIBUTE` clause.

CANCEL INSERT instruction

In a list controlled by an `INPUT ARRAY`, row creation can be canceled by the program with the `CANCEL INSERT` instruction. This instruction can only be used in the `BEFORE INSERT` and `AFTER INSERT` control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after `CANCEL INSERT` will be skipped.

If the row creation condition is known before the insert/append action occurs, disable the insert and/or append actions to prevent the user from creating new rows, with `DIALOG.setActionActive()`:

```
CALL DIALOG.setActionActive("insert", FALSE)
CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list, when moving down after the last row. To prevent row creation completely, use the `INSERT ROW = FALSE` and `APPEND ROW =FALSE` options in the `ATTRIBUTE` clause of `INPUT ARRAY`, or combine with the `AUTO APPEND = FALSE` attribute.

CANCEL INSERT in BEFORE INSERT

A `CANCEL INSERT` executed inside a `BEFORE INSERT` block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The `BEFORE INSERT` block is terminated (further instructions are skipped).
3. The `BEFORE ROW` and `BEFORE FIELD` triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```
   BEFORE INSERT
      IF NOT user_can_insert THEN
         ERROR "You are not allowed to insert rows"
         CANCEL INSERT
      END IF
```

Executing `CANCEL INSERT` in `BEFORE INSERT` will also cancel a temporary row creation, except when there are no more rows in the list. In this case, `CANCEL INSERT` will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). You can prevent automatic temporary row creation with the `AUTO APPEND=FALSE` attribute. If `AUTO APPEND=FALSE` and a `CANCEL INSERT` is executed in `BEFORE INSERT` (user has invoked an append action), the temporary row will be deleted and list will remain empty if it was the last row.

CANCEL INSERT in AFTER INSERT

A `CANCEL INSERT` executed inside an `AFTER INSERT` block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The `AFTER INSERT` block is terminated (further instructions are skipped).
3. The `BEFORE ROW` and `BEFORE FIELD` triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```
   AFTER INSERT
      WHENEVER ERROR CONTINUE
      LET r = DIALOG.getCurrentRow("s_items")
      INSERT INTO items VALUES ( p_items[r].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<>0 THEN
         ERROR SQLERRMESSAGE
         CANCEL INSERT
      END IF
```

Examples

Example 1: DIALOG controlling two lists

Form file "lists.per":

```
LAYOUT
GRID
{
<t t1              >
[f11  |f12         ]
<                  >
<t t2              >
[f21  |f22         ]
```

```
<                  >
}
END
END
ATTRIBUTES
EDIT f11 = FORMONLY.column_11;
EDIT f12 = FORMONLY.column_12;
EDIT f21 = FORMONLY.column_21;
EDIT f22 = FORMONLY.column_22;
END
INSTRUCTIONS
SCREEN RECORD sr1(FORMONLY.column_11,FORMONLY.column_12);
SCREEN RECORD sr2(FORMONLY.column_21,FORMONLY.column_22);
END
```

Program file:

```
DEFINE
   arr1 DYNAMIC ARRAY OF RECORD
        column_11 INTEGER,
        column_12 VARCHAR(10)
   END RECORD,
   arr2 DYNAMIC ARRAY OF RECORD
        column_21 INTEGER,
        column_22 VARCHAR(10)
   END RECORD

MAIN
  DEFINE i INTEGER
  FOR i = 1 TO 20
    LET arr1[i].column_11 = i
    LET arr1[i].column_12 = "aaa "||i
    LET arr2[i].column_21 = i
    LET arr2[i].column_22 = "aaa "||i
  END FOR
  OPTIONS INPUT WRAP
  OPEN FORM f FROM "lists"
  DISPLAY FORM f
  DIALOG ATTRIBUTES(UNBUFFERED)
    DISPLAY ARRAY arr1 TO sr1.*
      BEFORE DISPLAY
        MESSAGE "We are in list one"
    END DISPLAY
    DISPLAY ARRAY arr2 TO sr2.*
      BEFORE DISPLAY
        MESSAGE "We are in list two"
    END DISPLAY
    ON ACTION close
      EXIT DIALOG
  END DIALOG
END MAIN
```

Example 2: DIALOG with CONSTRUCT and DISPLAY ARRAY

Form file "form1.per":

```
LAYOUT
GRID
{
<g g1                                >
 Name:     [f1                       ]
 State:    [f2                       ]
 City:     [f3                       ]
```

```
 Zip-code: [f4                          ]
[                      :cc    :sr    ]
<                                      >
<g g2                                  >
 <t t1                                >
  Id        Name
 [c1       |c2                        ]
 [c1       |c2                        ]
 [c1       |c2                        ]
 <                                     >
<                                       >
[                             :cw    ]
}
END
END

ATTRIBUTES
GROUP g1: TEXT = "Search criteria";
EDIT f1 = FORMONLY.cust_name TYPE VARCHAR;
EDIT f2 = FORMONLY.cust_state TYPE VARCHAR;
EDIT f3 = FORMONLY.cust_city TYPE VARCHAR;
EDIT f4 = FORMONLY.cust_zipcode TYPE VARCHAR;
BUTTON cc: clear, TEXT="Clear";
BUTTON sr: fetch, TEXT="Fetch";
GROUP g2: TEXT = "Customer list";
EDIT c1 = FORMONLY.c_id TYPE INTEGER;
EDIT c2 = FORMONLY.c_name TYPE VARCHAR;
BUTTON cw: close;
END

INSTRUCTIONS
SCREEN RECORD sr (FORMONLY.c_id, FORMONLY.c_name);
END
```

Program file:

```
MAIN
  DEFINE custarr DYNAMIC ARRAY OF RECORD
            c_id INTEGER,
            c_name VARCHAR(50)
          END RECORD
  DEFINE where_clause STRING

  OPTIONS INPUT WRAP

  OPEN FORM f1 FROM "form1"
  DISPLAY FORM f1

  DIALOG ATTRIBUTES(FIELD ORDER FORM, UNBUFFERED)

     CONSTRUCT BY NAME where_clause
          ON cust_name, cust_state, cust_city, cust_zipcode
        ON ACTION clear
          CLEAR cust_name, cust_state, cust_city, cust_zipcode
     END CONSTRUCT

     DISPLAY ARRAY custarr TO sr.*
       BEFORE ROW
         MESSAGE SFMT("Row: %1/%2", DIALOG.getCurrentRow("sr"),
                                    DIALOG.getArrayLength("sr"))
     END DISPLAY

     ON ACTION fetch
```

```
        MESSAGE "Where:", where_clause
        -- Execute SQL query here to fill custarr ...

     ON ACTION close
        EXIT DIALOG

  END DIALOG

END MAIN
```

Example 3: DIALOG with SUBDIALOG

Form file "comment.per":

```
LAYOUT
GRID
{
[cmt                              ]
}
END
END
ATTRIBUTES
TEXTEDIT cmd = FORMONLY.the_comment, STRETCH=BOTH;
END
```

The module "comment.4gl":

```
DEFINE the_comment VARCHAR(200)

DIALOG comment_input()
    INPUT BY NAME the_comment
        ON ACTION add_sep
           LET the_comment = the_comment || "\n---"
    END INPUT
END DIALOG
```

Form file "form1.per":

```
LAYOUT
VBOX
GRID
{
Id:   [f1        ]
Name: [f2                                        ]
}
END
FORM "comment"
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.cust_id TYPE INTEGER;
EDIT f2 = FORMONLY.cust_name TYPE VARCHAR;
END
```

Program file:

```
IMPORT FGL comment

MAIN
  DEFINE cust RECORD
            cust_id INTEGER,
            cust_name VARCHAR(50)
```

```
          END RECORD

  OPTIONS INPUT WRAP

  OPEN FORM f1 FROM "form1"
  DISPLAY FORM f1

  DIALOG ATTRIBUTES(FIELD ORDER FORM, UNBUFFERED)

     INPUT BY NAME cust.*
        ON ACTION check_exists
           MESSAGE "Check if customer record exists"
     END INPUT

     SUBDIALOG comment_input

     ON ACTION close
        EXIT DIALOG

  END DIALOG

END MAIN
```

Parallel dialogs (START DIALOG)

The `START DIALOG` and `TERMINATE DIALOG` instructions provide multiple dialogs functionality executing concurrently in different application forms.

- Understanding parallel dialogs on page 1199
- Syntax of the declarative DIALOG block on page 1201
- Syntax of the START DIALOG instruction on page 1207
- Syntax of the TERMINATE DIALOG instruction on page 1207
- Parallel dialog programming steps on page 1208
- Using parallel dialogs on page 1209
  - Structure of a declarative DIALOG block on page 1209
  - Declarative DIALOG block configuration on page 1215
  - Default actions created by a DIALOG block on page 1162
  - DIALOG data blocks on page 1163
  - DIALOG control blocks on page 1164
  - DIALOG interaction blocks on page 1179
  - DIALOG control instructions on page 1189
- Examples on page 1247
  - Example 1: Two independent record lists on page 1247

Understanding parallel dialogs

Parallel dialogs refers to the use of different declarative `DIALOG` blocks, in conjunction with the `START DIALOG` and `TERMINATE DIALOG` instructions, and an event loop using the `fgl_eventLoop()` built-in function, in order to control several forms simultaneously.

> **Important:** This feature is only for mobile platforms.

Each dialog acts independently to control several elements of a window/form. During the execution of parallel dialogs, the user can switch to a window/form that is controlled by another running declarative `DIALOG` block. For more details about categories of dialogs, see Introducing dialogs on page 1250.

The parallel dialog feature was introduced to implement mobile applications, where several forms can be accessed simultaneously, for example to get "split views" on mobile devices:

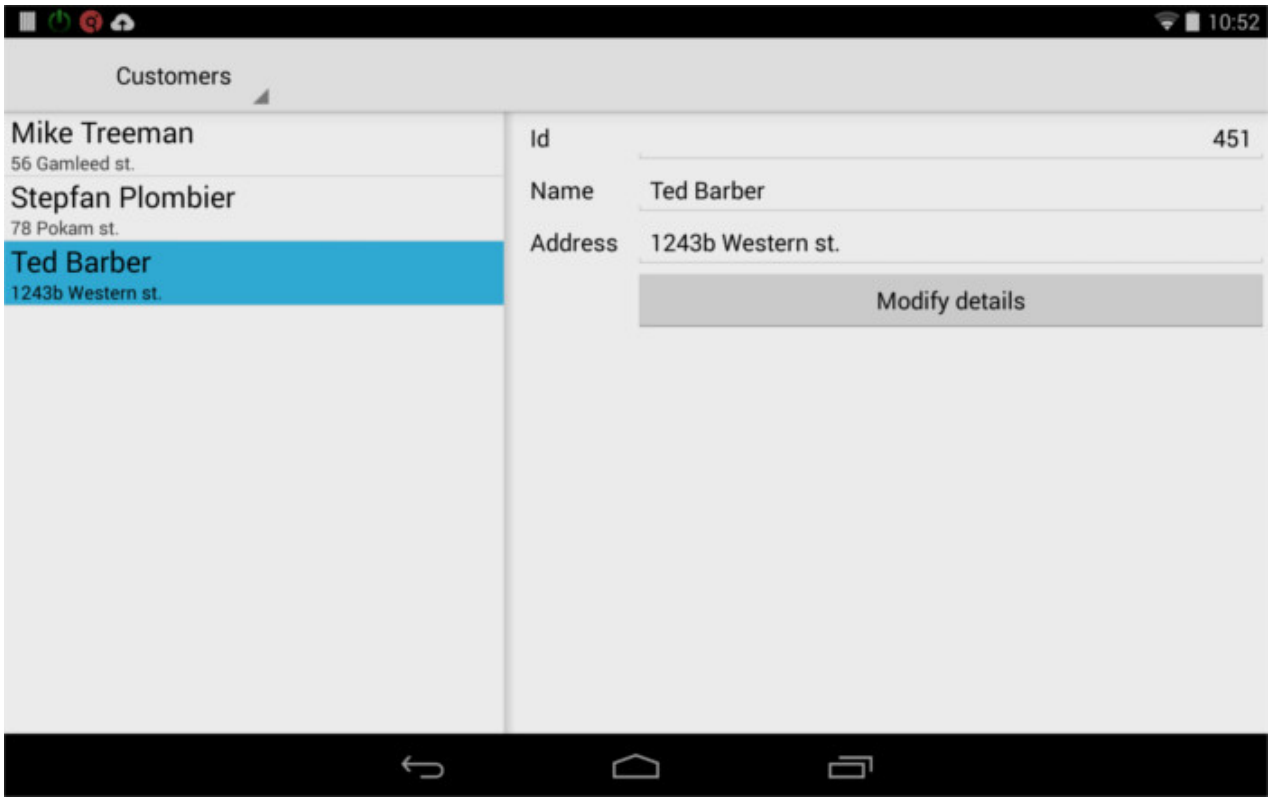


Figure 77: Form with Split View (Android™)

A *declarative dialog block* is a module element defined at the same level as a `FUNCTION` or `REPORT` routine:

```
-- Module orders.4gl
SCHEMA stock
DEFINE arr DYNAMIC ARRAY OF RECORD LIKE orders.*
DIALOG orders_dlg()
  DEFINE x INT
  DISPLAY ARRAY arr TO sr_orders.*
     ...
  END DISPLAY
END DIALOG
```

The name of a declarative dialog is mandatory. It can be referenced by a `SUBDIALOG` clause, by a `START DIALOG` and `TERMINATE DIALOG` instruction, and can identify sub-dialog actions with a prefix. Specifically, the name of the declarative dialog will be referenced in a `START DIALOG` and `TERMINATE DIALOG` instruction to implement parallel dialogs.

In terms of semantics, behavior and control block execution, a declarative dialog started with a `START DIALOG` instruction behaves like a procedural `DIALOG` block.

> **Important:** Parallel dialogs implictly use the `UNBUFFERED` mode. It is not possible to change this mode when using parallel dialogs.

When using the `DIALOG` keyword inside a declarative dialog block to use `ui.Dialog` class methods, it references the current instance of the dialog object.

In order to execute parallel dialogs, you must implement a main interaction event loop, by using the `fgl_eventLoop()` built-in function. The minimal event loop code to implement is:

```
WHILE fgl_eventLoop()
```

```
END WHILE
```

Once the declarative dialogs and the interaction even loop are defined, it is possible to create the windows with `OPEN WINDOW`, and initiate the dialogs with the `START DIALOG` instruction.

If needed, show a given dialog window with the `CURRENT WINDOW` instruction. Additionally, (especially when implementing split views), you may want to "restart" a detail dialog, for example when selecting a new row in the main record list. To restart the detail dialog, execute `TERMINATE DIALOG`, followed by `START DIALOG` for the detail dialog. See split view programming for more details.

To finish a given dialog, execute the `TERMINATE DIALOG` instruction and close the dedicated window with `CLOSE WINDOW window-name`.

From a set of running parallel dialogs, it is possible to switch to a modal dialog by creating a dedicated window, and executing a procedural dialog instruction. When the procedural dialog is terminated, close the dedicated window, and the control will go back to the parallel dialog set.

Syntax of the declarative DIALOG block
The declarative `DIALOG` block defines an interactive instruction that can be used by a parent `DIALOG`, or as parallel dialog.

Syntax

```
[ PRIVATE | PUBLIC ] DIALOG dialog-name ()
   [ define-block ]
   { menu-block
   | record-input-block
   | construct-block
   | display-array-block
   | input-array-block
   }
END DIALOG
```

1. *dialog-name* defines the identifier for the declarative `DIALOG` block.

where *define-block* is a local variable declaration block.

where *menu-block* is:

```
MENU
   [ BEFORE MENU
       menu-statement
     [...]
   ]
   menu-option
   [...]
END MENU
```

where *menu-option* is:

```
{ COMMAND option-name
      [option-comment] [ HELP help-number ]
    menu-statement
    [...]
| COMMAND KEY ( key-name ) option-name
      [option-comment] [ HELP help-number ]
    menu-statement
    [...]
| COMMAND KEY ( key-name )
    menu-statement
    [...]
| ON ACTION action-name
```

```
              [ ATTRIBUTES ( action-attributes-menu ) ]
    menu-statement
    [...]
}
```

where *action-attributes-menu* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
| DISCLOSUREINDICATOR
    [,...] }
```

where *menu-statement* is:

```
{ statement
|  NEXT OPTION option
|  SHOW OPTION { ALL | option [,...] }
|  HIDE OPTION { ALL | option [,...] }
}
```

1. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
2. *option-name* is a string expression defining the label of the menu option and identifying the action that can be executed by the user.
3. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
4. *help-number* is an integer that allows you to associate a help message number with the menu *option*.
5. *action-name* identifies an action that can be executed by the user.
6. *idle-seconds* is an integer literal or variable that defines a number of seconds.
7. *action-name* identifies an action that can be executed by the user.
8. *action-attributes* are dialog-specific action attributes.

where *record-input-block* is:

```
INPUT { BY NAME { variable | record.* } [,...]
      | variable | record.* } [,...] FROM field-list
      }
   [ ATTRIBUTES ( input-control-attribute [,...] ) ]
   [ input-control-block
       [...]
   ]
END INPUT
```

where *input-control-attribute* is:

```
{ HELP = help-number
| NAME = "sub-dialog-name"
| WITHOUT DEFAULTS [ = boolean ]
}
```

where *input-control-block* is one of:

```
{ BEFORE INPUT
| BEFORE FIELD field-spec [,...]
| ON CHANGE field-spec [,...]
| AFTER FIELD field-spec [,...]
| AFTER INPUT
```

```
| ON ACTION action-name
             [ INFIELD field-spec ]
             [ ATTRIBUTES ( action-attributes-input ) ]
| ON KEY ( key-name [,...] )}
    dialog-statement
    [...]
```

where *action-attributes-input* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| VALIDATE = NO
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *construct-block* is:

```
CONSTRUCT { BY NAME variable ON column-list
          | variable ON column-list FROM field-list
          }
   [ ATTRIBUTES ( construct-control-attribute [,...] ) ]
   [ construct-control-block
       [...]
   ]
END CONSTRUCT
```

where *construct-control-attribute* is:

```
{ HELP = help-number
| NAME = "sub-dialog-name"
}
```

where *construct-control-block* is one of:

```
{ BEFORE CONSTRUCT
| BEFORE FIELD field-spec [,...]
| AFTER FIELD field-spec [,...]
| AFTER CONSTRUCT
| ON ACTION action-name
             [INFIELD field-spec]
             [ ATTRIBUTES ( action-attributes-construct ) ]
| ON KEY ( key-name [,...] )}
    dialog-statement
    [...]
```

where *action-attributes-construct* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
    [,...] }
```

where *display-array-block* is:

```
DISPLAY ARRAY array TO screen-array.*
```

```
   [ ATTRIBUTES ( display-array-control-attribute [,...] ) ]
   [ display-array-control-block
      [...]
   ]
END DISPLAY
```

where *display-array-control-attribute* is:

```
{ HELP = help-number
| COUNT = row-count
| KEEP CURRENT ROW = [ = boolean ]
| DETAILACTION = action-name
| DOUBLECLICK = action-name
| ACCESSORYTYPE = { DETAIBUTTON | DISCLOSUREINDICATOR | CHECKMARK }
}
```

where *display-array-control-block* is one of:

```
{ BEFORE DISPLAY
| BEFORE ROW
| AFTER ROW
| AFTER DISPLAY
| ON ACTION action-name
             [ ATTRIBUTES ( action-attributes-display-array ) ]
| ON KEY ( key-name [,...] )
| ON FILL BUFFER
| ON SELECTION CHANGE
| ON SORT
| ON APPEND [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON INSERT [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON UPDATE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON DELETE [ ATTRIBUTES ( action-attributes-listmod-triggers ) ]
| ON EXPAND ( row-index )
| ON COLLAPSE ( row-index )
| ON DRAG_START ( dnd-object )
| ON DRAG_FINISH ( dnd-object )
| ON DRAG_ENTER( dnd-object )
| ON DRAG_OVER ( dnd-object )
| ON DROP ( dnd-object ) }
    dialog-statement
    [...]
```

where *action-attributes-display-array* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
| ROWBOUND
    [,...] }
```

where *action-attributes-listmod-triggers* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| CONTEXTMENU = { YES | NO | AUTO }
```

```
    [,...] }
```

where *input-array-block* is:

```
INPUT ARRAY array FROM screen-array.*
   [ ATTRIBUTES ( input-array-control-attribute [,...] ) ]
   [ input-array-control-block
      [...]
   ]
END INPUT
```

where *input-array-control-attribute* is:

```
{ APPEND ROW [ = boolean ]
| AUTO APPEND [ = boolean ]
| COUNT = row-count
| DELETE ROW [ = boolean ]
| HELP = help-number
| INSERT ROW [ = boolean ]
| KEEP CURRENT ROW [ = boolean ]
| MAXCOUNT = max-row-count
| WITHOUT DEFAULTS [ = boolean ]
}
```

where *input-array-control-block* is one of:

```
{ BEFORE INPUT
| BEFORE ROW
| BEFORE FIELD [,...]
| ON CHANGE field-spec [,...]
| AFTER FIELD field-spec [,...]
| ON ROW CHANGE
| ON SORT
| AFTER ROW
| BEFORE DELETE
| AFTER DELETE
| BEFORE INSERT
| AFTER INSERT
| AFTER INPUT
| ON ACTION action-name
             [INFIELD field-spec]
             [ ATTRIBUTES ( action-attributes-input-array ) ]
| ON KEY ( key-name [,...] ) }
    dialog-statement
    [...]
```

where *action-attributes-input-array* is:

```
{ TEXT = string
| COMMENT = string
| IMAGE = string
| ACCELERATOR = string
| DEFAULTVIEW = { YES | NO | AUTO }
| VALIDATE = NO
| CONTEXTMENU = { YES | NO | AUTO }
| ROWBOUND
    [,...] }
```

where *dialog-statement* is one of:

```
{ statement
| ACCEPT DIALOG
```

```
| CONTINUE DIALOG
| EXIT DIALOG
| NEXT FIELD
   { CURRENT
   | NEXT
   | PREVIOUS
   | field-spec
   }
}
```

where *field-list* defines a list of fields with one or more of:

```
{ field-name
| table-name.*
| table-name.field-name
| screen-array[line].*
| screen-array[line].field-name
| screen-record.*
| screen-record.field-name
} [,...]
```

where *field-spec* identifies a unique field with one of:

```
{ field-name
| table-name.field-name
| screen-array.field-name
| screen-record.field-name
}
```

where *column-list* defines a list of database columns as:

```
{ column-name
| table-name.*
| table-name.column-name
} [,...]
```

1. *variable-definition* is a variable declaration with data type as in a regular `DEFINE` statement.
2. *array* is the array of records used by the `DIALOG` statement.
3. *help-number* is an integer that allows you to associate a help message number with the command.
4. *field-name* is the identifier of a field of the current form.
5. *option-name* is a string expression defining the label of the action and identifying the action that can be executed by the user.
6. *option-comment* is a string expression containing a description for the menu option, displayed when *option-name* is the current.
7. *column-name* is the identifier of a database column of the current form.
8. *table-name* is the identifier of a database table of the current form.
9. *variable* is a simple program variable (not a record).
10. *record* is a program record (structured variable).
11. *screen-array* is the screen array that will be used in the current form.
12. *line* is a screen array line in the form.
13. *screen-record* is the identifier of a screen record of the current form.
14. *action-name* identifies an action that can be executed by the user.
15. *seconds* is an integer literal or variable that defines a number of seconds.
16. *key-name* is a hot-key identifier (like `F11` or `Control-z`).
17. *row-index* identifies the program variable which holds the row index corresponding to the tree node that has been expanded or collapsed.
18. *dnd-object* references a `ui.DragDrop` variable defined in the scope of the dialog.

19.*statement* is any instruction supported by the language.
20.*action-attributes* are dialog-specific action attributes for the action.

Syntax of the START DIALOG instruction
Starts the instance of a declarative dialog.

Syntax

```
START DIALOG dialog-name
```

1. *dialog-name* is the identifier of a declarative `DIALOG` block.

Usage

The `START DIALOG` instruction starts the declarative dialog block identified by the name passed.

The current window/form will be used to attach form fields and action views to the variables and action handlers implemented in the referenced declarative dialog.

The `START DIALOG` does in fact register the specified dialog to be activated when the parallel dialog event loop executes.

The started dialog can be terminated with `TERMINATE DIALOG`.

Example

This example shows a `START DIALOG` instruction in a function that initializes a parallel dialog in a split view context:

```
FUNCTION params()
  IF ui.Window.forName("w_params") IS NULL THEN
     OPEN WINDOW w_params WITH FORM "parameters"
 ATTRIBUTES(TYPE=LEFT)
     LET params.user_name="Tom"
     LET params.auto_sync="Y"
     DISPLAY BY NAME params.*
     START DIALOG d_params_menu
  END IF
  CURRENT WINDOW IS w_params
END FUNCTION
```

Syntax of the TERMINATE DIALOG instruction
Terminates the instance of a declarative dialog.

Syntax

```
TERMINATE DIALOG dialog-name
```

1. *dialog-name* is the identifier of a declarative `DIALOG` block.

Usage

The `TERMINATE DIALOG` instruction stops a declarative dialog identified by the name passed.

If the intent is to finish the parallel dialog, the corresponding window/form bound to the dialog should be closed after `TERMINATE DIALOG`.

However, `TERMINATE DIALOG` can also be used in conjunction with `START DIALOG`, to achieve a "restart" of the parallel dialog.

Note: `TERMINATE DIALOG` will not raise an error, if the dialog was not yet started with `START DIALOG`. This is required to implement the "restart" pattern.

The next code example shows a typical restart pattern on a detail parallel dialog, when a new row is selected in the master list:

```
DIALOG d_list_view()
  DISPLAY ARRAY arr TO sr.*
                ATTRIBUTES(ACCESSORYTYPE=DISCLOSUREINDICATOR)
      BEFORE ROW -- in BEFORE ROW, we restart the details view
        CURRENT WINDOW IS w_right
        TERMINATE DIALOG d_detail_view
        LET curr_pa = arr_curr()
        DISPLAY BY NAME arr[curr_pa].*
        DISPLAY SFMT("tapped row %1",arr_curr()) TO info
        START DIALOG d_detail_view
        CURRENT WINDOW IS w_left
   ...
```

Parallel dialog programming steps

This procedure describes how to implement parallel dialogs with a declarative `DIALOG` block.

1. Create a form specification file containing screen record(s) and/or screen array(s). The screen records and screen arrays identify the presentation elements to be used by the runtime system to display the data models (the content of program variables bound to the `DIALOG` blocks).
2. Create a dedicated .4gl module to implement the declarative `DIALOG` block.
3. With the `DEFINE` instruction, declare program variables (records and arrays) that will be used as data models. These will typically be defined as `PRIVATE` module variables. For record lists (`DISPLAY ARRAY` or `INPUT ARRAY`), the members of the program array must correspond to the elements of the screen array, by number and data types. To handle record lists, use dynamic arrays instead of static arrays.
4. Define the declarative `DIALOG` block in the module, to handle interaction. Define a sub-dialog with program variables to be used as data models. The sub-dialog will define how variables will be used (display or input).
   a) Inside the sub-dialog instruction, define the behavior with control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE FIELD`, and interaction blocks such as `ON ACTION`.
5. Define a `FUNCTION` to create the dialog instance.
   a) Add a test to check if the window and form combination dedicated to the dialog is already created, using `ui.Window.forName()`. If the window does not yet exist, create it by using the `OPEN WINDOW` *`window-name`* `WITH FORM` instruction. If the window exists, make it current with the `CURRENT WINDOW IS` *`window-name`* instruction.
   b) Fill the module variables (the data model) with data. For lists, you typically use a result set cursor.
   c) Start the dialog with the `START DIALOG` *`dialog-name`* instruction.
6. Define a `FUNCTION` to terminate the dialog instance.
   a) In the function, finish the dialog with `TERMINATE DIALOG` *`dialog-name`*.
   b) Close the window dedicated to the dialog with `CLOSE WINDOW` *`window-name`*.
   c) If needed, free the data model (clear large program arrays) and database cursors, to save memory.
7. If needed, add an `ON ACTION close` action handler to the declarative dialog, that calls the terminate function. This allows the end user to close the front-end window and stop the dialog.
8. In another module, implement the `WHILE` loop using the `fgl_eventLoop()` built-in function to handle interaction events for parallel dialogs. This module uses the start and terminate functions to control the individual dialog modules.

The simplest form of the user interaction event loop is:

```
WHILE fgl_eventLoop()
END WHILE
```

Using parallel dialogs

To use parallel dialogs, you must understand how they work and how to structure the code.

Structure of a declarative DIALOG block

A declarative `DIALOG` instruction is made of a single sub-dialog block, with an optional `DEFINE` clause to declare local variables.

> **Important:** Unlike procedural `DIALOG` blocks, declarative `DIALOG` blocks can only define one sub-dialog block.

The dialog instruction in the declarative DIALOG block binds program variables to form fields and define the type of interaction that will take place for the data model (simple input, list input or query). The dialog implement individual control blocks which let you control the behavior of the interactive instruction. The dialog can also hold action handlers.

The declarative `DIALOG` block can define the following dialog types:

- A list of choices controlled by a `MENU` sub-dialog block.
- Simple record input with the `INPUT` sub-dialog block.
- Query by example input with the `CONSTRUCT` sub-dialog block.
- Read-only record list navigation with the `DISPLAY ARRAY` sub-dialog block.
- Editable record list handling with the `INPUT ARRAY` sub-dialog block.

The DEFINE clause

The `DEFINE` clause can be used to define program variables with a scope that is local to the declarative dialog block.

This clause must be placed before any other sub-dialog block:

```
DIALOG ()
    DEFINE checked BOOLEAN,
           tmp STRING

    INPUT BY NAME ...
        ...
    END INPUT

END DIALOG
```

The `DEFINE` clause is only allowed in declarative dialog blocks. Variables used locally in a procedural dialog block should be defined in the scope of the function containing the procedural dialog block.

The MENU sub-dialog

The `MENU` sub-dialog implements a list of choices for the user by using action handlers.

MENU implements a list of action handlers

The following code example shows a `MENU` sub-dialog implementing a couple of action handlers with an `ON ACTION` clause or with a `COMMAND` clause (action views of `COMMAND` can get the focus):

```
DIALOG ()
   MENU
      ON ACTION customer_view
         ...
      ON ACTION order_view
         ...
   END MENU
```

```
END DIALOG
```

Control blocks in MENU

Simple record input declared with the INPUT sub-dialog can raise the following triggers:

- BEFORE MENU

In the singular MENU instruction, BEFORE MENU and AFTER MENU blocks are typically used as initialization and finalization blocks. In an MENU sub-dialog of a DIALOG block, BEFORE MENU and AFTER MENU blocks will be executed each time the focus goes to (BEFORE) or leaves (AFTER) the action views (buttons) controlled by this sub-dialog.

The INPUT sub-dialog
The INPUT sub-dialog implement single record input in fields of the current form.

Program variable to form field binding

Each record member variable is bound to the corresponding field of a screen record, in order to manipulate the values that the user enters in the form fields.

The INPUT clause can be used in two forms:

1. INPUT BY NAME *variable-list*
2. INPUT *variable-list* FROM *field-list*

The BY NAME clause implicitly binds the fields to the variables that have the same identifiers as the field names. The variables must be declared with the same names as the fields from which they accept input. The runtime system ignores any record name prefix when making the match. The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, the runtime system generates an exceptions, and sets the STATUS variable to a negative value.

```
DEFINE p_cust RECORD
       cust_num INTEGER,
       cust_name VARCHAR(50),
       cust_address VARCHAR(100)
   END RECORD
   ...
DIALOG
   INPUT BY NAME p_cust.*
      BEFORE FIELD cust_name
       ...
   END INPUT
   ...
END DIALOG
```

The FROM clause explicitly binds the fields in the screen record to a list of program variables by position. The number of variables or record members must equal the number of fields listed in the FROM clause. Each variable must be of the same (or a compatible) data types as the corresponding screen field. When the user enters data, the runtime system checks the entered value against the data type of the variable, not the data type of the screen field.

```
DEFINE c_name VARCHAR(50)
       c_addr VARCHAR(100)
   ...
DIALOG
  INPUT c_name,
        c_addr
   FROM FORMONLY.field01,
        FORMONLY.field02
    BEFORE FIELD cust_name
       ...
```

```
  END INPUT
   ...
END DIALOG
```

Identifying an INPUT sub-dialog

The name of an `INPUT` sub-dialog can be used to qualify sub-dialog actions with a prefix.

In order to identify the `INPUT` sub-dialog with a specific name, you can use the `ATTRIBUTES` clause to set the `NAME` attribute:

```
INPUT BY NAME p_cust.*
      ATTRIBUTES (NAME = "cust")
  ...
```

Control blocks in INPUT

Simple record input declared with the `INPUT` sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE FIELD
- ON CHANGE
- AFTER FIELD
- AFTER INPUT

In the singular `INPUT` instruction, `BEFORE INPUT` and `AFTER INPUT` blocks are typically used as initialization and finalization blocks. In an `INPUT` sub-dialog of a `DIALOG` block, `BEFORE INPUT` and `AFTER INPUT` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The CONSTRUCT sub-dialog
The `CONSTRUCT` sub-dialog provides database query by example feature, converting search criteria entered by the user into an SQL `WHERE` condition that can be use to execute a `SELECT` statement.

Defining query by example fields

The `CONSTRUCT` sub-dialog requires a character string variable to hold the `WHERE` clause, and a list of screen fields where the user can enter search criteria.

```
DEFINE sql_condition STRING
  ...
DIALOG
  CONSTRUCT BY NAME sql_condition
      ON customer.cust_name, customer.cust_address
    BEFORE FIELD cust_name
    ...
  END CONSTRUCT
  ...
END DIALOG
```

Make sure the character string variable is large enough to store all possible SQL conditions. It is better to use a `STRING` data type to avoid any size problems.

`CONSTRUCT` uses the field data types defined in the current form file to produce the SQL conditions. This is different from other interactive instructions, where the data types of the program variables define the way to handle input/display. It is *strongly* recommended (but not mandatory) that the form field data types correspond to the data types of the program variables used for input. This is implicit if both form fields and program variables are based on the database schema file.

The `CONSTRUCT` clause can be used in two forms:

1. `CONSTRUCT BY NAME` *`string-variable`* `ON` *`column-list`*
2. `CONSTRUCT` *`string-variable`* `ON` *`column-list`* `FROM` *`field-list`*

The `BY NAME` clause implicitly binds the form fields to the columns, where the form field identifiers match the column names specified in the column-list after the `ON` keyword. You can specify the individual column names (separated by commas) or use the `tablename.*` shortcut to include all columns defined for a table in the database schema file.

The `FROM` clause explicitly binds the form fields listed after the `FROM` keyword with the column definitions listed after the `ON` keyword.

In both cases, the name of the columns in *column-list* will be used to produce the SQL condition in *string-variable*.

Identifying a CONSTRUCT sub-dialog

The name of a `CONSTRUCT` sub-dialog can be used to qualify sub-dialog actions with a prefix. In order to identify the `CONSTRUCT` sub-dialog with a specific name, use the `ATTRIBUTES` clause to set the `NAME` attribute:

```
CONSTRUCT BY NAME sql_condition ON customer.*
  ATTRIBUTES (NAME = "q_cust")
  ...
```

Control blocks in CONSTRUCT

A Query By Example declared with the `CONSTRUCT` clause can raise the following triggers:

- BEFORE CONSTRUCT
- BEFORE FIELD
- AFTER FIELD
- AFTER CONSTRUCT

In the singular `CONSTRUCT` instruction, `BEFORE CONSTRUCT` and `AFTER CONSTRUCT` blocks are typically used as initialization and finalization blocks. In `DIALOG` block, `BEFORE CONSTRUCT` and `AFTER CONSTRUCT` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The DISPLAY ARRAY sub-dialog
The `DISPLAY ARRAY` sub-dialog is the controller to implement the navigation in a list of records, with option data modification actions.

Program array to screen array binding

The `DISPLAY ARRAY` sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the `TO` keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the `DIALOG` instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```
DEFINE p_items DYNAMIC ARRAY OF RECORD
       item_num INTEGER,
       item_name VARCHAR(50),
       item_price DECIMAL(6,2)
   END RECORD
   ...
```

```
DIALOG
   DISPLAY ARRAY p_items TO sa.*
      BEFORE ROW
      ...
   END DISPLAY
   ...
END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
   ...
DIALOG
   DISPLAY ARRAY p_names TO sa.*
      BEFORE DELETE
      ...
   END DISPLAY
   ...
END DIALOG
```

Identifying a DISPLAY ARRAY sub-dialog

The name of the screen array specified with the `TO` clause identifies the list. The dialog class method such as takes the name of the screen array as the parameter, identifying the list. For example, you would use `DIALOG.getCurrentRow("screen-array")` to query for the current row in the list identified by 'screen-array'. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in DISPLAY ARRAY

Read-only record lists declared with the `DISPLAY ARRAY` sub-dialog can raise the following triggers:

- BEFORE DISPLAY
- BEFORE ROW
- AFTER ROW
- AFTER DISPLAY

In the singular `DISPLAY ARRAY`instruction, `BEFORE DISPLAY` and `AFTER DISPLAY` blocks are typically used as initialization and finalization blocks. In a `DISPLAY ARRAY` sub-dialog of a `DIALOG` block, `BEFORE DISPLAY` and `AFTER DISPLAY` blocks will be executed each time the focus goes to (`BEFORE`) or leaves (`AFTER`) the group of fields defined by this sub-dialog.

The INPUT ARRAY sub-dialog
The `INPUT ARRAY` sub-dialog is the controller to implement the navigation and edition in a list of records.

> **Important:** This feature is not supported on mobile platforms.

Program array to screen array binding

The `INPUT ARRAY` sub-dialog binds the members of the flat record (or the primitive member) of an array to the screen-array or screen-record fields specified with the `FROM` keyword. The number of variables in each record of the program array must be the same as the number of fields in each screen record (that is, in a single row of the screen array).

You typically bind a program array to a screen-array in order to display a page of records. However, the `DIALOG` instruction can also bind the program array to a simple flat screen-record. In this case, only one record will be visible at a time.

The next code example defines an array with a flat record and binds it to a screen array:

```
DEFINE p_items DYNAMIC ARRAY OF RECORD
```

```
        item_num INTEGER,
        item_name VARCHAR(50),
        item_price DECIMAL(6,2)
    END RECORD
    ...
DIALOG
    INPUT ARRAY p_items FROM sa.*
       BEFORE INSERT
       ...
    END INPUT
    ...
END DIALOG
```

If the screen array is defined with one field only, you can bind an array defined with a primitive type:

```
DEFINE p_names DYNAMIC ARRAY OF VARCHAR(50)
    ...
DIALOG
    INPUT ARRAY p_names FROM sa.*
       BEFORE DELETE
       ...
    END INPUT
    ...
END DIALOG
```

Identifying an INPUT ARRAY sub-dialog

The name of the screen array specified with the `FROM` clause will be used to identify the list. For example, the dialog class method such as `DIALOG.getCurrentRow("screen-array")` takes the name of the screen array as the parameter, to identify the list you want to query for the current row. The name of the screen-array is also used to qualify sub-dialog actions with a prefix.

Control blocks in INPUT ARRAY

Editable record lists declared with the `INPUT ARRAY` sub-dialog can raise the following triggers:

- BEFORE INPUT
- BEFORE ROW
- BEFORE FIELD
- ON CHANGE
- AFTER FIELD
- ON ROW CHANGE
- AFTER ROW
- BEFORE DELETE
- AFTER DELETE
- BEFORE INSERT
- AFTER INSERT
- AFTER INPUT

In the singular `INPUT ARRAY` instruction, `BEFORE INPUT` and `AFTER INPUT` blocks are typically used as initialization and finalization blocks. In the `INPUT ARRAY` sub-dialog of a `DIALOG` block, `BEFORE INPUT` and `AFTER INPUT` blocks will be executed each time the focus goes to (`BEFORE`)or leaves (`AFTER`) the group of fields defined by this sub-dialog.

Declarative DIALOG block configuration

Attributes defined in the ATTRIBUTES clause of dialogs can be used to configure a declarative DIALOG block and its sub-dialogs.

The ATTRIBUTES clause of dialogs overrides all default attributes and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields.

- INPUT ATTRIBUTES clause on page 1159
- DISPLAY ARRAY ATTRIBUTES clause on page 1159
- INPUT ARRAY ATTRIBUTES clause on page 1160
- CONSTRUCT ATTRIBUTES clause on page 1162

INPUT ATTRIBUTES clause
Attributes of the INPUT clause of a DIALOG block.

NAME option

The NAME attribute can be used to identify the INPUT sub-dialog, especially useful to qualify sub-dialog actions.

HELP option

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the INPUT sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

WITHOUT DEFAULTS option

By default, sub-dialogs use the default values defined in the form files. If you want to use the values stored in the program variables bound to the dialog, you must use the WITHOUT DEFAULTS attribute. For more details see WITHOUT DEFAULTS option.

DISPLAY ARRAY ATTRIBUTES clause
Attributes of the DISPLAY ARRAY clause of a DIALOG block.

HELP option

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the DISPLAY ARRAY sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

COUNT option

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array, unless page mode is used. In this case, the COUNT attribute must be used to define the total number of rows, because the dynamic array will only hold a page of the entire row set. If the value of COUNT is negative or zero, it defines an empty list.

See also Controlling the total number of rows on page 1350.

DOUBLECLICK option

The DOUBLICKCLICK option can be used to define the action that will be fired when the user chooses a row from the list. On front-end platforms using a mouse-device, this corresponds to a physical double-click on a row with the mouse. On mobile front-ends, this corresponds to a tap on the row with a finger.

Note that this attribute can also be defined for the TABLE/TREE containers in form files; DOUBLECLICK in DISPLAY ARRAY attributes has a higher precedence as DOUBLECLICK in the form file. For more details, see Defining the action for a row choice on page 1360.

ACCESSORTYPE option

Important: This feature is only for mobile platforms.

The ACCESSORYTYPE attribute can be used to define the decoration of rows, typically used on a iOS device. Values can be DETAILBUTTON, DISCLOSUREINDICATOR, CHECKMARK to respectively get a (i), > or checkmark icon. For more details, see Row configuration on iOS devices on page 1369.

DETAILACTION option

Important: This feature is only for mobile platforms.

The DETAILACTION attribute can be used to define the action that will be fired when the user selects the detail button of a row. The detail button is typically shown with a (i) icon on iOS devices. Note that the DOUBLECLICK attribute can be used to distinguish the action when the user selects the row instead of the detail button in the row. For more details, see Row configuration on iOS devices on page 1369.

INPUT ARRAY ATTRIBUTES clause
Attributes of the INPUT ARRAY clause of a DIALOG block.

INPUT ARRAY specific attributes can be defined in the ATTRIBUTE clause of the sub-dialog header:

HELP option

The HELP clause specifies the number of a help message to display if the user invokes the help the INPUT ARRAY dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action. The HELP clause overrides the HELP attribute.

WITHOUT DEFAULTS option

You typically use the INPUT ARRAY sub-dialog with the WITHOUT DEFAULTS attribute. If this attribute is not set when using an INPUT ARRAY sub-dialog, the list is empty even if the array holds data. For more details see WITHOUT DEFAULTS option.

COUNT option

The COUNT attribute defines the number of valid rows in the static array to be displayed as default rows. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, so the screen array remains empty. The COUNT option is ignored when using a dynamic array. If you specify the COUNT attribute, the WITHOUT DEFAULTS option is not required because it is implicit. If the COUNT attribute is greater than MAXCOUNT, the runtime system will take MAXCOUNT as the actual number of rows. If the value of COUNT is negative or zero, it defines an empty list.

MAXCOUNT option

The MAXCOUNT attribute defines the maximum number of rows that can be inserted in the program array. This attribute allows you to give an upper limit of the total number of rows the user can enter. It can be used with static or dynamic arrays.

When binding a static array, MAXCOUNT is used as upper limit if it is lower or equal to the actual declared static array size. If MAXCOUNT is greater than the array size, the size of the static array is used as the upper limit. If MAXCOUNT is lower than the COUNT attribute (or to the SET_COUNT() parameter when using a singular INPUT ARRAY), the actual number of rows in the array will be reduced to MAXCOUNT.

When binding a dynamic array, the user can enter an infinite number of rows unless the MAXCOUNT attribute is used. If MAXCOUNT is lower than the actual size of the dynamic array, the number of rows in the array will be reduced to MAXCOUNT.

If MAXCOUNT is negative or equal to zero, the user cannot insert rows.

APPEND ROW option

The APPEND ROW attribute can be set to FALSE to avoid the append default action, and deny the user to add rows at the end of the list. If APPEND ROW =FALSE, it is still possible to insert rows in the middle of the list. Use the INSERT ROW attribute to disallow the user from inserting rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

INSERT ROW option

The INSERT ROW attribute can be set to FALSE to avoid the insert default action, and deny the user to insert new rows in the middle of the list. However, even if INSERT ROW is FALSE, it is still possible to append rows at the end of the list. Use the APPEND ROW attribute to disallow the user from appending rows. Additionally, even with APPEND ROW=FALSE and INSERT ROW=FALSE, you can still get automatic temporary row creation if AUTO APPEND is not set to FALSE.

DELETE ROW option

The DELETE ROW attribute can be set to FALSE to avoid the delete default action, and deny the user to remove rows from the list.

AUTO APPEND option

By default, an INPUT ARRAY controller creates a temporary row when needed (for example, when the user deletes the last row of the list, an new row will be automatically created). You can prevent this default behavior by setting the AUTO APPEND attribute to FALSE. When this attribute is set to FALSE, the only way to create a new temporary row is to execute the append action.

If both the APPEND ROW and INSERT ROW attributes are set to FALSE, the dialog automatically behaves as if AUTO APPEND equals FALSE.

KEEP CURRENT ROW option

Depending on the list container used in the form, the current row may be highlighted during the execution of the dialog, and cleared when the instruction ends. You can change this default behavior by using the KEEP CURRENT ROW attribute, to force the runtime system to keep the current row highlighted.

CONSTRUCT ATTRIBUTES clause
Attributes of the CONSTRUCT clause of a DIALOG block.

HELP option

The HELP attribute defines the number of the help message to be displayed when invoked and focus is in the list controlled by the CONSTRUCT sub-dialog. The predefined 'help' action is automatically created by the runtime system. You can bind action views to the 'help' action.

The HELP clause overrides the HELP attribute.

NAME option

The NAME attribute can be used to identify the CONSTRUCT sub-dialog; this is especially useful to qualify sub-dialog actions.

Default actions created by a DIALOG block

Default actions ease the implementation of the controller by providing expected actions.

According to the sub-dialogs defined in a (declarative or procedural) `DIALOG` block, the runtime system creates a set of default actions. These actions are provided to ease the implementation of the controller. For example, when using an `INPUT ARRAY` sub-dialog, the dialog instruction will automatically create the insert, append and delete default actions.

Table 282: Default actions created for the DIALOG block on page 1218 lists the default actions created for the `DIALOG` interactive instruction, according to the sub-dialogs defined:

Table 282: Default actions created for the DIALOG block

Default action	Control Block execution order
`help`	Shows the help topic defined by the `HELP` clause. *Only created when a* `HELP` *clause or option is defined for the sub-dialog.*
`insert`	Inserts a new row before current row. *Only created if* `INPUT ARRAY` is *used; action creation can be avoided with* `INSERT ROW = FALSE` *attribute.*
`append`	Appends a new row at the end of the list. *Only created if* `INPUT ARRAY` is *used; action creation can be avoided with* `APPEND ROW = FALSE` *attribute.*
`delete`	Deletes the current row. *Only created if* `INPUT ARRAY` is *used;* action *creation can be avoided with* `DELETE ROW = FALSE` *attribute.*
`nextrow`	Moves to the next row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`prevrow`	Moves to the previous row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`firstrow`	Moves to the first row in a list displayed in one row of fields. *Only created if* `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`lastrow`	Moves to the last row in a list displayed in one row of fields.

Default action	Control Block execution order
	Only created if `DISPLAY ARRAY` *or* `INPUT ARRAY` *used with a screen record having only one row.*
`find`	Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value. *Only created if the context allows built-in find.*
`findnext`	Seeks to the next row matching the value entered during the fglfind dialog. *Only created if the context allows built-in find.*

The `insert`, `append` and `delete` default actions can be avoided with dialog control attributes:

```
INPUT ARRAY arr TO sr.* ATTRIBUTES( INSERT ROW=FALSE, APPEND
 ROW=FALSE, ... )
     ...
```

DIALOG data blocks
Dialog data blocks are dialog triggers invoked when the dialog controller needs data to feed the view with values.

Such blocks are typically used when record list data is provided dynamically, with the paged mode or when implementing dynamic tree-views.

- ON FILL BUFFER block on page 1082
- ON EXPAND block on page 1082
- ON COLLAPSE block on page 1082

ON FILL BUFFER block
The `ON FILL BUFFER` block is used to fill a page of rows into the dynamic array, according to an offset and a number of rows.

This data block is used in the `DISPLAY ARRAY` blocks.

The offset can be retrieved with the `FGL_DIALOG_GETBUFFERSTART()` built-in function and the number of rows to provide is defined by the `FGL_DIALOG_GETBUFFERLENGTH()` built-in function.

ON EXPAND block
The `ON EXPAND` block is executed when a tree view node is expanded (i.e. opened).

This data block is used to implement dynamic trees in a `DISPLAY ARRAY`, where nodes are added according to the nodes opened by the end user.

ON COLLAPSE block
The `ON COLLAPSE` block is executed when a tree view node is collapsed (i.e. closed).

This data block is used to implement dynamic trees in a `DISPLAY ARRAY`, where nodes are removed according to the nodes closed by the end user.

DIALOG control blocks
Dialog control blocks are predefined dialog triggers where you can implement specific code to control the interactive instruction.

The code could involve using `ui.Dialog` class methods or dialog specific instructions such as `NEXT FIELD` or `CONTINUE DIALOG`.

- Control block execution order in parallel dialogs on page 1220
- BEFORE FIELD block on page 1069

- AFTER FIELD block on page 1070
- ON CHANGE block on page 1069
- BEFORE INPUT block on page 1067
- AFTER INPUT block on page 1068
- BEFORE CONSTRUCT block on page 1135
- AFTER CONSTRUCT block on page 1135
- BEFORE DISPLAY block on page 1082
- AFTER DISPLAY block on page 1083
- BEFORE ROW block on page 1083
- ON ROW CHANGE block on page 1110
- AFTER ROW block on page 1084
- BEFORE INSERT block on page 1113
- AFTER INSERT block on page 1113
- BEFORE DELETE block on page 1114
- AFTER DELETE block on page 1114
- BEFORE MENU block on page 1054

Control block execution order in parallel dialogs

The order in which control blocks are executed in a declarative `DIALOG` used as parallel dialog is the same as when executing a singular dialog.

According to the type of dialog defined in the declarative `DIALOG`, see:

- INPUT control blocks execution order on page 1066
- DISPLAY ARRAY control blocks execution order on page 1082
- CONSTRUCT control blocks execution order on page 1134
- INPUT ARRAY control blocks execution order on page 1106

For control block execution order in the context of a procedural `DIALOG` block, see Control block execution order in multiple dialogs on page 1164.

BEFORE FIELD block

For fields controlled by an `INPUT`, `INPUT ARRAY` or by a `CONSTRUCT` instructions, the `BEFORE FIELD` block is executed every time the cursor enters into the specified field.

For editable lists driven by `INPUT ARRAY`, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The `BEFORE FIELD` block is also executed when performing a `NEXT FIELD` instruction.

The `BEFORE FIELD` keywords must be followed by a list of form field specification. The screen-record name can be omitted.

`BEFORE FIELD` is executed after `BEFORE INPUT`, `BEFORE CONSTRUCT`, `BEFORE ROW` and `BEFORE INSERT`.

Use this block to do some field value initialization, or to display a message to the user:

```
INPUT BY NAME p_cust.* ...
   BEFORE FIELD cust_status
     LET p_cust.cust_comment = NULL
     MESSAGE "Enter customer status"
```

When using the default `FIELD ORDER CONSTRAINT` mode, the dialog executes the `BEFORE FIELD` block of the field corresponding to the first variable of an `INPUT` or `INPUT ARRAY`, even if that field is not editable (`NOENTRY`, hidden or disabled). The block is executed when you enter the dialog and every time you create a new row in the case of `INPUT ARRAY`. This behavior is supported for backward compatibility.

The block is not executed when using the FIELD ORDER FORM, the mode recommended for DIALOG instructions.

With the FIELD ORDER FORM mode, for each dialog executing the first time with a specific form, the BEFORE FIELD block might be invoked for the first field of the initial tabbing list defined by the form, even if that field was hidden or moved around in a table. The dialog then behaves as if a NEXT FIELD first-visible-column would have been done in the BEFORE FIELD of that field.

When form-level validation occurs and a field contains an invalid value, the dialog gives the focus to the field, but no BEFORE FIELD trigger will be executed.

AFTER FIELD block

In dialog parts driven by a simple INPUT, INPUT ARRAY or by a CONSTRUCT sub-dialog, the AFTER FIELD block is executed every time the focus leaves the specified field. For editable lists driven by INPUT ARRAY, this block is executed when moving the focus from field to field in the same row, or when moving to another row in the same column.

The AFTER FIELD keywords must be followed by a list of form field specifications. The screen-record name can be omitted.

AFTER FIELD is executed before AFTER INSERT, ON ROW CHANGE, AFTER ROW, AFTER INPUT or AFTER CONSTRUCT.

When a NEXT FIELD instruction is executed in an AFTER FIELD block, the cursor moves to the specified field, which can be the current field. This can be used to prevent the user from moving to another field / row during data input. Note that the BEFORE FIELD block is also executed when NEXT FIELD is invoked.

The AFTER FIELD block of the current field is not executed when performing a NEXT FIELD; only BEFORE INPUT, BEFORE CONSTRUCT, BEFORE ROW, and BEFORE FIELD of the target item might be executed, based on the sub-dialog type.

When ACCEPT DIALOG, ACCEPT INPUT or ACCEPT CONTRUCT is performed, the AFTER FIELD trigger of the current field is executed.

Use the AFTER FIELD block to implement field validation rules:

```
INPUT BY NAME p_item.* ...
   AFTER FIELD item_quantity
     IF p_item.item_quantity <= 0 THEN
        ERROR "Item quantity cannot be negative or zero"
        LET p_item.item_quantity = 0
        NEXT FIELD item_quantity
     END IF
```

ON CHANGE block

The ON CHANGE block can be used to detect that a field changed by user input. The ON CHANGE block is executed if the value has changed since the field got the focus and if the modification flag is set. The ON CHANGE block can only be used for fields controlled by an INPUT or INPUT ARRAY dialog, it is not available in CONSTRUCT.

For editable fields defined as EDIT, TEXTEDIT or BUTTONEDIT, the ON CHANGE block is executed when leaving a field, if the value of the specified field has changed since the field got the focus and if the modification flag is set for the field. You leave the field when you validate the dialog, when you move to another field, or when you move to another row in an INPUT ARRAY. However, if the text edit field is defined with the COMPLETER attribute to enable autocompletion, the ON CHANGE trigger will be fired after a short period of time, when the user has typed characters in.

For editable fields defined as CHECKBOX, COMBOBOX, DATEEDIT, DATETIMEEDIT, TIMEEDIT, RADIOGROUP, SPINEDIT, SLIDER or URL-based WEBCOMPONENT (when the COMPONENTTYPE attribute is not used), the ON CHANGE block is invoked immediately when the user changes the value with the widget edition feature. For example, when toggling the state of a CHECKBOX, when selecting an item in

a COMBOBOX list, or when choosing a date in the calendar of a DATEEDIT. Note that for such item types, when ON CHANGE is fired, the modification flag is always set.

```
   ON CHANGE order_checked -- Defined as CHECKBOX
      CALL setup_dialog(DIALOG)
```

If both an ON CHANGE block and AFTER FIELD block are defined for a field, the ON CHANGE block is executed before the AFTER FIELD block.

When changing the value of the current field by program in an ON ACTION block, the ON CHANGE block will be executed when leaving the field if the value is different from the reference value and if the modification flag is set (after previous user input or when the touched flag has been changed by program).

When using the NEXT FIELD instruction, the comparison value is reassigned as if the user had leaved and reentered the field. Therefore, when using NEXT FIELD in ON CHANGE block or in an ON ACTION block, the ON CHANGE block will only be invoked again if the value is different from the reference value. This denies to do field validation in ON CHANGE blocks: you must do validations in AFTER FIELD blocks and/or AFTER INPUT blocks.

BEFORE INPUT block

BEFORE INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular INPUT, INPUT ARRAY instruction, or when used as parallel dialog, the BEFORE INPUT is only executed once when the dialog is started.

The BEFORE INPUT block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
INPUT BY NAME cust_rec.* ...
   BEFORE INPUT
       MESSAGE "Input customer information"
       CALL DIALOG.setActionActive("check_info", is_super_user() )
       CALL DIALOG.setFieldActive("cust_comment", is_super_user() )
   ...
```

The fields are initialized with the defaults values before the BEFORE INPUT block is executed. When the INPUT instruction uses the WITHOUT DEFAULTS option, the default values are taken from the program variables bound to the fields, otherwise (with defaults), the DEFAULT attributes of the form fields are used.

Use the NEXT FIELD control instruction in the BEFORE INPUT block, to jump to a specific field when the dialog starts.

BEFORE INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an INPUT or INPUT ARRAY sub-dialog of a procedural DIALOG instruction, the BEFORE INPUT block is executed when the focus goes to a group of fields driven by the sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

When the focus is in a list driven by an INPUT ARRAY sub-dialog, moving to a different row will not invoke the BEFORE INPUT block.

BEFORE INPUT is executed after the BEFORE DIALOG block andbefore the BEFORE ROW, BEFORE FIELD blocks.

In this example, the BEFORE INPUT block is used to set up a specific action and display a message:

```
INPUT BY NAME p_order.*
   BEFORE INPUT
       CALL DIALOG.setActionActive("validate_order", TRUE)
```

AFTER INPUT block

AFTER INPUT block in singular and parallel INPUT, INPUT ARRAY dialogs

In a singular `INPUT`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `AFTER INPUT` is only executed once when dialog ends.

The `AFTER INPUT` block is executed after the user has validated or canceled the `INPUT` or `INPUT ARRAY` dialog with the accept or cancel default actions, or when the `ACCEPT INPUT` instruction is executed.

The `AFTER INPUT` block is not executed when the `EXIT INPUT` instruction is performed.

In singular and parallel dialogs, this block is typically used to implement global dialog validation rules depending from several fields. If the values entered by the user do not satisfy the constraints, use the `NEXT FIELD` instruction to force the dialog to continue. The `CONTINUE INPUT` instruction can be used instead of `NEXT FIELD`, when no particular field has to be select.

Before checking the validation rules, make sure that the `INT_FLAG` variable is `FALSE`: in case if the user cancels the dialog, the validation rules must be skipped.

```
INPUT BY NAME cust_rec.*
   WITHOUT DEFAULTS ATTRIBUTES ( UNBUFFERED )
  ...

  AFTER INPUT
    IF NOT INT_FLAG THEN
       IF cust_rec.cust_address IS NOT NULL
          AND cust_rec.cust_zipcode IS NULL THEN
          ERROR "Address is incomplete, enter a zipcode."
          NEXT FIELD zipcode
       END IF
    END IF
END INPUT
```

To limit the validation to fields that have been modified by the end user, you can call the `FIELD_TOUCHED()` function or the `DIALOG.getFieldTouched()` method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER INPUT block in INPUT and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER INPUT` block is executed when the focus is lost by a group of fields driven by an `INPUT` or `INPUT ARRAY` sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus. When the focus is in a list driven by an `INPUT ARRAY` sub-dialog, moving to a different row will not invoke the `AFTER INPUT` block.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

`AFTER INPUT` is executed after the `AFTER FIELD`, `AFTER ROW` blocks and before the `AFTER DIALOG` block.

Executing a `NEXT FIELD` in the `AFTER INPUT` control block will keep the focus in the group of fields. Within an `INPUT ARRAY` sub-dialog, `NEXT FIELD` will keep the focus in the list and stay in the current row. You typically use this behavior to control user input.

In this example, the `AFTER INPUT` block is used to validate data and disable an action that can only be used in the current group:

```
INPUT BY NAME p_order.*
   AFTER INPUT
      IF NOT check_order_data(DIALOG) THEN
```

```
        NEXT FIELD CURRENT
      END IF
      CALL DIALOG.setFieldActive("validate_order", FALSE)
```

BEFORE CONSTRUCT block

BEFORE CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular `CONSTRUCT` instruction, or when used as parallel dialog, the `BEFORE CONSTRUCT` is only executed once when dialog is started.

The `BEFORE CONSTRUCT` block is executed once at dialog startup, before the runtime system gives control to the user for criteria input. This block can be used to display messages to the user, initialize form fields with default search criteria values, and setup the dialog instance by deactivating unused fields or actions the user is not allowed to execute.

```
CONSTRUCT BY NAME where_part ON ...
   BEFORE CONSTRUCT
       MESSAGE "Enter customer search filter"
       CALL DIALOG.setActionActive("clean", FALSE )
   ...
```

The fields are cleared before the `BEFORE CONSTRUCT` block is executed.

You can use the `NEXT FIELD` control instruction in the `BEFORE CONSTRUCT` block, to jump to a specific field when the dialog starts.

BEFORE CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a `CONSTRUCT` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE CONSTRUCT` block is executed when the focus goes to a group of fields driven by a `CONSTRUCT` sub-dialog. This trigger is only invoked if a field of the sub-dialog gets the focus, and none of the other fields had the focus.

`BEFORE CONSTRUCT` is executed after the `BEFORE DIALOG` block and before the `BEFORE FIELD` blocks.

In this example, the `BEFORE CONSTRUCT` block is used to display a message:

```
CONSTRUCT BY NAME sql ON customer.*
  BEFORE CONSTRUCT
    MESSAGE "Enter customer search filter"
```

AFTER CONSTRUCT block

AFTER CONSTRUCT block in singular and parallel CONSTRUCT dialogs

In a singular `CONSTRUCT` instruction, or when used as parallel dialog, the `AFTER CONSTRUCT` is only executed once when dialog is ended.

Use an `AFTER CONSTRUCT` block to execute instructions after the user has finished search criteria input.

`AFTER CONSTRUCT` is not executed if an `EXIT CONSTRUCT` is performed.

The code in `AFTER CONSTRUCT` can for example check if a criteria combination of different fields is required or denied, and force the end use to enter all

Before checking the content of the fields used in the `CONSTRUCT`, make sure that the `INT_FLAG` variable is `FALSE`: in case if the user cancels the dialog, the validation rules must be skipped.

Since no program variables are associated to the form fields, you must query the input buffers of the fields to get the values entered by the user.

```
CONSTRUCT BY NAME where_part ON ...
  ...
```

```
  AFTER CONSTRUCT
    IF NOT INT_FLAG THEN
       IF length(DIALOG.getFieldBuffer(cust_name))==0
       OR length(DIALOG.getFieldBuffer(cust_addr))==0 THEN
          ERROR "Enter a search criteria for customer name and address
 fields."
          NEXT FIELD CURRENT
       END IF
    END IF
END CONSTRUCT
```

To limit the validation to fields that have been modified by the end user, you can call the `FIELD_TOUCHED()` function or the `DIALOG.getFieldTouched()` method to check if a field has changed during the dialog execution. This will make your validation code faster if the user has only modified a couple of fields in a large form.

AFTER CONSTRUCT block in CONSTRUCT of procedural DIALOG

In a `CONSTRUCT` sub-dialog of a procedural `DIALOG` instruction, the `AFTER CONSTRUCT` block is executed when the focus is lost by a group of fields driven by a `CONSTRUCT` sub-dialog. This trigger is invoked if a field of the sub-dialog loses the focus, and a field of a different sub-dialog gets the focus.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

`AFTER CONSTRUCT` is executed after the `AFTER FIELD` and before the `AFTER DIALOG` block.

Executing a `NEXT FIELD` in the `AFTER CONSTRUCT` control block will keep the focus in the group of fields.

In this example, the `AFTER CONSTRUCT` block is used to build the SELECT statement:

```
CONSTRUCT BY NAME sql ON customer.*
   AFTER CONSTRUCT
       LET sql = "SELECT * FROM customers WHERE " || sql
```

BEFORE DISPLAY block

BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a singular `DISPLAY ARRAY` instruction, or when used as parallel dialog, the `BEFORE DISPLAY` is only executed once when the dialog is started.

The `BEFORE DISPLAY` block is executed once at dialog startup, before the runtime system gives control to the user. This block can be used to display messages to the user, initialize program variables and setup the dialog instance by deactivating actions the user is not allowed to execute.

```
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", is_super_user())
```

BEFORE DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a `DISPLAY ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE DISPLAY` block is executed when a `DISPLAY ARRAY` list gets the focus.

`BEFORE DISPLAY` is executed before the `BEFORE ROW` block.

In this example the `BEFORE DISPLAY` block enables an action and displays a message:

```
DISPLAY ARRAY p_items TO s_items.*
   BEFORE DISPLAY
```

```
        CALL DIALOG.setActionActive("print_list", TRUE)
        MESSAGE "You are now in the list of items"
```

AFTER DISPLAY block

AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a singular `DISPLAY ARRAY` instruction, or when used as parallel dialog, the `AFTER DISPLAY` is only executed once when dialog is ended.

You typically implement dialog finalization in this block.

```
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       DISPLAY "Current row is: ", arr_curr()
```

AFTER DISPLAY block in singular and parallel DISPLAY ARRAY dialogs

In a `DISPLAY ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER DISPLAY` block is executed when a `DISPLAY ARRAY` list loses the focus and goes to another sub-dialog.

If the focus leaves the current group and goes to an action view, this trigger is not executed, because the focus did not go to another sub-dialog yet.

`AFTER DISPLAY` is executed after the `AFTER ROW` block.

In this example, the `AFTER DISPLAY` block disables an action that is specific to the current list:

```
DISPLAY ARRAY p_items TO s_items.*
   AFTER DISPLAY
       CALL DIALOG.setActionActive("clear_item_list", FALSE)
```

BEFORE ROW block

BEFORE ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular `DISPLAY ARRAY`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `BEFORE ROW` block is executed each time the user moves to another row. This trigger can also be executed in other situations, such as when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the `BEFORE ROW` block, because it indicates that the user selected a new row or entered in the list.

When the dialog starts, `BEFORE ROW` will be executed for the current row, but only if there are data rows in the array.

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the current row.

In this example, the `BEFORE ROW` block gets the new row number and displays it in a message:

```
DISPLAY ARRAY ...
   ...
   BEFORE ROW
     MESSAGE "We are on row # ", arr_curr()
   ...
```

BEFORE ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `BEFORE ROW` block is executed when a `DISPLAY ARRAY` or `INPUT ARRAY` list gets the focus, or when the user moves to

another row inside a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user tries to insert a row but the maximum number of rows in the list is reached.

You typically do some dialog setup / message display in the `BEFORE ROW` block, because it indicates that the user selected a new row. Do not use this trigger to detect focus changes; Use the `BEFORE DISPLAY` or `BEFORE INPUT` blocks instead.

In `DISPLAY ARRAY`, `BEFORE ROW` is executed after the `BEFORE DISPLAY` block. In `INPUT ARRAY`, `BEFORE ROW` is executed before the `BEFORE INSERT` and `BEFORE FIELD` blocks and after the `BEFORE INPUT` blocks.

When the procedural dialog starts, `BEFORE ROW` will only be executed if the list has received the focus and there is a current row (the array is not empty). If you have other elements in the form which can get the focus before the list, `BEFORE ROW` will not be triggered when the dialog starts. You must pay attention to this, because this behavior is different to the behavior of singular `DISPLAY ARRAY` or `INPUT ARRAY`. In singular dialogs, the `BEFORE ROW` block is always executed when the dialog starts (and there are rows in the array).

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the current row.

In this example the `BEFORE ROW` block displays a message with the current row number:

```
DISPLAY ARRAY p_items TO s_items.*
  BEFORE ROW
    MESSAGE "We are in items, on row #", DIALOG.getCurrentRow("s_items")
```

ON ROW CHANGE block

The `ON ROW CHANGE` block is executed in a list controlled by an `INPUT ARRAY`, when leaving the current row and when the row has been modified since it got the focus. This is typically used to detect row modification.

The code in `ON ROW CHANGE` will not be executed when leaving new rows created by the user with the default append or insert action. To detect row creation, you must use the `BEFORE INSERT` or `AFTER INSERT` control blocks.

The `ON ROW CHANGE` block is only executed if at least one field value in the current row has changed since the row was entered, and the modification flag of the field is set. The modified field(s) might not be the current field, and several field values can be changed. Values might have been changed by the user or by the program. The modification flag is reset for all fields when entering another row, when going to another sub-dialog, or when leaving the dialog instruction.

`ON ROW CHANGE` is executed after the `AFTER FIELD` block and before the `AFTER ROW` block.

When called in this block, DIALOG.getCurrentRow() / `arr_curr()` return the index of the current row that has been changed.

You can, for example, code database modifications (`UPDATE`) in the `ON ROW CHANGE` block:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   ON ROW CHANGE
     LET r = DIALOG.getCurrentRow("s_items")
     UPDATE items SET
             items.item_code        = p_items[r].item_code,
             items.item_description = p_items[r].item_description,
             items.item_price       = p_items[r].item_price,
             items.item_updatedate  = TODAY
            WHERE items.item_num = p_items[r].item_num
```

AFTER ROW block

AFTER ROW block in singular and parallel DISPLAY ARRAY, INPUT ARRAY dialogs

In a singular `DISPLAY ARRAY`, `INPUT ARRAY` instruction, or when used as parallel dialog, the `AFTER ROW` block is executed each time the user moves to another row, before the current row is left. This trigger can also be executed in other situations, such as when you delete a row, or when the user inserts a new row.

A `NEXT FIELD` instruction executed in the `AFTER ROW` control block will keep the user entry in the current row. Use this behavior to implement row validation and prevent the user from leaving the list or moving to another row.

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the row that you are leaving.

AFTER ROW block in DISPLAY ARRAY and INPUT ARRAY of procedural DIALOG

In an `INPUT` or `INPUT ARRAY` sub-dialog of a procedural `DIALOG` instruction, the `AFTER ROW` block is executed when a `DISPLAY ARRAY` or `INPUT ARRAY` list loses the focus, or when the user moves to another row in a list. This trigger can also be executed in other situations, for example when you delete a row, or when the user inserts a new row.

`AFTER ROW` is executed after the `AFTER FIELD`, `AFTER INSERT` and before `AFTER DISPLAY` or `AFTER INPUT` blocks.

When called in this block, `DIALOG.getCurrentRow()` / `arr_curr()` return the index of the of the row that you are leaving.

For both `INPUT ARRAY` and `DISPLAY ARRAY` sub-dialogs, a `NEXT FIELD` executed in the `AFTER ROW` control block will keep the focus in the list and stay in the current row. Use this feature to implement row validation and prevent the user from leaving the list or moving to another row.

AFTER ROW and temporary rows in INPUT ARRAY

> **Important: After creating a temporary row at the end of a list driven by INPUT ARRAY, if you leave that row to a previous row without data input (setting the touched flag), or when the cancel action is invoked, the temporary row will be automatically removed. The `AFTER ROW` block will be executed for the temporary row, but `ui.Dialog.getCurrentRow()`/`arr_curr()` will be one row greater than `ui.Dialog.getArrayLength()`/`ARR_COUNT()`. In this case, you should ignore the `AFTER ROW` event. For example, you should not try to execute a `NEXT FIELD` or `CONTINUE INPUT` instruction, nor should you try to access the dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array.**

In this example, the `AFTER ROW` block checks the current row index and verifies a variable value to forces the focus to stay in the current row if the value is wrong:

```
INPUT ARRAY p_items FROM s_items.*
   ...
   AFTER ROW
      LET r = DIALOG.getCurrentRow("s_items")
      IF r <= DIALOG.getArrayLength("s_items") THEN
        IF NOT item_is_valid_quantity(p_item[r].item_quantity) THEN
           ERROR "Item quantity is not valid"
           NEXT FIELD item_quantity'
         END IF
      END IF
```

Another way to handle the case of temporary rows in AFTER ROW is to use a flag to know if the AFTER INSERT block was executed: The AFTER INSERT block is not executed if the temporary row is automatically removed. By setting a first value in BEFORE INSERT and changing the flag in AFTER INSERT, you can detect if the row was permanently added to the list:

```
INPUT ARRAY p_items FROM s_items.*
  ...
  BEFORE INSERT
    LET op = "T"
    ...
  AFTER INSERT
    LET op = "I"
    ...
  AFTER ROW
    IF op == "I" THEN
      IF NOT item_is_valid_quantity(p_item[arr_curr()].item_quantity) THEN
         ERROR "Item quantity is not valid"
         NEXT FIELD item_quantity
        END IF
      WHENEVER ERROR CONTINUE
      INSERT INTO items (item_num, item_name, item_quantity)
                         VALUES ( p_item[arr_curr()].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<0 THEN
         ERROR "Could not insert the record into database!"
         NEXT FIELD CURRENT
      ELSE
         MESSAGE "Record has been inserted successfully"
      END IF
   END IF
...
```

BEFORE INSERT block

The BEFORE INSERT block is executed when a new row ins created in an INPUT ARRAY. You typically use this trigger to set some default values in the new created row. A new row can be created by moving down after the last row, by executing a insert action, or by executing an append action.

The BEFORE INSERT block is executed after the BEFORE ROW block and before the BEFORE FIELD block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the new created row.

To distinguish row insertion from an appended row, compare the current row (DIALOG.getCurrentRow("*screen-array*")) with the total number of rows (DIALOG.getArrayLength("*screen-array*")). If the current row index and the total number of rows correspond, the BEFORE INSERT concerns a temporary row, otherwise it concerns an inserted row.

Row creation can be stopped by using the CANCEL INSERT instruction inside BEFORE INSERT. If possible, it is however better to disable the insert and append actions to prevent the user to execute the actions with DIALOG.setActionActive().

In this example, the BEFORE INSERT block checks if the user can create rows and denies new row creation if needed; otherwise, it sets some default values:

```
INPUT ARRAY p_items FROM s_items.*
    ...
    BEFORE INSERT
      IF NOT user_can_append THEN
         ERROR "You are not allowed to append rows"
         CANCEL INSERT
      END IF
```

```
      LET r = DIALOG.getCurrentRow("s_items")
      LET p_items[r].item_num = get_new_serial("items")
      LET p_items[r].item_name = "undefined"
```

AFTER INSERT block

The AFTER INSERT block of INPUT ARRAY is executed when the creation of a new row is validated. In this block, you can for example implement SQL to insert a new row in the database table.

The AFTER INSERT block is executed afterthe AFTER FIELD block and before the AFTER ROW block.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the new created row.

When the user appends a new row at the end of the list, then moves UP to another row or validates the dialog, the AFTER INSERT block is only executed if at least one field was edited. If no data entry is detected, the dialog automatically removes the new appended row and thus does not trigger the AFTER INSERT block.

When executing a NEXT FIELD in the AFTER INSERT block, the dialog will keep the focus in the list and stay in the current row. Use this behavior to implement row input validation and prevent the user from leaving the list or moving to another row. However, this will not cancel the row insertion and will not invoke the BEFORE INSERT / AFTER INSERT triggers again. The only way to keep the focus in the current row after the row was inserted is to execute a NEXT FIELD in the AFTER ROW block.

In this example, the AFTER INSERT block inserts a new row in the database and cancels the operation if the SQL command fails:

```
INPUT ARRAY p_items FROM s_items.*
    ...
    AFTER INSERT
      WHENEVER ERROR CONTINUE
      INSERT INTO items VALUES
 ( p_items[DIALOG.getCurrentRow("s_items")].* )
      WHENEVER ERROR STOP
      IF SQLCA.SQLCODE<>0 THEN
         ERROR SQLERRMESSAGE
         CANCEL INSERT
      END IF
```

BEFORE DELETE block

The BEFORE DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, before the row is removed from the list.

You typically code the database table synchronization in the BEFORE DELETE block, by executing a DELETE SQL statement using the primary key of the current row. In the BEFORE DELETE block, the row to be deleted still exists in the program array, so you can access its data to identify what record needs to be removed.

The BEFORE DELETE block is executed before the AFTER DELETE block.

If needed, the deletion can be canceled with the CANCEL DELETE instruction.

When called in this block, DIALOG.getCurrentRow() / arr_curr() return the index of the row that will be deleted.

The next example uses the BEFORE DELETE block to remove the row from the database table and cancels the deletion operation if an SQL error occurs:

```
INPUT ARRAY p_items FROM s_items.*
  BEFORE DELETE
    LET r = DIALOG.getCurrentRow("s_items")
    WHENEVER ERROR CONTINUE
```

```
    DELETE FROM items
       WHERE item_num = p_items[r].item_num
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<>0 VALUES
      ERROR SQLERRMESSAGE
      CANCEL DELETE
    END IF
...
```

AFTER DELETE block

The AFTER DELETE block is executed each time the user deletes a row of an INPUT ARRAY list, after the row has been deleted from the list.

The AFTER DELETE block is executed after the BEFORE DELETE block and before the AFTER ROW block for the deleted row and the BEFORE ROW block of the new current row.

When an AFTER DELETE block executes, the program array has already been modified; the deleted row no longer exists in the array (except in the special case when deleting the last row). The arr_curr() function or the ui.Dialog.getCurrentRow() method returns the same index as in BEFORE ROW, but it is the index of the new current row. The AFTER ROW block is also executed just after the AFTER DELETE block.

> **Important: When deleting the last row of the list, AFTER DELETE is executed for the delete row, and DIALOG.getCurrentRow() / arr_curr() will be one higher as DIALOG.getArrayLength() / ARR_COUNT(). You should not access a dynamic array with a row index that is greater than the total number of rows, otherwise the runtime system will adapt the total number of rows to the actual number of rows in the program array. When using a static array, you must ignore the values in the rows after ARR_COUNT().**

Here the AFTER DELETE block is used to re-number the rows with a new item line number (note that DIALOG.getArrayLength() / ARR_COUNT() may return zero):

```
INPUT ARRAY p_items FROM s_items.*
   AFTER DELETE
      LET r = DIALOG.getCurrentRow("s_items")
      FOR i=r TO DIALOG.getArrayLength("s_items")
          LET p_items[i].item_lineno = i
      END FOR
...
```

It is not possible to use the CANCEL DELETE instruction in an AFTER DELETE block. At this time it is too late to cancel row deletion, as the data row no longer exists in the program array.

BEFORE MENU block

If the MENU block contains a BEFORE MENU clause, statements within this clause will be executed before the menu dialog starts.

This block is typically used to hide or disable some menu options according to the current context of the program. For example, when the current user is not allowed to create new records, the menu options can be disabled as follows:

```
MENU "Orders"
   BEFORE MENU
      CALL DIALOG.setActionActive("append", can_user_append() )
      ...
   COMMAND "Append" -- creates "append" action (lowercase)
      ...
   ...
END MENU
```

In TUI mode, the menu options can also be disabled, but they will still be displayed on the screen. The end user will see the option, but cannot select it. In this case it's more convenient to hide the option to the end user with the `DIALOG.setActionHidden()` method, instead of disabling the action.

DIALOG interaction blocks

Dialog interaction blocks are dialog triggers that can be used to execute specific code when the user executes an action in the dialog. For example, when pressing a button in the form, the corresponding `ON ACTION` interaction block will be executed.

Interaction blocks also include special handlers such as timeout event handler, drag & drop handlers, and modification triggers for `DISPLAY ARRAY` sub-dialogs.

- ON ACTION block on page 1056
- ON IDLE block on page 1046
- ON KEY block on page 1046
- ON APPEND block on page 1088
- ON INSERT block on page 1088
- ON UPDATE block on page 1089
- ON DELETE block on page 1090
- ON SELECTION CHANGE block on page 1090
- ON DRAG_START block on page 1091
- ON DRAG_FINISHED block on page 1091
- ON DRAG_ENTER block on page 1092
- ON DRAG_OVER block on page 1093
- ON DROP block on page 1094

ON ACTION block

The `ON ACTION` *`action-name`* blocks execute a sequence of instructions when the user triggers a specific action.

A typical action handler block looks like this:

```
ON ACTION action-name
   instruction
   ...
```

Action blocks will be bound by name to action views (like buttons) in the current form. Action views can be buttons in forms, toolbar buttons, topmenu options, and if no explicit action view is defined, actions are rendered with a default action view, depending on the type of front-end.

The next example defines an action block to open a typical zoom window and let the user select a customer record:

```
ON ACTION zoom
   CALL zoom_customers() RETURNING st, rec.cust_id, rec.cust_name
```

In a dialog handling user input such as `INPUT`, `INPUT ARRAY` and `CONSTRUCT`, if an action is specific to a field, add the `INFIELD` clause to have the action automatically enabled when the corresponding field gets the focus:

```
ON ACTION zoom INFIELD cust_city
   CALL zoom_cities() RETURN st, rec.cust_city
```

In most cases actions are decoration with action defaults in form files, but there can be cases where the `ON ACTION` handler needs to define its own attributes at the program level. This can be done by adding the `ATTRIBUTES()` clause of `ON ACTION`:

```
ON ACTION custinfo ATTRIBUTES(DISCLOSUREINDICATOR, IMAGE="info")
```

```
      CALL show_customer_info()
```

For more details about action handlers, and action configuration, see Dialog actions on page 1276.

ON IDLE block

The `ON IDLE` *`seconds`* clause defines a set of instructions that must be executed after a given period of user inactivity. This interaction block can be used, for example, to quit the dialog after the user has not interacted with the program for a specified period of time.

The parameter of `ON IDLE` must be an integer literal or variable. If it the value is zero, the dialog timeout is disabled.

It is not recommended to use the `ON IDLE` trigger with a short timeout period such as 1 or 2 seconds; The purpose of this trigger is to give the control back to the program after a relatively long period of inactivity (10, 30 or 60 seconds). This is typically the case when the end user leaves the workstation, or got a phone call. The program can then execute some code before the user gets the control back.

```
ON IDLE 30
   IF ask_question(
      "Do you want to reload information the database?") THEN
      -- Fetch data back from the db server
   END IF
```

> **Important:** The timeout value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, any change of the variable will have no effect if the variable is changed after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON KEY block

An `ON KEY (`*`key-name`*`)` block defines an action with a hidden action view (no default button is visible), that executes a sequence of instructions when the user presses the specified key.

The `ON KEY` block is supported for backward compatibility with TUI mode applications.

An `ON KEY` block can specify up to four different keys. Each key creates a specific action objects that will be identified by the key name in lowercase. For example, `ON KEY(F5,F6)` creates two actions with the names `f5` and `f6`. Each action object will get an `ACCELERATORNAME` assigned with the corresponding accelerator name. The specified keys must be one of the virtual keys.

In GUI mode, action defaults are applied for `ON KEY` actions by using the name of the action (the key name). You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the factory action defaults file provided by default.

By default, `ON KEY` actions are not decorated with a default button in the action frame (the default action view). You can show the default button by configuring a `text` attribute with the action defaults.

```
ON KEY (CONTROL-Z)
   CALL open_zoom()
```

ON TIMER block

The `ON TIMER` *`seconds`* clause defines a set of instructions that must be executed at regular intervals. This interaction block can be used, for example, to check if a message has arrived in a queue, and needs to be processed.

The parameter of ON TIMER must be an integer literal or variable. If the value is zero, the dialog timeout is disabled.

It is not recommended to use the ON TIMER trigger with a short timeout period, such as 1 or 2 seconds. The purpose of this trigger is to give the control back to the program after a reasonable period of time, such as 10, 20 or 60 seconds.

```
ON TIMER 30
   CALL check_for_messages()
```

> **Important:** The timer value is taken into account when the dialog initializes its internal data structures. If you use a program variable instead of an integer constant, a change of the variable has no effect if the change takes place after the dialog has initialized. If you what to change the value of the timeout variable, it must be done before the dialog block.

ON APPEND block

Similarly to the ON INSERT control block, the ON APPEND trigger can be used to enable row creation during a DISPLAY ARRAY dialog. If this block is defined, the dialog will automatically create the append action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an ON ACTION append interaction block and the ON APPEND block is used, the compiler will stop with error -8408.

When the user fires the append action, the dialog first execute the user code of the AFTER ROW block if defined. Then the dialog moves to the end of the list, and creates a new row after the last existing row. After creating the row, the dialog executes the user code of the ON APPEND block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular INPUT statement, to let the end user enter data for the new created row. This is typically done with an INPUT binding explicitly array fields to the screen record fields. The new current row in the program array is identified with arr_curr(), and the current screen line in the form is defined by SCR_LINE():

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON APPEND
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the INPUT instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the INT_FLAG global variable is FALSE and STATUS is zero, the new row is kept in the program array, and the BEFORE ROW block is executed for the new created row.
- If the INT_FLAG global variable is TRUE or STATUS is different from zero, the new row is removed from the program array, and the BEFORE ROW block is executed for the row that was existing at the current position, before the new row was created.

The DISPLAY ARRAY dialog always resets INT_FLAG to FALSE and STATUS to zero before executing the user code of the ON APPEND block.

The append action is disabled if the maximum number of rows is reached.

If needed, the ON APPEND handler can be configured with action attributes by added an ATTRIBUTES() clause, as with user-defined action handlers:

```
  ON APPEND ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON INSERT block

Similarly to the `ON APPEND` control block, the `ON INSERT` trigger can be used to enable row creation during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the insert action. This action can be decorated, enabled and disabled as a regular action.

If the dialog defines an `ON ACTION insert` interaction block and the `ON INSERT` block is used, the compiler will stop with error -8408.

When the user fires the insert action, the dialog first execute the user code of the `AFTER ROW` block if defined. Then the new row is created: The insert action creates a new row before current row in the list. After creating the row, the dialog executes the user code of the `ON INSERT` block.

The dialog handles only row creation actions and navigation, you must program the record input with a regular `INPUT` statement, to let the end user enter data for the new created row. This is typically done with an `INPUT` binding explicitly array fields to the screen record fields. The new current row in the program array is identified with `arr_curr()`, and the current screen line in the form is defined by `scr_line()`:

```
DISPLAY ARRAY arr TO sr.*
 ...
  ON INSERT
    INPUT arr[arr_curr()].* FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the `INPUT` instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the new row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the new row is kept in the program array, and the `BEFORE ROW` block is executed for the new created row.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the new row is removed from the program array, and the `BEFORE ROW` block is executed for the row that was existing at the current position, before the new row was created.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON INSERT` block.

The insert action is disabled if the maximum number of rows is reached.

If needed, the `ON INSERT` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
  ON INSERT ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON UPDATE block

The `ON UPDATE` trigger can be used to enable row modification during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the update action. This action can be decorated, enabled and disabled as regular actions.

You typically configure the `TABLE` container in the form by defining the `DOUBLECLICK` attribute to "update", in order to trigger the update action when the user double-clicks on a row.

If the dialog defines an `ON ACTION update` interaction block and the `ON UPDATE` block is used, the compiler will stop with error -8408.

When the user fires the *update* action, the dialog executes the user code of the `ON UPDATE` block.

The dialog handles only the row modification action and navigation, you must program the record input with a regular `INPUT` statement, to let the end user modify the data of the current row. This is typically done with an `INPUT` binding explicitly array fields to the screen record fields, with the `WITHOUT DEFAULTS`

clause. The current row in the program array is identified with `arr_curr()`, and the current screen line in the form is defined by `scr_line()`:

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON UPDATE
    INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sr[scr_line()].* ;
    ...
```

Pay attention to the semicolon ending the `INPUT` instruction, which is usually needed here to solve a language grammar conflict when nested dialog instructions are implemented.

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the modified values of the current row are kept in the program array.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the old values of the current row are restored in the program array.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON UPDATE` block.

If needed, the `ON UPDATE` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
  ON UPDATE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON DELETE block

The `ON DELETE` trigger can be used to enable row deletion during a `DISPLAY ARRAY` dialog. If this block is defined, the dialog will automatically create the delete action. This action can be decorated, enabled and disabled as regular actions.

If the dialog defines an `ON ACTION delete` interaction block and the `ON DELETE` block is used, the compiler will stop with error -8408.

When the user fires the delete action, the dialog executes the user code of the `ON DELETE` block.

The dialog handles only the row deletion action and navigation, you can typically program a validation dialog box to let the user confirm the deletion. The current row in the program array is identified with `arr_curr()`:

```
DISPLAY ARRAY arr TO sr.*
  ...
  ON DELETE
    IF fgl_winQuestion("Delete",
          "Do you want to delete this record?",
          "yes", "no|yes", "help", 0) == "no"
    THEN
       LET int_flag = TRUE
    END IF
    ...
```

After the user code is executed, the dialog gets the control back and processes the current row as follows:

- If the `INT_FLAG` global variable is `FALSE` and `STATUS` is zero, the current row is deleted from the program array, and the `BEFORE ROW` block is executed for the next row in the list.
- If the `INT_FLAG` global variable is `TRUE` or `STATUS` is different from zero, the current row is kept in the program array, and the `BEFORE ROW` block is executed again for the current row.

The `DISPLAY ARRAY` dialog always resets `INT_FLAG` to `FALSE` and `STATUS` to zero before executing the user code of the `ON DELETE` block.

If needed, the `ON DELETE` handler can be configured with action attributes by added an `ATTRIBUTES()` clause, as with user-defined action handlers:

```
  ON DELETE ATTRIBUTES(TEXT=%"custlist.delete", IMAGE="listdel")
```

ON SELECTION CHANGE block

The `ON SELECTION CHANGE` trigger can be used to enable multi-row selection and detect when rows are selected or de-selected by the end user during a `DISPLAY ARRAY` dialog. If this block is defined, multi-row selection is automatically enableb. However, the feature can be enabled/disabled with the `setSelectionMode()` dialog method.

ON SORT block

The `ON SORT` interfaction block can be used to detect when rows have to be sorted in a `DISPLAY ARRAY` or `INPUT ARRAY` dialog.

`ON SORT` is used in two different contexts:

1. In a regular `DISPLAY ARRAY` / `INPUT ARRAY` dialog (not using paged mode), the `ON SORT` trigger can be used to detect that a list sort was performed. In this case, the (visual) sort is already done by the runtime system and the `ON SORT` block is only used to execute post-sort tasks, such as displaying current row information, by using arrayToVisualIndex() dialog method. It is also possible to get the sort column and order with the `getSortKey()` and `getSortSelection()` dialog methods.
2. In a `DISPLAY ARRAY` using paged mode (`ON FILL BUFFER`), built-in row sorting is not available because data is provided by pages. Use the `ON SORT` trigger to detect a sort request and perform a new SQL query to re-order the rows. In this case, sort column and order is available with the `getSortKey()` and `getSortSelection()` dialog methods. See Populating a DISPLAY ARRAY on page 1372.

ON DRAG_START block

The `ON DRAG_START` block is executed when the end user has begun the drag operation. If this dialog trigger has not been defined, default dragging is enabled for this dialog.

In the `ON DRAG_START` block, the program typically specifies the type of drag & drop operation by calling `ui.DragDrop.setOperation()` with "move" or "copy". This call will define the default and unique drag operation. If needed, the program can allow another type of drag operation with `ui.DragDrop.addPossibleOperation()`. The end user can then choose to move or copy the dragged object, if the drag & drop target allows it.

If the dragged object can be dropped outside the program, must define the MIME type and drag/drop data with `ui.DragDrop.setMimeType()` and `ui.DragDrop.setBuffer()` methods.

Example:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    CALL dnd.setOperation("move") -- Move is the default operation
    CALL dnd.addPossibleOperation("copy") -- User can toggle to copy if
 needed
    CALL dnd.setMimeType("text/plain")
    CALL dnd.setBuffer(arr[arr_curr()].cust_name)
    ...
END DISPLAY
```

ON DRAG_FINISHED block

Execution of the `ON DRAG_FINISHED` block notifies the dialog where the drag started that the drop operation has been completed or terminated.

Call `ui.DragDrop.getOperation()` to get the final type of operation of the drop. On successful completion, the method returns "move" or "copy"; otherwise the function returns `NULL`. If `NULL` is returned, the `ON DRAG_FINISHED` trigger can be ignored.

In cases of successful moves to a target out of the current `DISPLAY ARRAY`, the application must remove the transferred data from the source model. For example, if a row was moved from dialog A to B, dialog A will get an `ON DRAG_FINISHED` execution after the row was dropped into B, and should remove the row from the list A.

The `ON DRAG_FINISHED` interaction block is optional.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_START (dnd)
    LET last_dragged_row = arr_curr()
    ...
  ON DRAG_FINISHED (dnd)
    IF dnd.getOperation() == "move" THEN
      CALL DIALOG.deleteRow(last_dragged_row)
    END IF
    ...
END DISPLAY
```

ON DRAG_ENTER block

When the `ON DROP` control block is defined, the `ON DRAG_ENTER` block will be executed when the mouse cursor enters the visual boundaries of the drop target dialog. Entering the target dialog is accepted by default if no `ON DRAG_ENTER` block is defined. However, when `ON DROP` is defined, you should also define `ON DRAG_ENTER` to deny the drop of objects with an unsupported MIME type that come from other applications.

The program can decide to deny or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a `NULL` to the method will deny drop.

To check what MIME type is available in the drag & drop buffer, the program uses the `ui.DragDrop.selectMimeType()` method. This method takes the MIME type as a parameter and returns `TRUE` if the passed MIME type is used. You can call this method several times to check the availability of different MIME types.

You may also define the visual effect when flying over the target list with `ui.DragDrop.setFeedback()`.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    IF dnd.selectMimeType("text/plain") THEN
      CALL dnd.setOperation("copy")
      CALL dnd.setFeedback("all")
    ELSE
      CALL dnd.setOperation(NULL)
    END IF
  ON DROP (dnd)
    ...
END DISPLAY
```

Once the mouse has entered the target area, subsequent mouse cursor moves can be detected with the `ON DRAG_OVER` trigger.

When using a table or tree-view as drop target, you can control the visual effect when the mouse flies over the rows, according to the type of drag & drop you want to achieve.

Basically, a dragged object can be:

1. Inserted in between two rows (visual effect must show where the object will be inserted)
2. Copied/merged to the current row (visual effect must show the row under the mouse)
3. Dropped somewhere on the target widget (the exact location inside the widget does not matter)

The visual effect can be defined with the `ui.DragDrop.setFeedback()` method, typically called in the `ON DRAG_ENTER` block.

The values to pass to the `setFeedback()` method to get the desired visual effects described are respectively:

1. `insert` (default)
2. `select`
3. `all`

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    IF canDrop() THEN
      CALL dnd.setOperation(NULL)
    ELSE
      CALL dnd.setFeedback("select")
    END IF
    ...
END DISPLAY
```

ON DRAG_OVER block

When the `ON DROP` control block is defined, the `ON DRAG_OVER` block will be executed after `ON DRAG_ENTER`, when the mouse cursor is moving over the drop target, or when the drag & drop operation has changed (toggling copy/move).

`ON DRAG_OVER` will be called only once per row, even if the mouse cursor moves over the row.

In the `ON DRAG_OVER` block, the method `ui.DragDrop.getLocationRow()` returns the index of the row in the target array, and can be used to allow or deny the drop. When using a tree-view, you must also check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and allow/deny the drop operation accordingly.

The program can change the drop operation at any execution of the `ON DRAG_OVER` block. You can deny or allow a specific drop operation with a call to `ui.DragDrop.setOperation()`; passing a `NULL` to the method will deny the drop.

The current operation (returned by `ui.DragDrop.getOperation()`) is the value set in previous `ON DRAG_ENTER` or `ON DRAG_OVER` events, or the operation selected by the end user, if it can toggle between copy and move. Thus, `ON DRAG_OVER` can occur even if the mouse position has not changed.

If dropping has been denied with `ui.DragDrop.setOperation(NULL)` in the previous `ON DRAG_OVER` event, the program can reset the operation to allow a drop with a call to `ui.DragDrop.setOperation()` with the operation parameter "move" or "copy".

`ON DRAG_OVER` will not be called if drop has been disabled in `ON DRAG_ENTER` with `ui.DragDrop.setOperation(NULL)`

`ON DRAG_OVER` is optional, and must only be defined if the operation or the acceptance of the drag object depends on the target row of the drop target.

```
DEFINE dnd ui.DragDrop
```

```
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    ...
  ON DRAG_OVER (dnd)
    IF arr[dnd.getLocationRow()].acceptsCopy THEN
      CALL dnd.setOperation("copy")
    ELSE
      CALL dnd.setOperation(NULL)
    END IF
  ON DROP (dnd)
    ...
END DISPLAY
```

During a drag & drop process, the end user (or the target application) can decide to modify the type of the operation, to indicate whether the dragged object has to be copied or moved from the source to the target. For example, in a typical file explorer, by default files are moved when doing a drag & drop on the same disk. To make a copy of a file, you must press the Ctrl key while doing the drag & drop with the mouse.

In the drop target dialog, you can detect such operation changes in the `ON DRAG_OVER` trigger and query the `ui.DragDrop` object for the current operation with `ui.DragDrop.getOperation()`. In the drag source dialog, you typically check `ui.DragDrop.getOperation()` in the `ON DRAG_FINISHED` trigger to know what sort of operation occurred, to keep ("copy" operation) or delete ("move" operation) the original dragged object.

This example tests the current operation in the drop target list and displays a message accordingly:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DRAG_ENTER (dnd)
    ...
  ON DRAG_OVER (dnd)
    CASE dnd.getOperation()
    WHEN "move"
      MESSAGE "The object will be moved to row ", dnd.getLocationRow()
    WHEN "copy"
      MESSAGE "The object will be copied to row ", dnd.getLocationRow()
    END CASE
    ...
  ON DROP (dnd)
    ...
END DISPLAY
```

ON DROP block

To enable drop actions on a list, you must define the `ON DROP` block; otherwise the list will not accept drop actions.

The `ON DROP` block is executed after the end user has released the mouse button to drop the dragged object. `ON DROP` will not occur if drop has been denied in the previous `ON DRAG_OVER` event or in `ON DRAG_ENTER` with a call to `ui.DragDrop.setOperation(NULL)`.

The program might also check the MIME type of the dragged object with `ui.DragDrop.getSelectedMimeType()`, and then call the `ui.DragDrop.getBuffer()` method to retrieve drag & drop data from external applications.

Ideally the drop operation should be accepted (no additional call to `ui.DragDrop.setOperation()`).

In this block, the `ui.DragDrop.getLocationRow()` method returns the index of the row in the target array, and can be used to execute the code to get the drop data / object into the row that has been chosen by the user.

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
  ...
  ON DROP (dnd)
    LET arr[dnd.getLocationRow()].capacity == dnd.getBuffer()
    ...
END DISPLAY
```

If the drag & drop operations are local to the same list or tree-view controller, you can use the `ui.DragDrop.dropInternal()` method to simplify the code. This method implements the typical move of the dragged rows or tree-view node. This is especially useful in case of a tree-view, but is also the preferred way to move rows around in simple tables.

This `ON DROP` code example uses the `dropInternal()` method:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr_tree TO sr_tree.* ...
  ...
  ON DROP (dnd)
    CALL dnd.dropInternal()
    ...
END DISPLAY
```

If you want to implement by hand the code to drop a node in a tree-view, you must check the index returned by the `ui.DragDrop.getLocationParent()` method to detect if the object was dropped as a sibling or as a child node, and execute the code corresponding to the drop operation: If the drop target row index returned by `getLocationRow()` is a child of the parent row index returned by `getLocationParent()` , the new row must be inserted before `getLocationRow()`; otherwise the new row must be added as a child of the parent node identified by `getLocationParent()`.

DIALOG control instructions

Dialog control instructions are language instructions dedicated to dialog control, to programmatically force the dialog to behave in a given way.

For example the `NEXT FIELD` instruction forces the focus to a specific form field.

- NEXT FIELD instruction on page 1121
- CLEAR instruction in dialogs on page 1073
- DISPLAY TO / BY NAME instruction on page 1192
- CONTINUE DIALOG instruction on page 1192
- EXIT DIALOG instruction on page 1193
- ACCEPT DIALOG instruction on page 1193
- CANCEL DELETE instruction on page 1119
- CANCEL INSERT instruction on page 1120

NEXT FIELD instruction

Understanding the NEXT FIELD instruction

The `NEXT FIELD` *`field-name`* instruction gives the focus to the specified field and forces the dialog to stay in that field.

This instruction can be used to control field input, in BEFORE FIELD, ON CHANGE or AFTER FIELD blocks, it can also force a DISPLAY ARRAY or INPUT ARRAY to stay in the current row when NEXT FIELD is used in the AFTER ROW block.

If it exists, the BEFORE FIELD block of the corresponding field is executed.

The purpose of the NEXT FIELD instruction is give the focus to an editable field. Make sure that the field specified in NEXT FIELD is active, or use NEXT FIELD CURRENT. Non-editable fields are fields defined with the NOENTRY attribute, fields disabled at runtime with DIALOG.setFieldActive(), or fields using a widget that does not allow input, such as a LABEL.

Instead of the NEXT FIELD instruction, you can use the DIALOG.nextField("*field-name*") method to register a field, for example when the name is not known at compile time. However, this method only registers the field: It does not stop code execution, like the NEXT FIELD instruction does. You must execute a CONTINUE DIALOG to get the same behavior as NEXT FIELD.

Form field identification with NEXT FIELD

With the NEXT FIELD instruction, fields are identified by the form field name specification, not the program variable name used by the dialog. Form fields are bound to program variables with the binding clause of dialog instruction (INPUT *variable-list* FROM *field-list*, INPUT BY NAME *variable-list*, CONSTRUCT BY NAME *sql* ON *column-list*,CONSTRUCT *sql* ON *column-list* FROM *field-list*, INPUT ARRAY *array-name* FROM *screen-array*.*).

The field name specification can be any of the following:

- *field-name*
- *table-name.field-name*
- *screen-record-name.field-name*
- FORMONLY.*field-name*

Here are some examples:

- "cust_name"
- "customer.cust_name"
- "cust_screen_record.cust_name"
- "item_screen_array.item_label"
- "formonly.total"

When no field name prefix is used, the first form field matching that simple field name is used.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog according to the field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using INPUT BY NAME *variable-list*), the field prefix must be the database table name (or FORMONLY) used in the form file, or any valid screen-record using that field. When the FROM clause of the dialog specifies an explicit screen-record (for example, in INPUT *variable-list* FROM *screen-record.* / field-list-with-screen-record-prefix* or INPUT ARRAY *array-name* FROM *screen-array*.*), the field prefix must be the screen-record name used in the FROM clause.

Abstract field identification is supported with the CURRENT, NEXT and PREVIOUS keywords. These keywords represent the current, next and previous fields respectively. When using FIELD ORDER FORM, the NEXT and PREVIOUS options follow the tabbing order defined by the form. Otherwise, they follow the order defined by the input binding list (with the FROM or BY NAME clause).

In a procedural dialog, if the focus is in the first field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD PREVIOUS will jump out of the current sub-dialog and set the focus to the previous sub-dialog. If the focus is in the last field of an INPUT or CONSTRUCT sub-dialog, NEXT FIELD NEXT will jump out of the current sub-dialog and set the focus to the next sub-dialog. NEXT FIELD NEXT or NEXT FIELD PREVIOUS also jumps to another sub-dialog when the focus is in a DISPLAY ARRAY sub-dialog. However, when using an INPUT ARRAY sub-dialog, NEXT FIELD NEXT from within the last column will loop to the first

column of the current row, and NEXT FIELD PREVIOUS from within the first column will jump to the last column of the current row - the focus stays in the current INPUT ARRAY sub-dialog. When another sub-dialog gets the focus because of a NEXT FIELD NEXT/PREVIOUS, the newly-selected field depends on the sub-dialog type, following the tabbing order as if the end-user had pressed the tab or Shift-Tab key combination.

NEXT FIELD to a non-editable INPUT / INPUT ARRAY / CONSTRUCT field

Non-editable fields are fields defined with the NOENTRY attribute, fields disabled with ui.Dialog.setFieldActive("*field-name*", FALSE), or fields using a widget that does not allow input, such as a LABEL.

If a NEXT FIELD instruction specifies a non-editable field, the BEFORE FIELD block of that field is executed. Then the dialog tries to give the focus to that field. Since the field cannot get the focus, the dialog will perform the last pressed navigation key (Tab, Shift-Tab, Left, Right, Up, Down, Accept) and execute the related control blocks, including the AFTER FIELD block of the non-editable field. If no last key is identified, the dialog considers Tab as fallback and moves to the next editable field as defined by the FIELD ORDER mode used by the dialog. Doing a NEXT FIELD to a non-editable field can lead to infinite loops in the dialog; Use NEXT FIELD CURRENT instead.

When selecting a non-editable field with NEXT FIELD NEXT, the runtime system will re-select the current field since it is the next editable field in the dialog. As a result the end user sees no change.

NEXT FIELD in procedural DIALOG blocks

In a procedural dialog block, the NEXT FIELD *field-name* instruction gives the focus to the specified field controlled by INPUT, INPUT ARRAY or CONSTRUCT, or to a read-only list when using DISPLAY ARRAY.

When using a DISPLAY ARRAY sub-dialog, it is possible to give the focus to the list, by specifying the name of the first column as argument for NEXT FIELD.

If the target field specified in the NEXT FIELD instruction is inside the current sub-dialog, neither AFTER FIELD nor AFTER ROW will be invoked for the field or list you are leaving. However, the BEFORE FIELD control blocks of the destination field (or the BEFORE ROW in case of read-only list) will be executed.

If the target field specified in the NEXT FIELD instruction is outside the current sub-dialog, the AFTER FIELD, AFTER INSERT, AFTER ROW and AFTER INPUT/DISPLAY/CONSTRUCT control blocks will be invoked for the field or list you are leaving. Form-level validation rules will also be checked, as if the user had selected the new sub-dialog himself. This guarantees the current sub-dialog is left in a consistent state. The BEFORE INPUT/DISPLAY/CONSTRUCT, BEFORE ROW and the BEFORE FIELD control blocks of the destination field / list will then be executed.

NEXT FIELD in record list control blocks

When using NEXT FIELD in AFTER ROW or in ON ROW CHANGE of a DISPLAY ARRAY or INPUT ARRAY, the dialog will stay in the current row and give control back to the user. This behavior allows you to implement data input rules:

```
AFTER ROW
  IF NOT int_flag AND arr_count()<=arr_curr() THEN
     IF arr[arr_curr()].it_count * arr[arr_curr()].it_value > maxval THEN
        ERROR "Amount of line exceeds max value."
        NEXT FIELD item_count
     END IF
  END IF
```

CLEAR instruction in dialogs

The CLEAR *field-list* and CLEAR SCREEN ARRAY *screen-array.** instructions clear the value buffer of specified form fields. The buffers are directly changed in the current form, and the program

variables bound to the dialog are left unchanged. CLEAR can be used outside any dialog instruction, such as the DISPLAY BY NAME / TO instructions.

When a dialog is configured with the UNBUFFERED mode, there is no reason to clear field buffers since any variable assignment will synchronize field buffers. Actually, changing the field buffers with DISPLAY or CLEAR instruction in an UNBUFFERED dialog will have no visual effect, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. To clear fields of an UNBUFEFERED dialog, just set to NULL the variables bound to the dialog. However, when using a CONSTRUCT, no program variables are associated to the dialog and no UNBUFFERED concept exits, and the CLEAR or DISPLAY TO / BY NAME instructions are the only way to modify the CONSTRUCT fields.

A screen array with a screen-line specification doesn't make much sense in a GUI application using TABLE containers, you can therefore use the CLEAR SCREEN ARRAY instruction to clear all rows of a list.

DISPLAY TO / BY NAME instruction

The DISPLAY *variable-list* TO *field-list* or DISPLAY BY NAME *variable-list* instruction fills the value buffers of specified form fields with the values contained in the specified program variables. The DISPLAY instruction changes the buffers directly in the current form, not the program variables bound to the dialog. DISPLAY can be used outside any dialog instruction, in the same way as the CLEAR instruction. DISPLAY also sets the modification flag of fields.

As DIALOG is typically used with the UNBUFFERED mode, there is no reason to set field buffers in a DIALOG block since any variable assignment will synchronize field buffers. Actually, changing the field buffers with the DISPLAY or CLEAR instruction will have no visual effect if the fields are used by a dialog working in UNBUFFERED mode, because the variables bound to the dialog will be used to reset the field buffer just before giving control back to the user. So if you want to set field values, just assign the variables and the fields will be synchronized. However, when using a CONSTRUCT binding, you may want to set field buffers with this DISPLAY instruction, as there are no program variables bound to fields (with CONSTRUCT, only one string variable is bound to hold the SQL condition).

Instead of using a DISPLAY instruction to set the modification flag of fields to simulate user input, use the DIALOG.setFieldTouched() method instead.

CONTINUE DIALOG instruction

The CONTINUE DIALOG statement continues the execution of a DIALOG instruction, skipping all statements appearing after this instruction.

Control returns to the dialog instruction, which executes remaining control blocks as if the program reached the end of the current control block. Then the control goes back to the user and the dialog waits for a new event.

The CONTINUE DIALOG statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user by skipping the rest of the statements.

In the following code example, an ON ACTION block gives control back to the dialog, skipping all instructions below line 04:

```
ON ACTION zoom
  IF p_cust.cust_id IS NULL OR p_cust.cust_name IS NULL THEN
    ERROR "Zoom window cannot be opened if no info to identify customer"
    CONTINUE DIALOG
  END IF
  IF p_cust.cust_address IS NULL THEN
     ...
```

If CONTINUE DIALOG is called in a control block that is not AFTER DIALOG, further control blocks might be executed according to the context. Actually, CONTINUE DIALOG just instructs the dialog to continue as if the code in the control block was terminated (it is a kind of GOTO end_of_control_block). However, when executed in AFTER DIALOG, the focus returns to the current field or read-only list. In this case the BEFORE ROW and BEFORE FIELD triggers will be invoked.

A CONTINUE DIALOG in AFTER FIELD, AFTER INPUT, AFTER DISPLAY or AFTER CONSTRUCT will only stop the program flow of the current block of statements; instructions after CONTINUE DIALOG will not be executed. If the user has selected a field in a different sub-dialog, this new field will get the focus and all necessary AFTER / BEFORE control blocks will be executed.

In case of input error in a field, the best practice is to use a NEXT FIELD instruction to stay in the dialog and set the focus to the field that the user has to correct.

EXIT DIALOG instruction

The EXIT DIALOG statement terminates a procedural DIALOG block without any further control block execution.

> **Note:** When used in a declarative DIALOG block, the EXIT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

Program flow resumes at the instruction following the END DIALOG keywords. Blocks such as AFTER DIALOG will not be executed.

```
   ON ACTION quit
       EXIT DIALOG
```

When leaving the DIALOG instruction, all form items used by the dialog will be disabled until another interactive statement takes control.

ACCEPT DIALOG instruction

The ACCEPT DIALOG statement validates all input fields bound to the DIALOG instruction and leaves the block if no error is raised.

> **Note:** When used in a declarative DIALOG block, the ACCEPT DIALOG instruction does only make sense when the declarative dialog block is included in a procedural dialog block with the SUBDIALOG clause.

When defined in the dialog block, ON CHANGE, AFTER FIELD, AFTER ROW, AFTER INPUT/DISPLAY/CONSTRUCT control blocks will be executed when ACCEPT DIALOG is performed.

The statements appearing after the ACCEPT DIALOG instruction will be skipped.

You typically code an ACCEPT DIALOG in an ON ACTION accept block:

```
ON ACTION accept ACCEPT DIALOG
```

Note that any usage of ACCEPT DIALOG outside an ON ACTION accept block is not intended and its behavior is undocumented.

Input field validation is a process that does several successive validation tasks:

1. The current field value is checked, according to the program variable data type (for example, the user must input a valid date in a DATE field).
2. NOT NULL field attributes are checked for all input fields. This attribute forces the field to have a value set by program or entered by the user. If the field contains no value, the constraint is not satisfied. Input values are right-trimmed, so if the user inputs only spaces, this corresponds to a NULL value which does not fulfill the NOT NULL constraint.
3. REQUIRED field attributes are checked for all input fields. This attribute forces the field to have a default value, or to be modified by the user or by program with a DISPLAY TO / BY NAME or DIALOG.setFieldTouched() call. If the field was not modified during the dialog, the REQUIRED constraint is not satisfied.
4. INCLUDE field attributes are checked for all input fields. This attribute forces the field to contain a value that is listed in the include list. If the field contains a value that is not in the list, the constraint is not satisfied.

If a field does not satisfy one of these constraints, dialog termination is canceled, an error message is displayed, and the focus goes to the first field causing a problem.

After input field validation has succeeded, different types of control blocks will be executed, such as `AFTER FIELD`, `AFTER ROW`, `AFTER INPUT` and `AFTER DIALOG`.

In order to validate some parts of the dialog without leaving the block, use the `DIALOG.validate()` method.

CANCEL DELETE instruction

In a list controlled by an `INPUT ARRAY`, row deletion can be canceled by using the `CANCEL DELETE` instruction in the `BEFORE DELETE` block. Using this instruction in a different place will generate a compilation error.

When the `CANCEL DELETE` instruction is executed, the current `BEFORE DELETE` block is terminated without any other trigger execution (no `BEFORE ROW` or `BEFORE FIELD` is executed), and the program execution continues in the user event loop.

You can, for example, prevent row deletion based on some condition:

```
BEFORE DELETE
   IF user_can_delete() == FALSE THEN
      ERROR "You are not allowed to delete rows"
      CANCEL DELETE
   END IF
```

The instructions that appear after `CANCEL DELETE` will be skipped.

If the row deletion condition is known before the delete action occurs, disable the delete action to prevent the user from performing a delete row action with the `DIALOG.setActionActive()` method:

```
CALL DIALOG.setActionActive("delete", FALSE)
```

It is also possible to prevent the user from deleting rows with the `DELETE ROW = FALSE` option in the `ATTRIBUTE` clause.

CANCEL INSERT instruction

In a list controlled by an `INPUT ARRAY`, row creation can be canceled by the program with the `CANCEL INSERT` instruction. This instruction can only be used in the `BEFORE INSERT` and `AFTER INSERT` control blocks. If it appears at a different place, the compiler will generate an error.

The instructions that appear after `CANCEL INSERT` will be skipped.

If the row creation condition is known before the insert/append action occurs, disable the insert and/or append actions to prevent the user from creating new rows, with `DIALOG.setActionActive()`:

```
CALL DIALOG.setActionActive("insert", FALSE)
CALL DIALOG.setActionActive("append", FALSE)
```

However, this will not prevent the user from appending a new temporary row at the end of the list, when moving down after the last row. To prevent row creation completely, use the `INSERT ROW = FALSE` and `APPEND ROW =FALSE` options in the `ATTRIBUTE` clause of `INPUT ARRAY`, or combine with the `AUTO APPEND = FALSE` attribute.

CANCEL INSERT in BEFORE INSERT

A `CANCEL INSERT` executed inside a `BEFORE INSERT` block prevents the new row creation. The following tasks are performed:

1. No new row will be created (the new row is not yet shown to the user).
2. The `BEFORE INSERT` block is terminated (further instructions are skipped).

3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. Control goes back to the user.

You can, for example, cancel a row creation if the user is not allowed to create rows:

```
BEFORE INSERT
   IF NOT user_can_insert THEN
      ERROR "You are not allowed to insert rows"
      CANCEL INSERT
   END IF
```

Executing CANCEL INSERT in BEFORE INSERT will also cancel a temporary row creation, except when there are no more rows in the list. In this case, CANCEL INSERT will just be ignored and leave the new row as is (otherwise, the instruction would loop without end). You can prevent automatic temporary row creation with the AUTO APPEND=FALSE attribute. If AUTO APPEND=FALSE and a CANCEL INSERT is executed in BEFORE INSERT (user has invoked an append action), the temporary row will be deleted and list will remain empty if it was the last row.

CANCEL INSERT in AFTER INSERT

A CANCEL INSERT executed inside an AFTER INSERT block removes the newly created row. The following tasks are performed:

1. The newly created row is removed from the list (the row exists now and user has entered data).
2. The AFTER INSERT block is terminated (further instructions are skipped).
3. The BEFORE ROW and BEFORE FIELD triggers are executed.
4. The control goes back to the user.

You can, for example, cancel a row insertion if a database error occurs when you try to insert the row into a database table:

```
AFTER INSERT
   WHENEVER ERROR CONTINUE
   LET r = DIALOG.getCurrentRow("s_items")
   INSERT INTO items VALUES ( p_items[r].* )
   WHENEVER ERROR STOP
   IF SQLCA.SQLCODE<>0 THEN
      ERROR SQLERRMESSAGE
      CANCEL INSERT
   END IF
```

Examples

Programming examples using parallel dialogs.

Example 1: Two independent record lists

Form file "simple_list.per":

```
LAYOUT
GRID
{
<T t1                           >
[c1  |c2                        ]
[c1  |c2                        ]
[c1  |c2                        ]
}
END
END
ATTRIBUTES
c1 = FORMONLY.col1;
c2 = FORMONLY.col2;
END
```

```
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

The module "list1.4gl":

```
DEFINE arr DYNAMIC ARRAY OF RECORD
           id INTEGER,
           name VARCHAR(50)
       END RECORD

FUNCTION start_list1()
    DEFINE i INTEGER
    IF ui.Window.forName("w_list1") IS NULL THEN
       FOR i=1 TO 10
           LET arr[i].id = i
           LET arr[i].name = "Record "||i
       END FOR
       OPEN WINDOW w_list1 WITH FORM "simple_list"
       START DIALOG control_list1
    ELSE
       CURRENT WINDOW IS w_list1
    END IF
END FUNCTION

FUNCTION terminate_list1()
    TERMINATE DIALOG control_list1
    CLOSE WINDOW w_list1
END FUNCTION

DIALOG control_list1()
    DISPLAY ARRAY arr TO sr.*
        ON ACTION add_row
           CALL DIALOG.appendRow("sr")
           LET arr[arr.getLength()].id = arr.getLength()
           LET arr[arr.getLength()].name = "[new record]"
        ON ACTION close
           CALL terminate_list1()
    END DISPLAY
END DIALOG
```

The module "list2.4gl" (quite the same code as list1.4gl):

```
DEFINE arr DYNAMIC ARRAY OF RECORD
           id INTEGER,
           name VARCHAR(50)
       END RECORD

FUNCTION start_list2()
    DEFINE i INTEGER
    IF ui.Window.forName("w_list2") IS NULL THEN
       FOR i=1 TO 10
           LET arr[i].id = i
           LET arr[i].name = "Record "||i
       END FOR
       OPEN WINDOW w_list2 WITH FORM "simple_list"
       START DIALOG control_list2
    ELSE
       CURRENT WINDOW IS w_list2
    END IF
END FUNCTION

FUNCTION terminate_list2()
```

```
    TERMINATE DIALOG control_list2
    CLOSE WINDOW w_list2
END FUNCTION

DIALOG control_list2()
    DISPLAY ARRAY arr TO sr.*
        ON ACTION clear_row
           INITIALIZE arr[arr_curr()].* TO NULL
        ON ACTION close
           CALL terminate_list2()
    END DISPLAY
END DIALOG
```

Program file:

```
IMPORT FGL list1
IMPORT FGL list2
MAIN
  OPTIONS INPUT WRAP
  CALL start_list1()
  CALL start_list2()
  WHILE fgl_eventloop()
  END WHILE
END MAIN
```

User interface programming

Describes how to program user interface and dialog instructions.

- Dialog programming basics on page 1249
- Dialog actions on page 1276
- Input fields on page 1260
- Table views on page 1345
- Tree views on page 1384
- Split views on page 1395
- Drag & drop on page 1411
- Web components on page 1416
- Canvases on page 1448
- Start menus on page 1454
- Window containers (WCI) on page 1458

Dialog programming basics

This section describes basic dialog programming concepts.

- The model-view-controller paradigm on page 1250
- Introducing dialogs on page 1250
- Dialog configuration with FGLPROFILE on page 1251
- The DIALOG control class on page 1252
- Dialog control functions on page 1252
- User interruption handling on page 1252
- Get program control if user inactivity on page 1254
- Get program control on a regular (timed) basis on page 1255

The model-view-controller paradigm

The dynamic user interface architecture is based on the Model-View-Controller (MVC) paradigm.

The model defines the object to be displayed (typically the application data that is stored in program variables). The view defines the decoration of the model (how the model must be displayed to the screen, this is typically the form). The controller is the interative instruction that implements the program code to handle the model.

Views are defined in the abstract user interface tree from compiled .42f forms loaded by programs. The program variables act as models, and you implement the controllers with interactive instructions, such as `DIALOG` or `INPUT`. Controllers also define action handlers that contain the program code to be executed when an action view is triggered.

Normally the controllers should not provide any decoration information, as that is the purpose of views. Because of the history of the language, however, some interactive instructions such as `MENU` define both the controller and some presentation information such as menu title, command labels, and comments. In this case, the runtime system automatically creates the view with that information; you can still associate other views to the same controller.

Introducing dialogs

Application forms are controlled by interactive instruction blocks called dialogs. These blocks perform the common tasks associated with the form, such as field input and action handling.

The *interactive instructions* allow the program to respond to user actions and data input.

Simple display (non-interactive)

The `DISPLAY BY NAME / TO` instruction allows you to display program variable data in the fields of a form and continue the program flow without giving control to the end user. This is in fact not an interactive instruction, as it just displays data to the current form, and returns immediately. However, it may be used in interactive instructions to display information to the end user. Note that when using the `UNBUFFERED` mode of a dialog, you do not need to is the DISPLAY BY NAME / TO instruction to synchronize program variables and form fields.

The `MESSAGE` and `ERROR` instructions are also simple display instructions without user interaction. These instructions are typically used to display a warning message to the end user.

The interactive dialog blocks

The singular `MENU` instruction handles a list of choices to activate a specific function of the program. No field input is possible with this instruction. The user can only select an action from the list.

The singular `INPUT` instruction is designed for simple record input. It enables the fields in a form for input, waits while the types data into the fields, and proceeds after the user accepts or cancels the dialog.

The singular `DISPLAY ARRAY` instruction is used to browse a list of records. It allows the user to view the contents of a program array of records, scrolling the record list on the screen and choosing a specific record. `DISPLAY ARRAY` implements by default a read-only list of records, but can be extended to become a modifiable list with list modification triggers such as `ON INSERT`.

The singular `INPUT ARRAY` instruction supports record list input. It allows the user to alter the contents of records of a program array, and to insert and delete records.

The singular `CONSTRUCT` instruction is designed to let the user enter search criteria for a database query. The user can enter a value or a range of values for one or several form fields, and your program looks up the database rows that satisfy the requirements.

The procedural `DIALOG` instruction (placed in the program flow) allows you to combine several `INPUT`, `DISPLAY ARRAY`, `INPUT ARRAY` and `CONSTRUCT` functionality within the same form.

The declarative `DIALOG` block (defined at the same level as a function) allows you to implement indiviual `MENU`, `INPUT`, `DISPLAY ARRAY`, `INPUT ARRAY` and `CONSTRUCT` functionality, that will perform in parallel

on sevaral forms, when used with the `START DIALOG` and `TERMINATE DIALOG` instructions. Declarative `DIALOG` blocks can also be associated to a procedural `DIALOG` instruction through the `SUBDIALOG` clause, it will then act as a procedural DIALOG sub-dialog.

Modal dialogs and parallel dialogs

Interactive instructions can be implemented as modal or parallel dialogs. *Modal dialogs* control a given window, and that window closes when the dialog is accepted or canceled. The window displays on the top of any existing windows which are not accessible while the modal dialog executes. *Parallel dialogs* allow access to several windows simultaneously; the user can switch from on window to the other.

Dialog configuration with FGLPROFILE

FGLPROFILE parameters can be used to configure dialog behavior.

By setting global parameters in FGLPROFILE, you can control the behavior of all dialogs of the program. These options are provided as global parameters to define a common pattern for all dialogs of your application. A complete description is available in the runtime configuration section.

List of FGLPROFILE entries affecting the behavior of dialogs:

1. `Dialog.fieldOrder` (only used by singular dialogs like INPUT)
2. `Dialog.currentRowVisibleAfterSort`

The Dialog.fieldOrder entry

```
Dialog.fieldOrder = {true|false}
```

The `Dialog.fieldOrder` FGLPROFILE entry defines the execution of `BEFORE FIELD` and `AFTER FIELD` triggers of intermediate fields.

When this parameter is set to **true**, as the end user moves to a new field with a mouse click, the runtime system executes the `BEFORE FIELD` and `AFTER FIELD` dialog control blocks of the input fields between the source field and the destination field. When the parameter is set to **false**, intermediate field triggers are not executed.

The `Dialog.fieldOrder` configuration parameter is ignored by the `DIALOG` multiple-dialog instruction or when using the `FIELD ORDER FORM` option in singular dialogs such as `INPUT`.

Do not use this feature for new developments: GUI applications allow users to jump from one field to any other field of the form by using the mouse. Therefore, it makes no sense to execute the `BEFORE FIELD` and `AFTER FIELD` triggers of intermediate fields in a graphical application.

> **Important:** The default setting for the runtime system is **false**; while the default setting in FGLPROFILE for `Dialog.fieldOrder` is **true**. As a result, the overall setting after installation is **true**. To modify the behavior of intermediate field trigger execution, change the setting of `Dialog.fieldOrder` in FGLPROFILE to **false**, or use the `FIELD ORDER FORM` program option.

The Dialog.currentRowVisibleAfterSort entry

```
Dialog.currentRowVisibleAfterSort = {true|false}
```

The `Dialog.currentRowVisibleAfterSort` FGLPROFILE entry controls the visibility of the current row after a sort in tables

When this parameter is set to **true**, the offset of table page is automatically adapted to show the current row after a sort. By default, the offset is not changed and current row may not be visible after sorting rows of a table. Changing this parameter has no impact on existing code, it is just an indicator to force the dialog to shift to the page of rows having the current row, as if the end-user had scrollbar. You can use this parameter to get the same behavior as well known e-mail readers.

The DIALOG control class

This topic explains the purpose of the `ui.DIALOG` class.

Inside a dialog instruction, the `DIALOG` predefined keyword represents the current dialog object. This dialog object can be used to execute methods provided by the `ui.Dialog` built-in class.

For example, you can enable or disable an action with the `setActionActive()` dialog method, or you can hide or show the default action view with the `setActionHidden()` method:

```
BEFORE INPUT
  CALL DIALOG.setActionActive("zoom",FALSE)
AFTER FIELD field1
  CALL DIALOG.setActionHidden("zoom",1)
```

The `setFieldActive()` method can be used to enable or disable a field during the dialog:

```
ON CHANGE custname
   CALL DIALOG.setFieldActive( "custaddr",
               (rec.custname IS NOT NULL) )
```

The `ui.Dialog` class provides also methods to configure the dialog, for example to enable multiple row selection:

```
BEFORE DIALOG
   CALL DIALOG.setSelectionMode( "sr1", 1 )
```

Dialog control functions

The language provides several built-in functions and operators to be used in a dialog instruction.

Use the dialog functions and operators to keep track of the relative states of the current row, the program array, and the screen array, or to access the field buffers and keystroke buffers.

Typical control functions used in dialogs are: `arr_curr()`, `arr_count()`, `fgl_set_arr_curr()`, `set_count()`, `field_touched()`, `GET_FLDBUF()`, `INFIELD()`, `fgl_dialog_getfieldname()`, `fgl_dialog_getbuffer()`.

As an alternative to functions and operators (especially for those taking hard-coded parameters such as `INFIELD()`, use the methods provided in the ui.Dialog class.

User interruption handling

Allow the end user to cancel the execution of a procedure in the program.

When do we need interruption handling?

If the program executes an interactive instruction, the GUI front end can send action events based on user actions. When the program performs a long process like a loop, a report, or a database query, the front end has no control. You might want to permit the user to stop a long-running process in the such case.

Detecting user interruptions in programs

To detect user interruptions coming from a GUI front-end, you define an action view with the name 'interrupt':

```
BUTTON sb: interrupt, TEXT="Stop";
```

When the runtime system takes control to process program code or execute a long running SQL query, the front end automatically enables the local 'interrupt' action to let the user send an asynchronous interruption request to the program.

A program (i.e. the runtime system) can also receive a SIGINT interruption signal from the operating system. The interruption request that comes from the front-end is a different source, however the runtime system handles both type of interruption events the same way.

When receiving an interrupt event from the front-end with a 'interrupt' special action, or from the system (SIGINT) the runtime system sets the INT_FLAG register to TRUE.

Consider using DEFER INTERRUPT and test the INT_FLAG register to properly handle user interruptions, and avoid immediate program termination: If the DEFER INTERRUPT instruction is not used, the program will stop immediately when an interruption event is caught. With DEFER INTERRUPT, the program continues, and can test INT_FLAG to check if an interruption event occurred. It is good practice to reset INT_FLAG to FALSE after detecting interruption:

```
WHILE ...
   IF INT_FLAG THEN
      LET INT_FLAG=FALSE
      ERROR "Procedure was interrupted by the user"
      EXIT WHILE
   END IF
   ...
END WHILE
```

SQL queries can be interrupted too, if the target database supports this feature. However, since the control is on the database server side while the SQL statement is running, it is not possible to execute program code to check INT_FLAG. In order to detect an SQL interruption, check the SQLCA.SQLCODE register after the query for SQL error -213, indicating that the last SQL statement was interrupted.

```
WHENEVER ERROR CONTINUE
-- Long running SQL statement
WHENEVER ERROR STOP
IF SQLCA.SQLCODE == -213 THEN
   ERROR "Database query interrupted by user"
   ...
END IF
```

When not using DEFER INTERRUPT, if the program enters in a long running procedure, a button with the action name 'interrupt' will become active. The user can then press that button, and the runtime system will stop the program, since DEFER INTERRUPT is not used. However, this will not happen when a dialog is active, because the 'interrupt' button will be automatically disabled in that context. Such situation can confuse the end user, expecting that the 'interrupt' button can stop the program in any context.

Note that the front end can not handle interruption requests properly if the display generates a lot of network traffic. In this case, the front end has to process a lot of user interface modifications and has no time to detect a mouse click on the 'interrupt' action view. A typical example is a program doing a loop from 1 to 10000, just displaying the value of the counter to a field and doing a refresh. This would generate hundreds of AUI tree modifications in a short period of time. In such a case, we recommended that you calculate a modulo and display steps 10 by 10 or 100 by 100.

Implementing interruption of a long running SQL query

```
-- db_busy.per
LAYOUT
 GRID
 {
  Database query in progress...
        [sb            ]
 }
 END
END
ATTRIBUTES
 BUTTON sb: interrupt, TEXT="Stop";
```

```
END
```

```
MAIN
  DEFINE oc INT
  DEFER INTERRUPT
  OPTIONS SQL INTERRUPT ON
  DATABASE stores
  OPEN FORM f FROM "db_busy"
  DISPLAY FORM f
  CALL ui.Interface.refresh()
  WHENEVER ERROR CONTINUE
  SELECT COUNT(*) INTO oc FROM orders
  WHENEVER ERROR STOP
  IF SQLCA.SQLCODE == -213 THEN
    ERROR "Database query has been interrupted..."
  END IF
END MAIN
```

Get program control if user inactivity
Execute some code after a given number of seconds, when the user does not interact with the program.

When to use the ON IDLE trigger?

If an interactive instruction has the control, the program waits for a user interaction like an action or field input. If the end user leaves the workstation, or switches to another application, the program cannot get the control and is frozen until the user comes back. You might want to execute some code, after a perio of inactivity, for example to refresh the displayed data by doing a new database query, or even after a long period, to terminate the program automatically.

Implementing the ON IDLE trigger

To detect user inactivity during a dialog, define an `ON IDLE` trigger in the dialog. This trigger is dialog specific, it is typically defined in the main dialog of the program, but it can also be defined in every dialog.

> **Important:** Consider using the `ON IDLE` interaction block in dialogs that do not handle field input, such as `DISPLAY ARRAY` and `MENU`: In input dialogs, this trigger might be executed when in the middle of a field input, and could force field value validation and raise an input error.

For example:

```
DEFINE seconds SMALLINT
LET seconds = 120
DISPLAY ARRAY ...
   ...
   ON IDLE seconds
      MESSAGE "Automatic data refresh..."
      -- Reload the array with a new database result set
...
```

Note that the parameter of the `ON IDLE` trigger can be a integer variable, but it will only be read when the dialog is started. Changing the variable during dialog execution will have no effect.

A value of zero of less of zero disables the timeout trigger.

Get program control on a regular (timed) basis

Execute some code after a given number of seconds, with or without user interaction with the program.

When to use the ON TIMER trigger?

In some cases, the application needs to execute code on a scheduled basis, for example to process a message arrived in a queue, refresh data on a dashboard, or display resources in time-based graphs.

> **Important:** Unlike the `ON IDLE` trigger which executes when there is no user activity, the `ON TIMER` trigger executes even when the user interacts with the application. Therefore, the code executed in an `ON TIMER` trigger must perform quickly, otherwise the end user will experience poor performance. As a general rule, make sure the time spent in the `ON TIMER` code is less than the timer interval. For example, if the processing time takes about 2 seconds, it doesn't make sense to have an `ON TIMER` that triggers every second.

Implementing the ON TIMER trigger

To return control to the program on regular intervals, use the `ON TIMER` *`seconds`* trigger in dialogs. This trigger is dialog specific. It is typically defined in the main dialog of the program, but it can be defined in every dialog.

> **Important:** Consider using the `ON TIMER` interaction block in dialogs that do not handle field input, such as `DISPLAY ARRAY` and `MENU`. If used in input dialogs, this trigger may execute in the middle of a field input, which can force field value validation and raise an input error.

For example:

```
DEFINE seconds SMALLINT
LET seconds = 120
DISPLAY ARRAY ...
   ...
   ON TIMER seconds
      MESSAGE "Check for messages in queue..."
      -- Query the message server for new messages.
...
```

Note that the parameter of the `ON TIMER` trigger can be a integer variable, but it will only be read when the dialog is started. Changing the variable during dialog execution will have no effect.

A value of zero or less than zero disables the timeout trigger.

Implementing dynamic dialogs

Dialogs can be created at runtime with the `ui.Dialog` class.

Dynamic dialog basics

The `ui.Dialog` class can create dialog objects at runtime, to implement generic code controlling forms that are created at runtime, when the data structure is not known at compile time.

> **Important: Dynamic dialogs are provided to resolve specific needs, like implementing a generic zoom window to select a record in a list, and control forms generated at runtime. This feature is not a replacement for regular "static" dialog instructions, used to control the forms defined in form specification files.**

The dynamic dialogs can be used in conjunction with `base.SqlHandle` objects, to get database table column information in order to build forms dynamically.

Unlike static dialog instructions, dynamic dialogs do not require a data model (i.e. program variables containing the values for fields): Dynamic dialogs hold the data model internally, and behave by default in unbuffered mode: When an action is fired and the corresponding trigger handler is executed, the field values are available.

Creating the form

Before you instanciate a new `ui.Dialog` object, you must load an existing compiled `.42f` form, or create a new form dynamically in your program.

Forms build at runtime must be created with the `ui.Window.createForm()` method, and must contain a valid definition with layout containers, form fields, and screen records.

> **Note:** See Genero BDL demos for a complete example of form creation at runtime.

The `createForm()` method will be invoked by using the current window. For the main form of the program, use directly the (empty) `SCREEN` window. For child windows, create the windows without a form by using following syntax:

```
OPEN WINDOW w1 WITH 1 ROWS, 1 COLUMNS
```

Assuming that there is a current empty window, you can then create the `ui.Form` object, to finally get the `om.DomNode` object to build your form:

```
DEFINE w ui.Window,
       f ui.Form,
       n om.DomNode
LET w = ui.Window.getCurrent()
LET f = w.createForm()
LET n = f.getNode()
...
```

Use `om` classes, to build you form dynamically. A good practice to create dynamic forms is to write first a `.per` file, that implements a static version of one of the forms you want to build at runtime. Compile the `.per` to a `.42f` and inspect the generated XML file, to understand the structure of the form file.

For more details, see:

- ui.Window.createForm on page 1770
- The om package on page 1833

Creating the dialog object

To reference the dialog object, first declare a variable with the type `ui.Dialog`:

```
DEFINE d ui.Dialog
```

The dynamic dialog creation methods take the list of field definitions as parameter, as a dynamic array with a record structure using two members to define the field name and data type.

In order to defined the fields used by the dynamic dialog, define a dynamic array with the following structure:

```
DEFINE fields DYNAMIC ARRAY OF RECORD
           name STRING,
           type STRING
       END RECORD
```

The field definition array will identify form fields and the data types to be used to store the values. The data types are provided as strings, using the same syntax as a regular Genero type:

```
LET fields[1].name = "formonly.cust_id"
LET fields[1].type = "INTEGER"
LET fields[2].name = "formonly.cust_name"
LET fields[2].type = "VARCHAR(50)"
LET fields[3].name = "formonly.cust_modts"
```

```
LET fields[3].type = "DATETIME YEAR TO FRACTION(5)"
```

> **Note:** The type names used by the dynamic dialog API is the same as the type names returned by the `base.SqlHandle.getResultType()` method.

When the list of field definition is complete, create the dynamic dialog object.

To create a dynamic dialog handling simple record input:

```
LET d = ui.Dialog.createInputByName(fields)
```

For more details, see ui.Dialog.createInputByName on page 1789.

To create a display array dynamic dialog:

```
LET d = ui.Dialog.createDisplayArrayTo(fields, "sr_custlist")
```

> **Note:** The list handling, the `createDisplayArrayTo()` method requires the name of the screen record used to group form fields, as defined in the `INSTRUCTIONS` section of the `.per` form file.

For more details, see ui.Dialog.createDisplayArrayTo on page 1791.

To create a dynamic dialog handling query by example:

```
LET d = ui.Dialog.createConstructByName(fields)
```

For more details, see ui.Dialog.createConstructByName on page 1788.

Add user-defined triggers

Dynamic dialogs can be configured with user-defined triggers, for example to execute code when a specific action is fired.

After creating the dialog object, add user-defined triggers with the `ui.Dialog.addTrigger()` method:

```
DEFINE d ui.Dialog
...
CALL d.addTrigger("ON ACTION print")
CALL d.addTrigger("ON DELETE")
...
```

Note that some triggers must be identified with the user-defined action name, as in `"ON ACTION print"`.

User-defined triggers will then be handled in the dynamic dialog loop, when the event occurs.

For more details, see: ui.Dialog.addTrigger on page 1793.

Handling dialog events

To implement the "body" of a dynamic dynamic, mix a `WHILE` loop with the `ui.Dialog.nextEvent()` method, to handle dialog events.

The `WHILE` loop will act as the main event handler of your dynamic dialog, and will loop, waiting for dialog events until you explicitely exist the loop with an `EXIT WHILE` instruction.

```
DEFINE d ui.Dialog
...
WHILE TRUE
    CASE d.nextEvent()
       WHEN "BEFORE DISPLAY"
          ...
       WHEN "ON ACTION print"
          ...
```

```
        WHEN "ON DELETE"
           ...
        WHEN "AFTER DISPLAY"
           ...
END WHILE
```

Several implicit trigger names are supported by dynamic dialogs, such as `"BEFORE ROW"`, `"AFTER FIELD` *`field-name`*`"`. These triggers are equivalent to the static dialog control blocks, to control the behavior of your dynamic dialog.

The event handlers for the user-defined triggers that have been added with the `addTrigger()` method must also be handled in the dynamic dialog loop.

Inside the `WHILE` loop, control the behavior of the dialog with the methods provided in the `ui.Dialog` class. For example, to jump to a different field when the "jump" action is fired:

```
WHILE TRUE
    CASE d.nextEvent()
       WHEN "ON ACTION jump"
          CALL d.nextField("customer.cust_name")
    ...
```

BEFORE/AFTER FIELD handlers must be identified with the field name (without the table/formonly prefix):

```
WHILE TRUE
    CASE d.nextEvent()
       WHEN "AFTER FIELD cust_name"
          IF LENGTH(d.getFieldValue("customer.cust_name")) < 3 THEN
             ERROR "Customer name is too short"
             CALL d.nextField("customer.cust_name")
          END IF
    ...
```

For more details, see the ui.Dialog.nextEvent() method reference.

Handling field values

A dynamic dialog stores field values in internal buffers created according to the field definitions provided in the creation method. Access to these values is required, to implement the dynamic dialog. For example, to set default values before entering the dialog loop, modifying and/or querying values during the dialog loop, and to get the entered values after dialog termination when accepted by the user.

To set or get values of fields controlled by a dynamic dialog, use respectively the `ui.Dialog.setFieldValue()` and `ui.Dialog.getFieldValue()` methods.

> **Note:** These methods take a form field name as parameter, that can be provided in different notations. See Identifying fields in dialog methods on page 1818 for more details.

When implementing a display array dynamic dialog handling a record list, the set/get field value methods apply to the current row: If you want to set or get field values of a particular row, first move to the row with the `ui.Dialog.setCurrentRow()` method.

The next example copies the values from the fields in the current row of a display array dynamic dialog (`d_list`), to the field buffers of a record input dynamic dialog (`d_rec`):

```
CALL d_list.setCurrentRow("sr_custlist", index)
FOR i=1 TO fields.getLength()
    CALL d_rec.setFieldValue( fields[i].name,
                d_list.getFieldValue(fields[i].name)
          )
END FOR
```

For more details, see:

- ui.Dialog.setFieldValue on page 1813
- ui.Dialog.getFieldValue on page 1801

Get query conditions for a field

A dynamic dialog created with ui.Dialog.createConstructByName on page 1788 handles query by example input.

To generate the SQL condition from the search value entered in a construct field, use the ui.Dialog.getQueryFromField on page 1802 method, by passing the field name as parameter:

```
LET field_condition = DIALOG.getQueryFromField("customer.cust_name")
```

To build the complete `WHERE` part for the `SELECT` statement, iter through all form fields and concatenate the form field condition by separating with the `AND` or with the `OR` operator:

```
FOR i=1 TO fields.getLength()
    LET field_condition = d.getQueryFromField(fields[i].name)
    IF field_condition IS NOT NULL THEN
       IF where_clause IS NOT NULL THEN
          LET where_clause = where_clause, " AND "
       END IF
       LET where_clause = where_clause, field_condition
    END IF
END FOR
```

Implementing the accept and cancel actions

Regular static dialog instructions implement the accept and cancel actions, to respectively validate or abort the dialog.

These actions are created automatically for static dialogs, but must be created by hand for dynamic dialogs.

In the case of cancel, you can mimic the behavior of static dialogs by setting the `INT_FLAG` register to `TRUE` and then leave the `WHILE` loop with an `EXIT WHILE`.

For the accept action, call the `ui.Dialog.accept()` method to validate field input and leave the dialog, and execute an `EXIT WHILE` in the `"AFTER INPUT"` event to leave the dialog loop.

For example, to implement the accept and cancel actions for a simple record input:

```
DEFINE d ui.Dialog
...
LET d = ui.Dialog.createInputByName(fields)
CALL d.addTrigger("ON ACTION cancel")
CALL d.addTrigger("ON ACTION accept")
...
WHILE TRUE
  CASE d.nextEvent()
    WHEN "ON ACTION cancel"
      LET int_flag = TRUE
      EXIT WHILE
    WHEN "ON ACTION accept"
      CALL d.accept()
    WHEN "AFTER INPUT"
      EXIT WHILE
  END CASE
END WHILE
```

Terminating the dialog

Some synchronization code needs to be implemented to properly destroy the dynamic dialog. A dialog needs to be destroyed by closing its corresponding window/form.

In order to terminate a dialog, assign `NULL` to the `ui.Dialog` variable referencing the dialog object. This will destroy the object, if no other variables references it, and the corresponding window can then be closed:

```
LET d = NULL
CLOSE WINDOW w1
```

Combining dynamic dialogs with dynamic cursors

To write generic code accessing a database, implement the dynamic dialog with field names and types coming from the `base.SqlHandle` cursor.

The next code example builds a list of fields according to the database table passed as first parameter. The function scans the result set column names and types of the `base.SqlHandle` cursor, to build the list of field definitions, that can then be used for the dynamic dialog creation:

```
FUNCTION build_field_list(dbtable, fields)
    DEFINE dbtable STRING,
           fields DYNAMIC ARRAY OF RECORD
               name STRING,
               type STRING
           END RECORD
    DEFINE h base.SqlHandle,
           i INT

    LET h = base.SqlHandle.create()
    CALL h.prepare("SELECT * FROM " || dbtable)
    CALL h.open()
    CALL h.fetch()
    CALL fields.clear()
    FOR i=1 TO h.getResultCount()
        LET fields[i].name = h.getResultName(i)
        LET fields[i].type = h.getResultType(i)
    END FOR

END FUNCTION
```

For more details, see The SqlHandle class on page 1725.

Input fields

Describes various concepts related to form field management in dialogs

Field input length

Field input length defines the amount of characters the user can type in a form field.

Input length basics

The *field input length* is used by interactive instructions to limit the size of the data that can be entered by the user. Additionally, when displaying a program variable to a form field with the `DISPLAY TO` or `DISPLAY BY NAME` instruction, the field input length is used to truncate the text resulting from the data conversion. For non-character values, if the resulting text does not fit into the input length, the field will show * stars to indicate an overflow.

Length semantics for character fields

When using byte length semantics (the default), the input length represents the number of bytes in the current character set. In other words, it is the number of bytes used by the character string in the character set used by the runtime system. For example, when using a Chinese BIG5 encoding, Latin characters (a,b,c) use one byte each, while Chinese ideograms use 2 bytes: If the input length is 6, the user can enter 6 Latin characters like "abcdef", or 4 Latin characters and one Chinese ideogram, or 3 Chinese ideograms.

When using character length semantics (FGL_LENGTH_SEMANTICS=CHAR environment variable), the unit for the input length is in characters. For example, in a UTF-8 character set, if the form field has a width of 6 cells, the field can hold 6 characters, from any alphabet. There is no limitation regarding the number of bytes the UTF-8 encoded string will use.

Input length control

The field input length is defined according to:

1. The type of layout (grid-based or stack-based layout)
2. The data type of the program variable bound to field by the interactive instruction.
3. In grid-based layout, the usage of the `SCROLL` attribute for `CHAR`/`VARCHAR`/`STRING` types.

Field width definition in grid-based containers

In a grid-based container, by default the input length is defined by the width of the field item tag in the `LAYOUT` section. The width of a field item tag is defined by the number of cell positions used between the square braces:

```
LAYOUT
GRID
{
  [f1 ]    -- width = 3 cells
  [f2    ] -- width = 6 cells
  ...
```

As a general rule, forms must define fields that can hold all possible values that the corresponding program variable can contain. For example, a `DATE` field must be defined with 10 cells, to hold date values in the format DD/MM/YYYY.

If the program variable is defined with a numeric data type like `INTEGER` or `DECIMAL`, the input length is defined by the width of the field defined in the form.

If the program variable is defined with character data type such as `CHAR`, `VARCHAR` or `STRING`, by default, the input length is defined by the width of the field defined in the form. The `SCROLL` attribute can be used to bypass this limit and force the input length to be as large as the program variable. For example, when using a `CHAR(20)` variable with a form field defined with width of 3 characters, the input length will be 20 characters instead of 3.

> **Note:** Using the `SCROLL` attribute must be an exception: Form fields should be large enough to hold all possible characters that fit in the corresponding program variable. Note also that for specific item types like `TEXTEDIT`, the `SCROLL` attribute behavior is implicit when the element is stretchable or allows scrollbars.

If the program variable is defined with a `DATE`, `DATETIME` or `INTERVAL` data type, the input length is defined by the data type. For example. a `DATE` field will allow 10 characters.

Field width definition in stack-based containers

In a stack-based layout, the input length is defined by the data type of the program variable.

In the next example, the `cust_id` field will allow numeric input length in the range of the `INTEGER` data type, and the `cust_name` field will allow up to 50 characters:

```
-- Form file
LAYOUT
 STACK
   EDIT customer.cust_id;
   EDIT customer.cust_name;
 ...

-- Program
MAIN
   DEFINE cust_rec RECORD
              cust_id INTEGER,
              cust_name VARCHAR(50)
          END RECORD
   ...
   INPUT BY NAME cust_rec.*
      ...
```

If the program variable is defined with a numeric data type like `INTEGER` or `DECIMAL` or a character data type such as `CHAR`, `VARCHAR` or `STRING`, the input length is defined by the value range of the program variable. For numeric values, you can use the `INCLUDE` attribute to define the range of possible values.

If the program variable is defined with a `DATE`, `DATETIME` or `INTERVAL` data type, the input length is defined by the data type. For example. a `DATE` field will allow 10 characters.

The buffered and unbuffered modes
The buffered and unbuffered mode control the synchronization of program variables and form fields.

Data model / view / controller paradigm

When bound to an interactive instruction (i.e. dialog), program variables act as a data model to display data or to get user input. To change the values of form fields by program, the corresponding variables must be set and displayed.

Synchronization of program variables with the form fields depends on the buffer mode used by the dialog. Use the unbuffered mode to get automatic data model / form field synchronization.

Configuring the buffer mode

By default, singular dialogs (`INPUT`, `DISPLAY ARRAY`) and procedural `DIALOG` blocks are using the buffered mode, while parallel dialogs are using the unbuffered mode by default.

The unbuffered mode can be set per (modal) dialog instruction, with the `UNBUFFERED` dialog attribute:

```
INPUT BY NAME p_site.* ATTRIBUTES(UNBUFFERED)
   ...
END INPUT
```

When using a procedural DIALOG block, all subdialogs defined locally or included with the SUBDIALOG clause inherit the buffer mode of the parent procedural dialog block:

```
DIALOG ATTRIBUTES(UNBUFFERED)
  INPUT BY NAME p_site.* -- unbuffered
     ...
  END INPUT
  DISPLAY ARRAY a_events TO sr_events.* -- unbuffered
     ...
  END DISPLAY
  SUBDIALOG d_comments -- unbuffered
```

```
END DIALOG
```

The unbuffered mode can also be set globally with the ui.Dialog.setDefaultUnbuffered() method, for singular and procedural dialogs:

```
CALL ui.Dialog.setDefaultUnbuffered(TRUE)
...
INPUT BY NAME rec_cust.* WITHOUT DEFAULTS -- uses unbuffered mode
   ...
END INPUT
```

In contrast with modal dialogs described above, when implementing parallel dialogs, all started dialogs are implicitly using the unbuffered mode, and it is not possible to use the buffered mode:

```
DIALOG d_customers()
  INPUT BY NAME r_cust.*
     ...
  END INPUT
END DIALOG
...
START DIALOG d_customers  -- will be unbufferd by default
...
```

The buffered mode

When you use the default "buffered" mode, program variable changes are not automatically displayed to form fields; you need to execute `DISPLAY TO` or `DISPLAY BY NAME`. Additionally, if an action is triggered, the value of the current field is not validated and is not copied into the corresponding program variable. The only way to get the text of a field is to use GET_FLDBUF() or DIALOG.getFieldBuffer(). These functions return the current text, which might not be a valid representation of a value of the field data type:

```
INPUT BY NAME p_item.*
   ON ACTION zoom
      CALL select_item()
           RETURNING p_item.code, p_item.desc
         DISPLAY BY NAME p_item.code, p_item.desc
      END IF
   ...
END INPUT
```

The unbuffered mode

With the unbuffered mode, program variables and form fields are automatically synchronized, and the dialog instruction is sensitive to program variable changes: You don't need to display values explicitly with `DISPLAY TO` or `DISPLAY BY NAME`. When an action is triggered, the value of the current field is validated and is copied into the corresponding program variable. If you need to display new data during the dialog execution, just assign the values to the program variables; the runtime system will automatically display the values to the screen after user code of the current control or interaction block has been executed:

```
INPUT BY NAME p_site.* ATTRIBUTES(UNBUFFERED)
   ON ACTION zoom
      CALL select_item()
           RETURNING p_item.code, p_item.desc
         -- no need to display desc.
      END IF
   ...
END INPUT
```

Actions configuration for field validation

During data input, values entered by the user in form fields are automatically validated and copied into the program variables. Actually the value entered in form fields is first available in the form field buffer. This buffer can be queried with built-in functions or dialog class methods. With the unbuffered mode, the field buffer is used to synchronize program variables each time control returns to the runtime system - for example, when the user clicks on a button to execute an action.

With the unbuffered mode, data validation must be prevented for some actions such as cancel or close . To avoid field validation for a given action, set the `validate` action default attribute to "no", in the `.4ad` file or in the `ACTION DEFAULTS` section of the form file:

```
ACTION DEFAULTS
   ACTION undo (TEXT = "Undo", VALIDATE = NO)
   ...
END
```

Some predefined actions are already configured with `validate=no` in the `default.4ad` file.

If field validation is disabled for an action, the code executed in the `ON ACTION` block acts as if the dialog was in buffered mode: The program variable is not set; however, the input buffer of the current field is updated. When returning from the user code, the dialog will not synchronize the form fields with program variables, and the current field will display the input buffer content. Therefore, if you change the value of the program variable during an `ON ACTION` block where validation is disabled, you must explicitly display the values to the fields with `DISPLAY TO / BY NAME`.

To illustrate this case, imagine that you want to implement an undo action to allow the modifications done by the user to be reverted (before these have been saved to the database of course). You typically copy the current record into a clone variable when the dialog starts, and copy these old values back to the input record when the undo action is invoked. An undo action is a good candidate to avoid field validation, since you want to ignore current values. If you don't re-display the values, the input buffer of the current field will remain when returning from the `ON ACTION` block:

```
DIALOG ATTRIBUTES(UNBUFFERED)
 INPUT BY NAME p_cust.*
   BEFORE INPUT
     LET p_cust_copy.* = p_cust.*
   ON ACTION undo -- Defined with VALIDATE=NO
     LET p_cust.* = p_cust_copy.*
     DISPLAY BY NAME p_cust.*
 END INPUT
END DIALOG
```

For more details, see Data validation at action invocation on page 1331.

Binding variables to form fields

Some dialogs need program variables to store form field values.

Dialogs handling data fields input or display (`INPUT`, `INPUT ARRAY`, `DISPLAY ARRAY`) need program variables to store the information displayed in form fields during the dialog execution. The exception is `CONSTRUCT`, which needs only one string variable that holds the SQL condition produced.

When declaring a dialog handling form fields, you specify what program variables must be bound to the form fields:

```
   INPUT BY NAME custrec.* ...
      ...
   END INPUT
```

There are different ways to bind program variables to screen record fields. Basically program variables can be bound to form fields by name or by position, according to the binding clause used in the dialog definition.

When binding program variables with a screen record followed by a `.*` (dotstar), program variables are bound to screen record fields by position, so you must make sure that the program variables are defined (or listed) in the same order as the screen array fields. This is true for `INPUT`, `DISPLAY ARRAY` and `INPUT ARRAY`.

The program variables can be of any simple data type supported by the dialogs; the runtime system will adapt input and display rules to the variable type. When the user enters data for an `INPUT` or `INPUT ARRAY` instruction, the runtime system checks the entered value against the data type of the variable, not the data type of the form field. For example, if you want to use a `DATE` variable, the dialog will check for a valid date value when the user enters a value in the corresponding form field.

With `CONSTRUCT`, no program variable is used for fields: Only one string variable is bound to that type of dialog, to hold the generated SQL condition. Note that the `CONSTRUCT` dialog uses the field data types defined in the form file.

Program variables are typically declared with a `DEFINE LIKE` clause to get the data type of a column as defined in the database schema file. When the form fields are also defined like a column of the database schema, this ensure that the program variable and form field data type matches the underlying database column type. If a variable is declared `LIKE` a SERIAL / SERIAL8 / BIGSERIAL column, the runtime system will treat the field as if it was defined as `NOENTRY` in the form file: Since values of serial columns are automatically generated by the database server, no user input is required for such fields.

Program variables (simple records and arrays) used in dialogs can have a flat definition, or structured definition with sub-records.

Data format for input and display of numeric (`DECIMAL`, `INTEGER`) and `DATE` fields can be defined with the `FORMAT` attribute. A default data format can be defined with environment variables (DBDATE, DBFORMAT, etc)

Some data validation rules can be defined at the form level, such as `NOT NULL`, `REQUIRED` and `INCLUDE` attributes. Data validation constraints are checked when leaving a field, or when the dialog is validated (for example, with the `ACCEPT DIALOG` instruction inside a `DIALOG` multiple dialog block).

If the program record or array has the same structure as a database table (this is the case when the variable is defined with a `DEFINE LIKE` clause), you may not want to display / use some of the columns. You can achieve this by used `PHANTOM` fields in the screen array definition. Phantom fields will only be used to bind program variables, and will not be transmitted to the front-end for display.

Form field initialization

Form field initialization can be controlled by the `WITHOUT DEFAULTS` dialog option.

The `INPUT` and `INPUT ARRAY` dialogs provide the `WITHOUT DEFAULTS` option to use program variable values when the dialog starts, or to apply the `DEFAULT` attribute defined in forms. The semantics of this options is slightly different in `INPUT` and `INPUT ARRAY` dialogs. The `WITHOUT DEFAULTS` clause should always be used in `INPUT ARRAY`.

The `WITHOUT DEFAULTS` option can be used in the binding clause or as an `ATTRIBUTES`. When used in the binding clause, the option is defined statically at compile time as `TRUE`. When used as an `ATTRIBUTES` option, it can be specified with an integer expression that is evaluated when the `DIALOG` interactive instruction starts:

```
INPUT BY NAME p_cust.* ATTRIBUTES (WITHOUT DEFAULTS = NOT new)
   ...
END INPUT
```

The WITHOUT DEFAULTS clause in INPUT

In the default mode, an INPUT clears the program variables and assigns the values defined by the DEFAULT attribute in the form file (or indirectly, the default value defined in the database schema files). This mode is typically used to input and INSERT a new record in the database. The REQUIRED field attributes are checked to make sure that the user has entered all data that is mandatory. Note that REQUIRED only forces the user to enter the field, and can leave the value NULL unless the NOT NULL attribute is used. Therefore, if you have an AFTER FIELD or ON CHANGE control block with validation rules, you can use the REQUIRED attribute to force the user to enter the field and trigger that block.

In contrast, the WITHOUT DEFAULTS option starts the INPUT dialog with the existing values of program variables. This mode is typically used in order to UPDATE an existing database row. Existing values are considered valid, thus the REQUIRED attributes are ignored when this option is used.

The NOT NULL field attribute is always checked at dialog validation, even if the WITHOUT DEFAULTS option is set.

The WITHOUT DEFAULTS clause in INPUT ARRAY

With an INPUT ARRAY, the WITHOUT DEFAULT option defines whether the program array is populated when the dialog begins. Once the dialog is started, existing rows are always handled as records to be updated in the database (i.e. WITHOUT DEFAULTS=TRUE), while newly created rows are handled as records to be inserted in the database (i.e. WITHOUT DEFAULTS=FALSE). In other words, column default values defined in the form specification file or the database schema files are only used for new created rows.

It is unusual to implement an INPUT ARRAY with no WITHOUT DEFAULTS option, because the program array would be cleared and the list would appear empty.

> **Important:** The default in INPUT ARRAY used inside DIALOG is WITHOUT DEFAULTS=TRUE, but in a singular INPUT ARRAY dialog, the default is WITHOUT DEFAULTS=FALSE.

Input field modification flag

Each input field controlled by a dialog instruction has a modification flag.

The modification flag is used to execute form-level validation rules and trigger ON CHANGE blocks. The flag can also be queried to detect if a field was touched/changed during the DIALOG instruction, for example with the FIELD_TOUCHED() operator or with ui.Dialog.getFieldTouched().

Both FIELD_TOUCHED() and ui.Dialog.getFieldTouched() accept a list of fields and/or the *screen-record.** notation in order to check the modification flag of multiple fields in a unique function call. You can also pass a simple * star as parameter, to reference all fields used by the dialog.

The modification flag is set to TRUE when the user enters data in a field, or when the program executes a DISPLAY TO / DISPLAY BY NAME instruction. The flag can also be set by program to TRUE or reset to FALSE with the ui.Dialog.setFieldTouched() method, to emulate user input by program or to reset the modification flags after data was saved in the database.

The modification flags of all fields are automatically reset to FALSE by the interactive instruction in the following cases:

- When the dialog instruction starts.
- In a DIALOG block, when entering a group of fields controlled by an INPUT or a CONSTRUCT sub-dialog .
- When moving to (or creating) a new row in an INPUT ARRAY.
- Withing a DISPLAY ARRAY, the modification flags are always TRUE for all fields.

When using a DISPLAY ARRAY, the modification flags are set to TRUE for all fields. This behavior exists because of backward compatibility. Since values cannot be modified by the user, the modification flags are not relevant in this dialog. However, you must pay attention when implementing nested dialogs, because DISPLAY ARRAY will set the modification flags of the fields driven by the parent dialog, for example when executing a DISPLAY ARRAY from an INPUT ARRAY.

Query the modification flags with the `ui.Dialog.getFieldTouched()` method, typically in the context of `AFTER INPUT`, `AFTER CONSTRUCT`, `AFTER INSERT` or `AFTER ROW` control blocks.

When using a list driven by an `INPUT ARRAY` binding, a temporary row added at the end of the list will be automatically removed if all fields have the modification flag is set to `FALSE`.

For typical `EDIT` fields, the modification flag is set when leaving the field. If you want to detect data modification earlier, you should use the dialogtouched predefined action. However, this event is only an indicator that the user started to modify a field, the value will not be available in the program variables.

Reacting to field value changes
This section describes the purpose of the `ON CHANGE` interaction block.

The `ON CHANGE` interaction block can be used in different ways:

- With form fields allowing only entire value input such as `CHECKBOX`, or using an additional widget such as a calendar in a `DATEEDIT`: `ON CHANGE` can be used to detect an immediate value change, or the selection of a value in the additional widget, without leaving the field.
- With text fields like `EDIT` (allowing incomplete values), defined with the `COMPLETER` attribute to implement autocompletion: In this case the `ON CHANGE` trigger is fired without leaving the field, when the user types characters in (after a short delay).
- With text fields like `EDIT` (allowing incomplete values): `ON CHANGE` can be used to detect a value change, when the field is left.

A typically usage of `ON CHANGE` is for example with a `CHECKBOX`, to enable/disable other form elements according to the value of the checkbox field:

```
INPUT BY NAME rec.* ...
    ...
    ON CHANGE input_details -- can be TRUE or FALSE
       CALL DIALOG.setFieldActive("address1", rec.input_details)
       CALL DIALOG.setFieldActive("address2", rec.input_details)
    ...
END INPUT
```

The `dialogtouched` predefined action can also be used to detect field changes immediately, but with this action you can't get the data in the target variables; this special action should only be used to detect that the user has started to modify data in the current dialog.

Immediate detection of user changes
This section describes the purpose of the predefined `dialogtouched` action.

The *dialogtouched* special predefined action can be used to detect user changes immediately and execute code in the program.

Singular interactive instruction are typically ended with an accept or cancel action. For example, a singular `INPUT` statement allows the end user to enter a database record, and validate or cancel the input for that record. The `INPUT` statement is then re-executed to input another record. Unlike singular dialogs, the `DIALOG` instruction can be used continuously for several data operations, such as navigation, creation, or modification. Typically, default is the navigation mode, and as soon as the user starts to modify a field, it switches to edit mode, to modify a record, or create a new record. In such case, the dialog must be notified when the user starts to modify the current record, for example to enable a save action. This is achieved with the `dialogtouched` predefined action.

The dialogtouched action works for any field controlled by the current interactive instruction, and with any type of form field: Every time the user modifies the value of a field (without leaving the field), the `ON ACTION dialogtouched` block will be executed; This can be triggered by typing characters in a text editor field, clicking a checkbox / radiogroup, or modifying a slider. When a `ON ACTION dialogtouched` action handler is defined, the front-end knows that it must send this action when the end-user modifies the current field (without leaving that field), just by a simple keystroke.

> **Important: The dialogtouched action must be enabled/disabled in accordance with the status of the dialog: If this action is enabled, the `ON ACTION dialogtouched` block will be invoked each time the user types characters (or modifies the value with copy/paste) in the current field; This can generate a lot of network traffic and is not the goal of this action: The dialogtouched action must be disabled as soon as it is detected, and the DIALOG can then enter in modification/edit mode. When user input is validated and saved in the database, the dialogtouched action can be enabled again.**

Use `ON ACTION dialogtouched` to detect the beginning of a record modification in a `DIALOG` block, to enable a "save" action for example. To prevent further dialogtouched action events, disable the action with a `DIALOG.setActionActive()` method. When the dialogtouched action is enabled, the `ON ACTION` block will be invoked each time the user types characters in an editable field. This programming pattern is illustrated by the next code example:

```
DIALOG
   ...
   ON ACTION dialogtouched
      CALL setup_dialog(DIALOG,TRUE)
   ...
   ON ACTION save
      CALL save_record()
      CALL setup_dialog(DIALOG,FALSE)
   ...
END DIALOG

FUNCTION setup_dialog(d,editing)
   DEFINE d ui.Dialog, editing BOOLEAN
    CALL DIALOG.setActionActive("dialogtouched", NOT editing)
    CALL DIALOG.setActionActive("save", editing)
    CALL DIALOG.setActionActive("query", NOT editing)
END FUNCTION
```

When a dialogtouched action occurs, the current field may contain some text that does not represent a valid value of the underlying field data type. For example, a form field bound to a `DATE` variable may contain only a part of a valid date string, such as `[12/24/ ]`. For this reason, the target variable cannot hold the current text displayed on the screen when the `ON ACTION dialogtouched` code is executed, even when using the `UNBUFFERED` mode.

To avoid data validation on action code execution, the dialogtouched action is defined with `validate="no"` attribute in the `FGLDIR/lib/default.4ad` action defaults file. This is mandatory when using the `UNBUFFERED` mode; otherwise the runtime would try to copy the input buffer into the program variable when a dialogtouched action is invoked. Since the text of the current field will in most cases contain only a part of a valid data value, using `validate="yes"` would always result in a conversion error.

In order to detect field input changes, you can use the `ON CHANGE` trigger, when the form item type allows to detect value changes immediately, for example in `COMBOBOX`, `CHECKBOX` or `DATEEDIT` fields.

Form-level validation rules

Form-level validation rules can be defined for each field controlled by a dialog.

Form-level validation can be specified at the form field level with attributes such as `NOT NULL`, `REQUIRED` and `INCLUDE`. These attributes are part of the business rules of the application and must be checked before saving data into the database.

Implicit validation rule checking

An `INPUT` or `INPUT ARRAY` block automatically executes form-level validation rules in the following cases:

- The `NOT NULL` attribute is satisfied if a value is in the field. `NOT NULL` is checked:

    - when the user moves to a different row in a list controlled by an `INPUT ARRAY`; However, if the row is temporary and none of the fields is touched, the attribute is ignored.
    - in a `DIALOG` block, when focus leaves the sub-dialog controlling the field;
    - in a `DIALOG` block, when `NEXT FIELD` gives the focus to a field in a different sub-dialog than the current sub-dialog.
    - when the dialog instruction is ended, for example when a procedural `DIALOG` is ended with `ACCEPT DIALOG`, or when an singular `INPUT` is ended with `ACCEPT INPUT` or with the implicit accept action.
- The `REQUIRED` attribute is satisfied if the field modification flag is true, if a `DEFAULT` value is defined, or if the `WITHOUT DEFAULTS` option is used. `REQUIRED` is checked:
    - when the user moves to a different row in a list controlled by an `INPUT ARRAY`; However, if the row is temporary and none of the fields is touched, the attribute is ignored.
    - in a `DIALOG` block, when focus leaves the sub-dialog controlling the field;
    - in a `DIALOG` block, when `NEXT FIELD` gives the focus to a field in a different sub-dialog than the current sub-dialog.
    - when the dialog instruction is ended, for example when a procedural `DIALOG` is ended with `ACCEPT DIALOG`, or when an singular `INPUT` is ended with `ACCEPT INPUT` or with the implicit accept action.
- The `INCLUDE` attribute is satisfied if the value is in the list defined by the attribute. `INCLUDE` is checked when the target program variable must be assigned. This happens:
    - when `UNBUFFERED` mode is used, focus is in the field, and an action is invoked;
    - when the focus leaves the field;
    - when the user moves to a different row in a list controlled by an `INPUT ARRAY`; However, if the row is temporary and none of the fields is touched, the attribute is ignored.
    - in a `DIALOG` block, when focus leaves the sub-dialog controlling the field;
    - in a `DIALOG` block, when `NEXT FIELD` gives the focus to a field in a different sub-dialog than the current sub-dialog.
    - when the dialog instruction is ended, for example when a procedural `DIALOG` is ended with `ACCEPT DIALOG`, or when an singular `INPUT` is ended with `ACCEPT INPUT` or with the implicit accept action.

Performing validation rules explicitly

Singular input dialogs (`INPUT` / `INPUT ARRAY`) create default accept / cancel actions. The form-level validation rules are typically performed when the implicit accept action is triggered.

The `DIALOG` procedural instruction can be used as in singular interactive instructions, with the typical OK / Cancel buttons (i.e. accept / cancel actions) to finish the instruction. The accept/cancel action handlers would respectively execute the `ACCEPT DIALOG` and `EXIT DIALOG` instructions. This solution lets the user input or modify one record at a time, and the program flow must reenter the`DIALOG` instruction to edit or create another record. Alternatively, the`DIALOG` instruction can let the user input / modify multiple records without leaving the dialog. In this case, you need a way to execute the form-level validation rules defined for each field, before saving the data to the database.

To validate a subset of fields controlled by the `DIALOG` instruction, use the `ui.Dialog.validate("field-list")` method, as shown in this example:

```
   ON ACTION save
      IF DIALOG.validate("cust.*") < 0 THEN
         CONTINUE DIALOG
      END IF
      CALL customer_save()
```

This method automatically displays an error message and registers the next field in case of error. It is mandatory to execute a `CONTINUE DIALOG` instruction if the function returns an error.

Within singular input dialogs, form-level validation rules can also be explicitly performed with the ACCEPT INPUT instruction, or with the DIALOG.validate("*") API call, followed by a CONTINUE INPUT in case of error.

Form field deactivation

The form fields bound to a dialog are by default active (i.e. they can get the focus). When needed, disable the fields that do not require user input, and reactivate them later during the dialog execution. For example, imagine a form containing an "Industry" COMBOBOX field , with the options *Healthcare*, *Education*, *Government*, *Manufacturing*, and *Other* . If the user selects "Other", an secondary EDIT field should be activated automatically, to let the user input the specific description of the industry. But if one of the predefined values is selected, there is no need for the additional field, so secondary field can be left disabled.

This can be achieved by enabling/disabling fields with the ui.Dialog.setFieldActive() method according to the context. The "Industry" field case described can be implemented as follows:

```
DIALOG ATTRIBUTES(UNBUFFERED)
   INPUT BY NAME rec.*
      ON CHANGE industry
         -- A value of 99 corresponds to the "Other" item
         CALL DIALOG.setFieldActive( "cust.industry", (rec.industry!=99) )
     ...
   END INPUT
   BEFORE DIALOG
     CALL DIALOG.setFieldActive( "cust.industry", FALSE )
     ...
END DIALOG
```

Consider centralizing field activation / deactivation in a setup function specific to the dialog, passing the DIALOG object as parameter.

Do not disable all fields of a dialog, otherwise the dialog execution stops (at least one field must get the focus during a dialog execution).

It is also possible to hide fields with the ui.Form.setFieldHidden() method of the form objects. The dialog considers hidden fields as disabled (i.e. there is no need to disable fields that are already hidden). But hiding form elements changes the space used in the window layout and the form may be displayed in unexpected way, except when hiding elements in containers prepared to that, such as tables.

Identifying sub-dialogs in procedural DIALOG

Sub-dialogs need to be identified by a name to distinguish the different contexts.

A procedural DIALOG block is a collection of sub-dialogs that act as controllers for different parts of a form. In order to program a procedural DIALOG block, there must be a unique identifier for each sub-dialog.

For example, to set the current row of a screen array with the DIALOG.setCurrentRow() method, you pass the name of the screen array to specify the sub-dialog to be affected. Sub-dialog identifiers are also used as a prefix to specify actions for the sub-dialog.

The following topics describe how to specify the names of the different types of DIALOG sub-dialogs:

- Identifying an INPUT sub-dialog on page 1154
- Identifying a DISPLAY ARRAY sub-dialog on page 1156
- Identifying an INPUT ARRAY sub-dialog on page 1157
- Identifying a CONSTRUCT sub-dialog on page 1155
- The SUBDIALOG clause on page 1158.

Defining the tabbing order

Control the order of tabbing trough the fields with the TABINDEX attribute.

When a dialog is executing, the end-user can jump from field to field with the keyboard by using the Tab and Shift-Tab keys.

> **Note:** One can tab out of an INPUT ARRAY sub-dialog with Ctrl-Tab and Shift-Ctrl-Tab accelerators (in INPUT ARRAY, Tab and Shift-Tab loop in the fields of the current row).

The order in which the fields can be visited with the Tab key can be controlled with a program option and the TABINDEX form field attribute.

The FIELD ORDER dialog attribute defines the way tabbing order works. Tabbing order can be based on the dialog binding list (FIELD ORDER CONSTRAINED, the default) or it can be based on the form tabbing order (FIELD ORDER FORM). It is recommended that you use the FIELD ORDER FORM option, to use the tabbing order specified in the form file.

The TABINDEX field attribute allows tabbing order in the form to be defined for each form item. By default, the form compiler assigns a tabbing index for each form item according to the position of the item in the layout.

Form elements that can get the focus are:

- Simple form fields controlled by INPUT or CONSTRUCT,
- Read-only lists controlled by DISPLAY ARRAY,
- Editable list cells controlled by INPUT ARRAY,
- Simple buttons controlled by a COMMAND interaction block.

If you use the keyboard to tab into a form element, the focus will go to the next (or previous) element that is visible and activated . In other words, if a form item is hidden or disabled, it is removed from the tabbing list.

The tabbing position of a read-only list driven by a DISPLAY ARRAY binding is defined by the TABINDEX of the first field .

When TABINDEX is set to zero, the form item is excluded from the tabbing list. However, the item with TABINDEX=0 can still get the focus with the mouse (or when you tap on it on a mobile device).

The NEXT FIELD instruction can also use the tabbing order, when executing NEXT FIELD NEXT and NEXT FIELD PREVIOUS.

If the form uses a TABLE container, the front-end resets the tab indexes when the user moves columns around. This way, the visual column order always corresponds to the input tabbing order. If the order of the columns in an editable list shouldn't be changed, you can freeze the table columns with the UNMOVABLECOLUMNS attribute.

Which form item has the focus?

Identify what element of the current form has the focus.

Sometimes it is important to know what form element has currently the focus. This is especially important when implementing a DIALOG block, that can control several parts of a form. For example, when several lists are driven by multiple DISPLAY ARRAY sub-dialogs, you may need to know what is the current list.

To get the name of the current form item, use the DIALOG.getCurrentItem() method. This method is the replacement of the former fgl_dialog_getfieldname() built-in function. It has been extended to return identifiers for fields, lists or actions identifiers.

```
DIALOG ATTRIBUTES(UNBUFFERED)
   DISPLAY ARRAY p_orders TO orders.*
     ...
   END DISPLAY
   DISPLAY ARRAY p_items TO items.*
     ...
   END DISPLAY
```

```
   ...
      IF DIALOG.getCurrentItem() == "items" THEN
         ...
      END IF
   ...
END DIALOG
```

It is also possible to detect when the focus enters or leaves a field or a group of fields by using control blocks such as `BEFORE INPUT/DISPLAY` or `AFTER INPUT/DISPLAY`.

Giving the focus to a form element
How to force the focus by program, to move or stay in a specific form element.

Use the `NEXT FIELD` instruction to force the focus to a specific field or screen record (list). The `NEXT FIELD` instruction expects a form field name.

In a `DIALOG` block, when the specified field is the first column identifier of a sub-dialog driven by a `DISPLAY ARRAY` block, the read-only list gets the focus. If the field name is not known at compile time, you can alternatively use the `ui.Dialog.nextfield()` method.

```
DIALOG ATTRIBUTES(UNBUFFERED)
   INPUT BY NAME p_cust ATTRIBUTES(NAME="cust")
     ...
   END DISPLAY
   DISPLAY ARRAY p_orders TO orders.*
     ...
   END DISPLAY
   ON ACTION go_to_header
      NEXT FIELD cust_num
   ON ACTION go_to_detail
      NEXT FIELD order_lineno
   ...
END DIALOG
```

When a `BUTTON` exist in the form layout, it can get the focus if the `DIALOG` block defines a `COMMAND` clause as action handler. Currently there is no way to give the focus to a `BUTTON` by program.

```
DIALOG ATTRIBUTES(UNBUFFERED)
   ...
   COMMAND "print"
      CALL print_order()
   ...
END DIALOG
```

In some seldom cases (especially when using folder tabs), it may be need to show a part of the form that is not controlled by the dialog (i.e. there is no active field or button that can get the focus in that form part, thus the above techniques cannot work). To show temporary a given part of the form that cannot get the focus, use the `ui.Form.ensureFieldVisible()` or `ui.Form.ensurelementVisible()` methods.

```
DEFINE form ui.Form
...
DIALOG ATTRIBUTES(UNBUFFERED)
   ...
   BEFORE DIALOG
      LET form = DIALOG.getForm()
   ...
   ON ACTION show_image1
      CALL form.ensureElementVisible("image1")
   ...
END DIALOG
```

Detection of focus changes

Describes how to detect when the focus goes from field to field or to a read-only list.

Detecting focus changes in a singular INPUT or CONSTRUCT

An singular `INPUT` or `CONSTRUCT` controls several fields that can get the focus and become current. In order to execute some code when a field gets (or loses) the focus, use the following control blocks:

- `BEFORE FIELD` (a specific field (or group of fields) gets the focus)
- `AFTER FIELD` (the field (or group of fields) loses focus)

Detecting focus changes in a singular DISPLAY ARRAY

An singular `DISPLAY ARRAY` controls rows of a list, that can get the focus and become current. In order to execute some code when a row gets (or loses) the focus, use the following control blocks:

- `BEFORE ROW` (a new row gets the focus inside a `DISPLAY ARRAY` or `INPUT ARRAY` list)
- `AFTER ROW` (a row inside a `DISPLAY ARRAY` or `INPUT ARRAY` list loses focus)

Detecting focus changes in a singular INPUT ARRAY

An singular `INPUT ARRAY` controls several fields and rows of a list, that can get the focus and become current. In order to execute some code when a field or a row gets (or loses) the focus, use the following control blocks:

- `BEFORE ROW` (a new row gets the focus inside a `DISPLAY ARRAY` or `INPUT ARRAY` list)
- `BEFORE FIELD` (a specific field (or group of fields) gets the focus)
- `AFTER FIELD` (the field (or group of fields) loses focus)
- `AFTER ROW` (a row inside a `DISPLAY ARRAY` or `INPUT ARRAY` list loses focus)

Detecting focus changes in a DIALOG

A `DIALOG` interaction block can handle different parts of a form simultaneously. In order to execute some code when a part of the form gets (or loses) the focus, use the following control blocks:

- `BEFORE INPUT` (a field of this `INPUT` or `INPUT ARRAY` sub-dialog gets the focus and none of its fields had focus before)
- `BEFORE CONSTRUCT` (a field of this `CONSTRUCT` sub-dialog gets the focus and none of its fields had focus before)
- `BEFORE DISPLAY` (this `DISPLAY ARRAY` sub-dialog gets the focus and none of its fields had focus before)
- `BEFORE ROW` (a new row gets the focus inside a `DISPLAY ARRAY` or `INPUT ARRAY` list)
- `BEFORE FIELD` (a specific field (or group of fields) gets the focus)
- `AFTER FIELD` (the field (or group of fields) loses focus)
- `AFTER ROW` (a row inside a `DISPLAY ARRAY` or `INPUT ARRAY` list loses focus)
- `AFTER DISPLAY` (this `DISPLAY ARRAY` sub-dialog loses the focus = focus goes to another sub-dialog)
- `AFTER CONSTRUCT` (this `CONSTRUCT` sub-dialog loses the focus = focus goes to another sub-dialog)
- `AFTER INPUT` (this `INPUT` or `INPUT ARRAY` sub-dialog loses focus = focus goes to another sub-dialog)

These triggers are also executed by `NEXT FIELD`.

Enabling autocompletion

Autocompletion allows to display a list of proposals while the user is typing text into a field.

Introduction to autocompletion

Text input fields (like `EDIT` and `BUTTONEDIT`) can be defined with autocompletion feature, by combining the `COMPLETER` form field attribute with program code providing the list of proposals in a dynamic array of strings, with the `DIALOG.setCompleterItems()` method, when the `ON CHANGE` trigger is fired for the autocompletion field.

Defining a form field for autocompletion

In order to enable autocompletion in a text form field, you must define the `COMPLETER` attribute:

```
EDIT f1 = FORMONLY.firstname, COMPLETER;
```

The `COMPLETER` attribute can be used for `EDIT` and `BUTTONEDIT` fields.

Providing the front-end with a list of proposals

The `DIALOG.setCompleterItems()` method must be used to provide the list of proposal during dialog execution:

```
DEFINE items DYNAMIC ARRAY OF STRING
-- fill the array with items
LET items[1] = "Ann"
LET items[2] = "Anna"
LET items[3] = "Annabel"
CALL DIALOG.setCompleterItems(items)
```

> **Important:** Consider the execution time of the code creating the proposal list. For example, avoid long complex SQL queries that can take more than a few milliseconds to complete.

The `setCompleterItems()` method will raise error -8114 if the list of items contains more than 50 elements. The purpose of autocompletion is to provide a short list of proposals to the user. Note that this error is not trappable with exception handlers like `TRY/CATCH`, the code must avoid to reach the limit.

Detecting user input

When implementing autocompletion, you must detect when the user modifies the field value, to adapt the list of items with the `setCompleterItems()` method.

In order to detect user input, define the `ON CHANGE` dialog control block, and call a custom function by passing the `DIALOG` object, and the value of the current field as paramter, to filter the proposal list accordingly:

```
INPUT BY NAME rec.firstname
   ...
   ON CHANGE firstname
      CALL fill_proposals_firstname(DIALOG, rec.firstname)
```

For text fields defined with the `COMPLETER` attribute, the `ON CHANGE` trigger will be fired without leaving the field, each time the user types characters in. The event is fired after a short delay, to not overload the UI exchanges between the front-end and the runtime system.

> **Note:** The item list for a field implementing autocompletion is not permanent: The program must re-define the autocompletion item list with `setCompleterItems()`, on every `ON CHANGE` event.

Example

The example below implements form field with autocompletion: Each time the `ON CHANGE` trigger is fired, the set of proposals is adapted to the current field value, to match names that start with the same characters typed by the user.

Form file (`compl.per`):

```
LAYOUT
GRID
{
[f1                             ]
[f2                             ]
}
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.field1, COMPLETER;
EDIT f2 = FORMONLY.field2;
END
```

Program file (`compl.4gl`):

```
DEFINE all_names DYNAMIC ARRAY OF STRING

MAIN
  DEFINE rec RECORD
              field1 STRING,
              field2 STRING
         END RECORD
  CALL fill_names()
  OPEN FORM f FROM "compl"
  DISPLAY FORM f
  OPTIONS INPUT WRAP
  INPUT BY NAME rec.* ATTRIBUTES(UNBUFFERED)
      ON CHANGE field1
         CALL fill_proposals(DIALOG, rec.field1)
  END INPUT
END MAIN

FUNCTION fill_names()
  DEFINE i INTEGER
  LET i=0
  LET all_names[i:=i+1] = "Amanda"
  LET all_names[i:=i+1] = "Ann"
  LET all_names[i:=i+1] = "Anna"
  LET all_names[i:=i+1] = "Annabelle"
  LET all_names[i:=i+1] = "Barbara"
  LET all_names[i:=i+1] = "Barry"
  LET all_names[i:=i+1] = "Brice"
END FUNCTION

FUNCTION fill_proposals(dlg, curr_val)
  DEFINE dlg ui.Dialog, curr_val STRING
  DEFINE curr_set DYNAMIC ARRAY OF STRING,
         i, x INTEGER
  LET x=0
  FOR i=1 TO all_names.getLength()
      IF upshift(all_names[i]) MATCHES upshift(curr_val)||"*"
 THEN
         LET curr_set[x:=x+1] = all_names[i]
      END IF
  END FOR
```

```
  CALL dlg.setCompleterItems(curr_set)
END FUNCTION
```

Dialog actions

Describes how to program action handling when the end user triggers an action on the front-end.

Action handling basics

This topic describes the basics of action views, action events and action handlers.

In the user interface of the application, *action views* can produce *action events*, that will execute user code in the corresponding *action handler* defined in the current interactive instruction of the program.

Actions views are for example `BUTTON` form items.

Action handlers are `ON ACTION` or `COMMAND` dialog blocks that execute user code, in the current interactive dialog.

Action views are bound to action handlers by name.

If no action view is explicitly defined in the current form, the front end will create a "*default action view*" for the action. This is typically a button that appears in a specific area, located and decorated according to the front end platform standards.

Actions can be configured with *action attributes*. These can be defined explicitly at the action view level (button in form), as dialog-specific action configuration (`ON ACTION` *name* `ATTRIBUTES(...)`), or with *action defaults*.

Special actions are supported, such as the `interrupt` action to the user cancel a running application procedure.

Defining action views in forms

How to define action views that will fire action events.

Actions views are form items that can be activated to fire an action event. The action event triggers user code in an `ON ACTION` block.

We distinguish action views defined explicitly in form files from default action views. A default action view will automatically appear when an action handler is implemented in the current dialog (if no explicit action view with the same name exists in the form). Default action view creation can be controlled with the `DEFAULTVIEW` action attribute.

To fire user code, action views are bound to action handlers by name.

Action view decoration attributes (`IMAGE` for icons, `TEXT` for label, `COMMENT` for hint) can be centralized in action defaults.

Action views can be items of form elements dedicated to action execution, such as `TOOLBAR` items (i.e. toolbar buttons) or `TOPMENU` options:

```
TOOLBAR
   ITEM accept
   ITEM cancel
   ...
END
```

Action views can be typical `BUTTON` items defined in the form `LAYOUT`:

```
LAYOUT
GRID
{
   [b1     ]
   ...
```

```
}
...
ATTRIBUTES
BUTTON b1 : print, IMAGE="printer";
...
```

Action views can be sub-elements of other elements, as when defining a `BUTTONEDIT` with an `ACTION` attribute:

```
LAYOUT
GRID
{
   [f1             ]
   ...
}
...
ATTRIBUTES
BUTTONEDIT f1 = customer.cust_city, ACTION=choose_city, IMAGE="zoom";
...
```

Action views can also be simple `IMAGE` items, when the `ACTION` attribute is specified:

```
LAYOUT
GRID
{
   [i1             ]
   ...
}
...
ATTRIBUTES
IMAGE i1: image1, ACTION=show_details, IMAGE="mylogo";
...
```

Note that `IMAGE` fields can be defined as `TABLE` columns and define the `ACTION` attribute to trigger user code:

```
LAYOUT
GRID
{
<TABLE t1          >
[c1  |c2       |c3 ]
[c1  |c2       |c3 ]
[c1  |c2       |c3 ]
...
}
...
ATTRIBUTES
...
IMAGE c3: FORMONLY.image, ACTION=delete;
...
```

For more details about image column actions see Defining actions on list columns with images on page 1355.

The row selection in a `TABLE` (or `TREE`) will be considered an action view when defining the `DOUBLECLICK` attribute:

```
DISPLAY ARRAY arr TO sr.*
        ATTRIBUTES(UNBUFFERED, DOUBLECLICK=select)
  ...
END DISPLAY
```

Action views can also be graphical elements that are standard action triggers on the front-end platform, such as the [x] cross button of desktop windows, that will automatically bind to a "close" action, or the FAB button of Android, which can be configured to trigger a specific action.

Implementing dialog action handlers
How to execute user code in ON ACTION blocks when an action is fired.

Actions handlers are typically defined in dialog instructions with the `ON ACTION` interaction block. You must specify the name of the action after the `ON ACTION` keywords:

```
INPUT BY NAME ...
   ...
   ON ACTION print
      -- user code
   ...
```

Action handlers can also be defined with the `COMMAND` syntax in `MENU` and `DIALOG` instructions:

```
MENU ...
   ...
   COMMAND "Print" "Print the current record"
      -- user code
   ...
```

`ON ACTION` blocks provide better abstraction than `COMMAND` blocks by using simple action identifiers and leaving the decoration in the form files or action defaults files. The `ON ACTION` block defines an action handler with a simple action name. The `COMMAND` block defines an action handler with an action name, but it also defines decoration attributes, such as the label and comment. Keyboard accelerators and help topic numbers can also be defined.

> **Note:** Action views controlled by `ON ACTION` handlers cannot get the focus. When using the `COMMAND` action handler, action views such as a `BUTTON` defined in the form layout can get the focus and are part of the tabbing item list.

Action handlers are bound to action views by name.

Binding action views to action handlers
How are action views of the forms bound to action handlers in the program code?

Action views (such as buttons) are bound to action handlers by the `name` attribute. Action handlers are defined in interactive instructions with an `ON ACTION` clause or `COMMAND` / `ON KEY` clauses.

For example, in the `ATTRIBUTES` section of the form, a button may be defined as follows:

```
BUTTON b1: show_help, TEXT="Show Help";
```

The corresponding action handler (code) in the program will use the "`show_help`" action name:

```
ON ACTION show_help
   CALL ShowHelp()
```

The `COMMAND` / `ON KEY` clauses are typically used to write text mode programs. Such clauses define the name of the action and the decoration label. It is recommended that you use `ON ACTION` clauses instead, because they identify user actions with an abstract name. However, if required, you can use a `COMMAND` clause in a non-menu dialog to include the corresponding action view in the focus-able form items.

In the `ON ACTION` *`action-name`* clause, the name of the action must be a valid identifier, preferably written in lowercase letters. In the abstract user interface tree (where the action views are defined), action names are case-sensitive (as they are standard DOM attribute values). However, identifiers are not case-

sensitive in the language. The fglcomp compiler always converts the action identifiers of `ON ACTION` clauses to lowercase:

```
ON ACTION PrintRecord   -- will be compiled as "printrecord"
```

To avoid confusion, always use lower-case names for action names (for example, `print_record` instead of `PrintRecord`).

Default action views
A default action view is created to render an action handler when no explicit action view exists for it.

If no explicit action view is defined, such as a toolbar button, a topmenu item or a simple button in the form layout, the front end creates a *default action view* for each `COMMAND` or `ON ACTION` action handler, or implicit action such as insert/delete in `INPUT ARRAY`, in the current interactive instruction.

The rendering of default action views depends from the platform. On a desktop front-end, the default action views appear as buttons in the action frame in the right-hand side of the current window. On a mobile device, the default action views will follow the mobile user interface standards, which can be vendor specific. For more details about default action views on mobile, see Rendering default action views on mobile on page 1279.

When creating action handlers with `ON KEY` (or `COMMAND KEY` without a command name in a `MENU`), the default action view is invisible. If you define a `text` attribute in the action defaults, the default action view is made visible.

Control the default action view visibility by using the `DEFAULTVIEW` action attribute.

If one or more action views are defined explicitly for a given action, the front end considers that the default view is not needed. Typically, if you define in the form a `BUTTONEDIT` field, a `BUTTON`, or a `TOOLBAR` item that triggers the action, you do not need an additional button in the action frame.

The presentation of the default action views can be controlled with presentations style attributes for the `Window` AUI tree nodes.

Rendering default action views on mobile
Default action views are rendered according to the mobile specific standards.

Default action view rendering on mobile

The top and/or bottom part of the app screen is dedicated to displaying default action views to the user.

Key functions of these areas:

- Make important actions prominent and accessible in a predictable way (such as **New** or **Search**).
- Support consistent navigation and view switching within apps.
- Reduce clutter by providing an action overflow for less-used actions.
- Provide a dedicated space for giving your app an identity with text and/or an image.

How actions are rendered on the mobile device depends on:

- the order of the `ON ACTION` statements in the current dialog of the running app.
- The type of platform (Android™/iOS).
- The type (phone/tablet) and orientation of the device.

Actions are mapped to the Android or iOS platform in a specific way, following the platform standard.

Actions can be programmatically enabled and disabled, and hidden and shown. The text, image and other properties of the action can be controlled with action attributes.

GUI elements to trigger actions on mobile devices

Each mobile platform provides its own standard to display action triggers.

GMA and GMI follow respectively the Android and iOS standards:

- Navigation controller on iOS devices on page 1280
- Action bar on Android devices on page 1285
- Floating action button on Android devices on page 1286

Decorating action views on mobile

Actions are typically decorated using the `IMAGE` or the `TEXT` action attribute. If these attributes are not defined or if the specified image resource is not available, the mobile front-end uses a default decoration. For some actions, the front-end always uses the platform-specific decoration. For example, on iOS devices, the "refresh" action always renders as a typical circular arrow icon.

Well-known actions use a default icon or text corresponding to the mobile platform GUI guidelines. As these follow the mobile OS standards, do not define your own text or icons for common actions such as "accept" or "cancel".

For a complete list of predefined action decorations, see:

- Default action views decoration on iOS devices on page 1284
- Default action views decoration on Android devices on page 1288

Rendering close/cancel/accept actions on Android devices

The physical back button on an Android device is considered a default action view for the "close", "cancel", or "accept" action in the current dialog:

- If a close action is defined, it is assigned to the back button.
- If the close action is not defined, but the cancel action is defined, it is assigned to the back button.
- If neither close nor cancel actions are defined, but the accept action is defined, it is assigned to the back button.

If accept or cancel cannot be assigned to the back button, a default action view appears in the action panel. For example, if all three actions (close, cancel and accept) exist and are active, the action panel shows a check mark for the accept action and a cross icon for the cancel action, while the back button fires the close action.

Navigation controller on iOS devices
On iOS devices, apps display a **navigation controller** on the top of the screen.

The iOS **navigation controller** is made of a **navigation bar** on the left side and a **common action pane** on the right side.

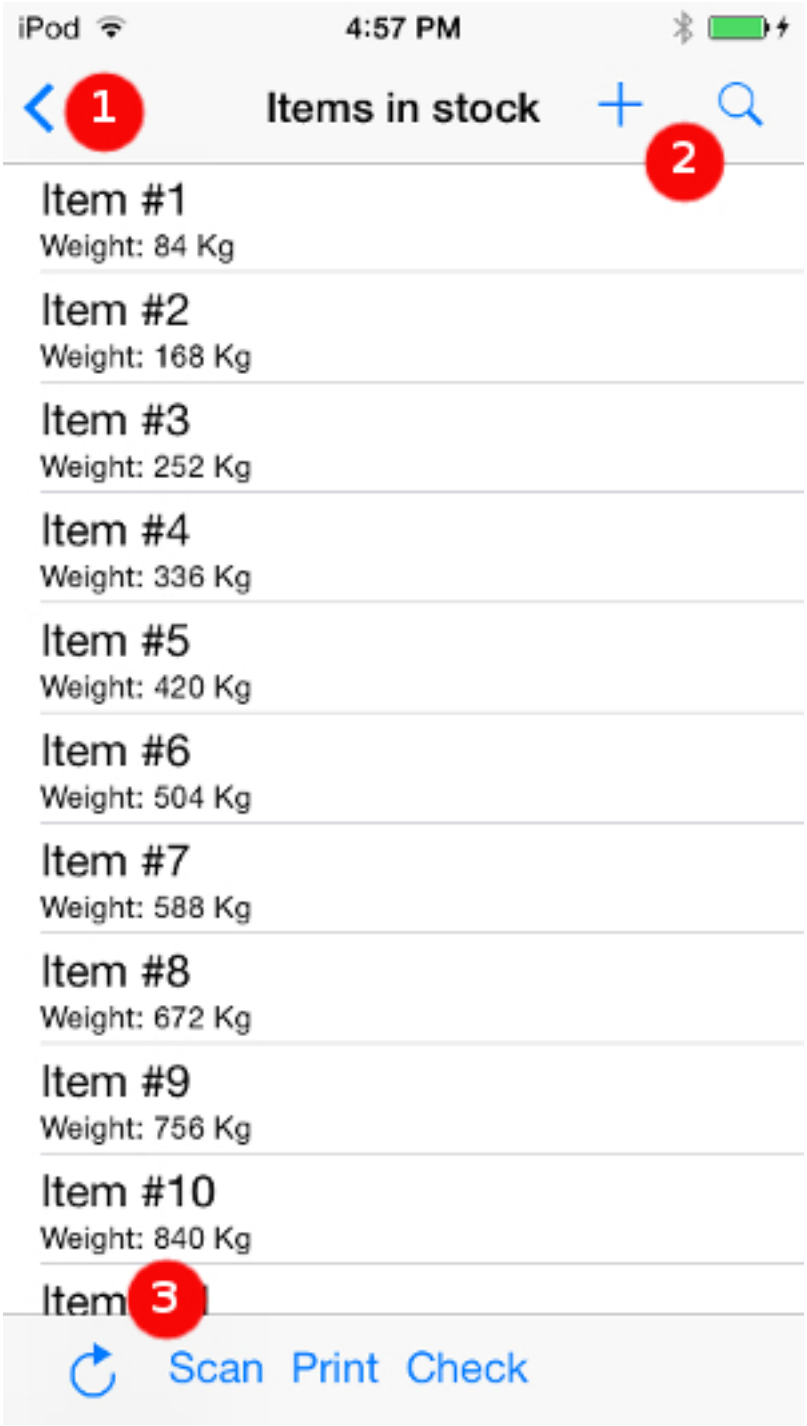


Figure 78: iOS app interface

Navigation bar (1)

The left side navigation bar provides a linear path through various screens. The `accept`, `cancel` or `close` action is rendered as back button, respectively in the order of precedence given here. If there is a previous form or window, then this button shows the title of the previous page. If there is not a form to return to, the "back" navigation button is shown.

Common action pane (2)	The right-hand side is the common action pane. Default action views are displayed here, in the order of the current dialog's `ON ACTION` statements of the current dialog.
Toolbar pane (3)	When default action views are displayed, if there is not enough room in the common action pane (2), the remaining actions are displayed in the toolbar pane at the bottom of the screen. If there is not enough space to display all action views in the toolbar pane, an overflow icon appears on the right. Tap on the overflow icon to show the remaining action views. Use a TOOLBAR in your form, to have full control on the toolbar pane. An action displayed as a TOOLBAR item in the toolbar pane will no longer display as default action view in the common action pane (2).

In this screen shot, the device is oriented in landscape mode. The app is the same, yet since there is enough space in the navigation bar, all default action views display in the common action pane.

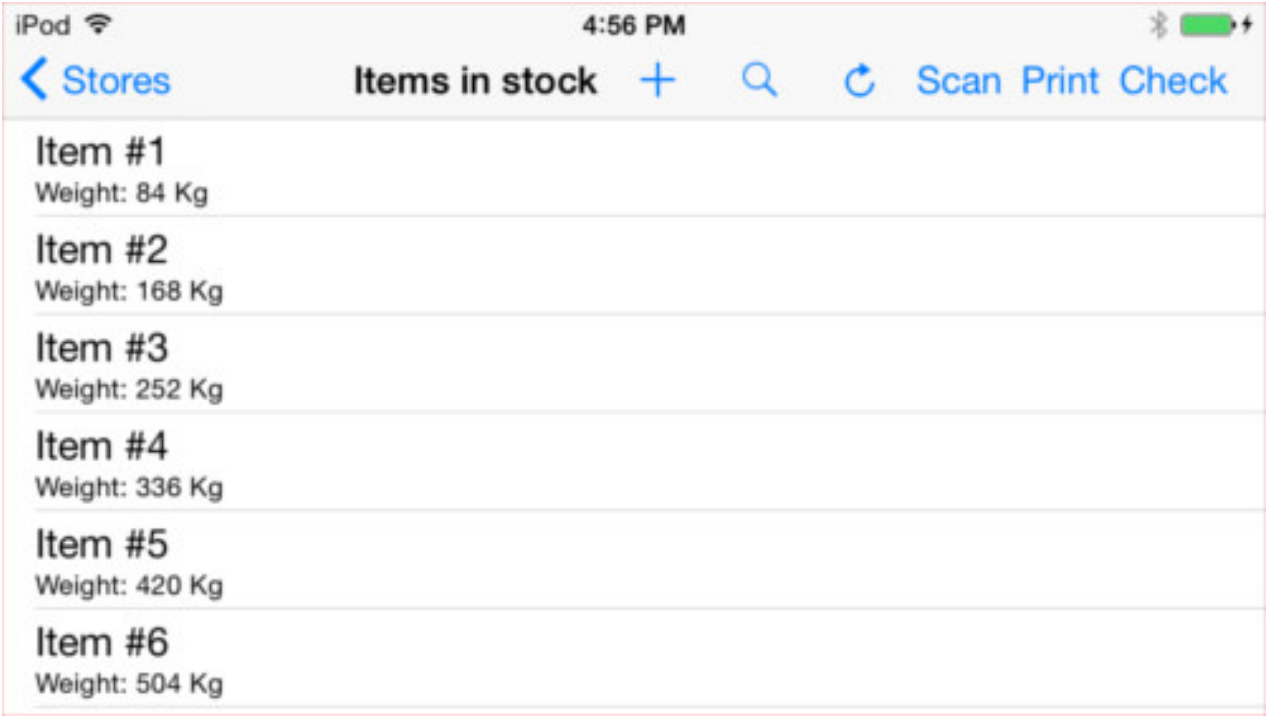


Figure 79: iOS app interface in landscape mode

To customize the application, define the colors of the iOS navigation bar and toolbar with the following Window-class style attributes:

- `iosTintColor`, for items in (1), (2) and (3) (and for other form items)
- `iosNavigationBarTintColor`, for (1) and (2).
- `iosToolBarTintColor`, for (3).

For example, by setting the following style attributes, the navigation bar will render as shown in the screen shot:

```
<Style name="Window">
```

```
  <StyleAttribute name="iosTintColor"                value="darkRed" />
  <StyleAttribute name="iosNavigationBarTintColor" value="orange"  />
  <StyleAttribute name="iosToolBarTintColor"        value="orange"  />
  <StyleAttribute name="iosTabBarTintColor"         value="orange"  />
</Style>
```

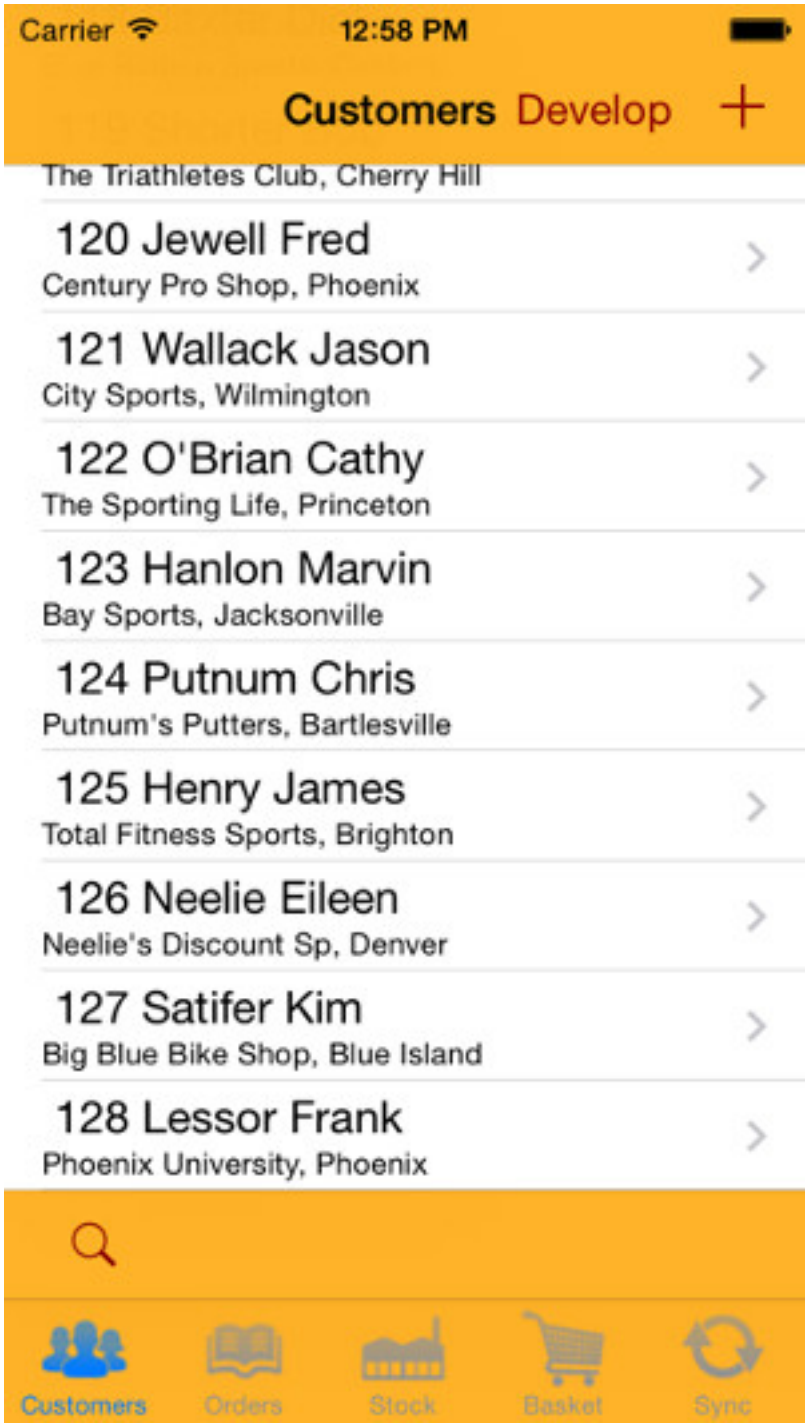


Figure 80: iOS (7) colored navigation bar

Default action views decoration on iOS devices
Common default action views get a decoration implicitly, following iOS standards.

On iOS devices, the decoration for well known actions can be a symbol or a text. When a text is used, it is internationalized. For example, the "accept" action translates to "Done" when the mobile language is English, "Fertig" in German and "OK" in French.

For the default action views of the common actions, the decoration will always follow the iOS standards, even if an attribute is explicitly specified for the action. For example, if you implement an `ON ACTION save` action handler with `ATTRIBUTES(TEXT="Write",IMAGE="disk")`, the action view renders with the "Save" text on an iOS device configured for the English language.

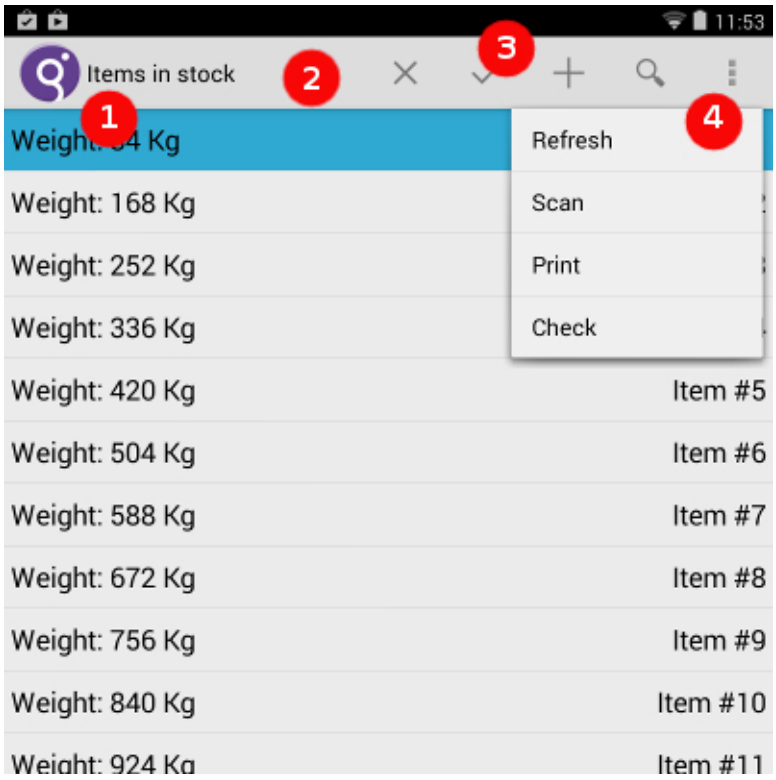
Table 283: Default rendering for common actions on iOS

Action name	iOS default rendering	Symbol
`accept`	Internationalized text (English: Done)	N/A
`cancel`	Internationalized text (English: Cancel)	N/A
`refresh`	Typical circular symbol	
`insert`	Typical plus sign symbol	
`append`	Typical plus sign symbol	
`delete`	Typical trash symbol	
`find`	Typical magnifier symbol	

Action name	iOS default rendering	Symbol
		
`search`	Typical magnifier symbol	
`edit`	Internationalized text (English: Edit)	N/A
`save`	Internationalized text (English: Save)	N/A

Action bar on Android™ devices
On Android devices, apps show an **action bar**.

The Android **action bar** displays in the top of the screen, with several elements having a specific purpose:



The app icon (1)

The app icon and the title of the current form display in the upper left corner.

The application title is defined by the TEXT attribute of the main window displayed by the application.

The icon that appears is either the icon set for the app in the packaging, or it is the image specified by the `ui.Interface.setImage` method. The application icons must be included in the deployment package (.apk) and follow the Android standards (several icon sizes are required).

The view control (2)

If your app implements different views controlled by a top-level navigator, this segment allows users to switch between views. For more details, see Navigator pane on page 1399. In an application handling multiple views in parallel, the view control item displays as a text button.

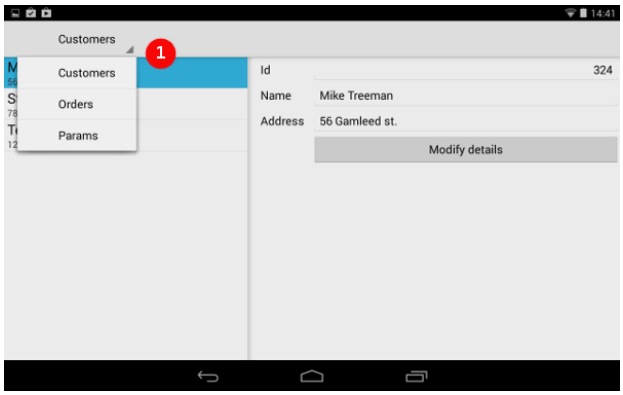


Figure 81: Android View Control

Action buttons (3) and Action overflow (4)

The right-hand side of the action bar shows the actions. The action buttons (3) show the most important actions of your app. Actions that do not fit in the action bar are moved to the action overflow, and an overflow icon appears on the right. Tap on the overflow icon to display the list of remaining action views. If the device has a physical Menu button, the overflow actions are accessible by pressing the physical Menu button and not from an action overflow icon.

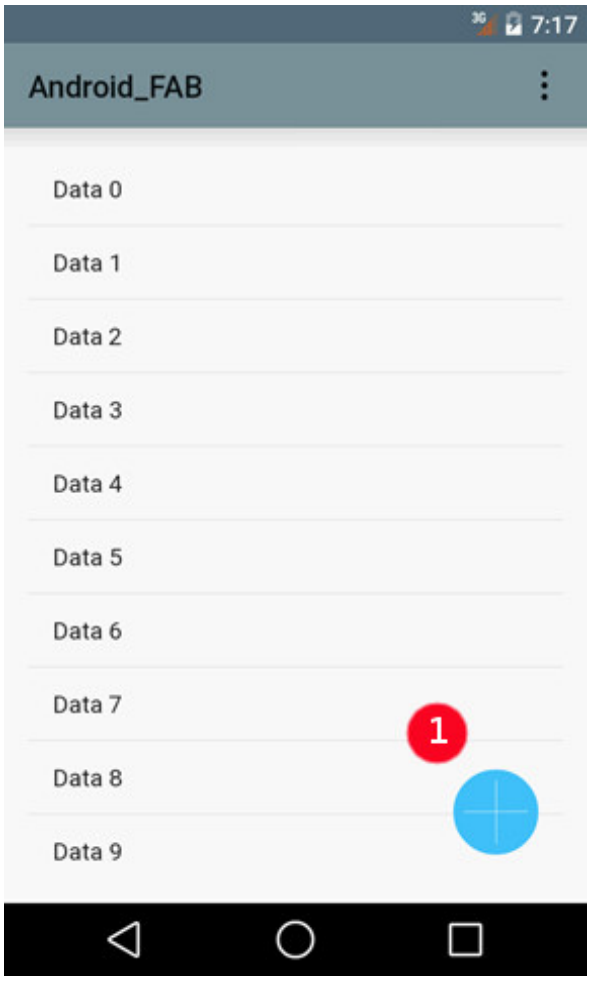
Actions display in the order of the `ON ACTION` statements of the current dialog. If a toolbar is defined, the actions defined in the toolbar take priority and list prior to other actions, in the order they are defined in the toolbar.

If an image is available, it is displayed, otherwise the action text is shown. Depending on the space available (space used by the app icon, screen size, orientation, and so on), the number of actions and the device type, Android displays either the icon or the icon and the text of the action.

Floating action button on Android™ devices

On Android devices, apps using material design show a **Floating Action Button (FAB)**.

The Android **floating action button** displays on the bottom right of the screen, and can be tapped to fire a specific action:



The floating action button (1)

The material design guidelines include the concept of promoted actions, that can be triggered with the floating action button.

Define the list of actions that can be fired from the FAB button with FAB configuration style attributes:

```
<Style name="Window">
    <StyleAttribute
 "materialFABActionList="accept,select,detail"
</Style>
```

The order of the actions define which action is triggered when the FAB button is tapped, and several matching actions are active. With the above example, if the "accept" action is disabled, and the "select" and "detail" actions are active, a tap on the FAB button fires the "select" action.

The icon of the FAB button is defined by the `IMAGE` attribute of the corresponding action. If no `IMAGE` attribute is defined for the action, a default icon is selected from the built-in icons, according to the name of the action. See Default action views decoration on Android devices on page 1288 for

more details about action names to default Android built-in icon mapping.

Default action views decoration on Android™ devices
Common default action views get a decoration implicitly, following Android standards.

On Android devices, when the `IMAGE` and the `TEXT` action attributes are not defined for an action, the default action view gets an implicit decoration.

The default icon is selected according to the name of the action: The symbol is picked from a built-in images (i.e. Android material design icons), if it has the same name as the action. If no icon corresponds, the default action view will get no icon.

The text defaults to the name of the action, converted to uppercase. The text displays only if the Android system considers that the screen is large enough to display the texts. Typically, texts are shown on tablets, but not on smartphones with small/medium screens.

For example, when implementing a `ON ACTION refresh` handler, GMA will implicitly use the default icon with the name "refresh" (the typical circular refresh symbol), and, if there is enough room, display the text "REFRESH" on the right of the icon.

Position and rendering of default action views can be controlled with Android specific style attributes. For more details, see Default action view style attributes on page 825.

Not also that some actions can be rendering as the Floating Action Button of material design, as described in Floating action button on Android devices on page 1286.

Default actions views displayed in the top control bar and in the overflow button will get a text but no icons, while the FAB material design button will get an icon but no text.

The next table shows the default icons that will be selected for common Genero BDL action names.

> **Note:** This table does not list all possible built-in icons: More images are available from the Android material design icon library, and the GMA will select the icon according to the action name. For example, an action with the name "audio" will get the Android music symbol icon:

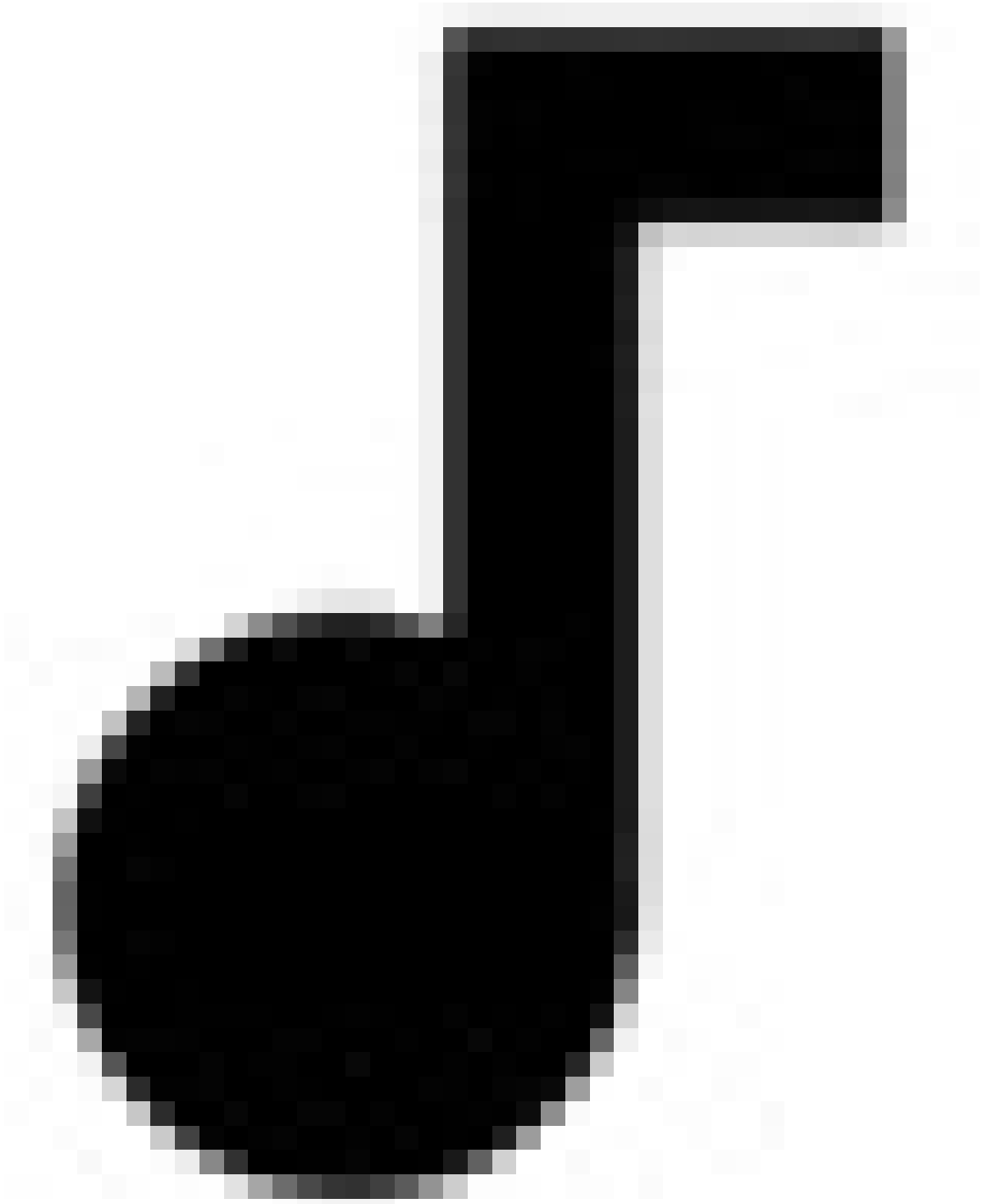
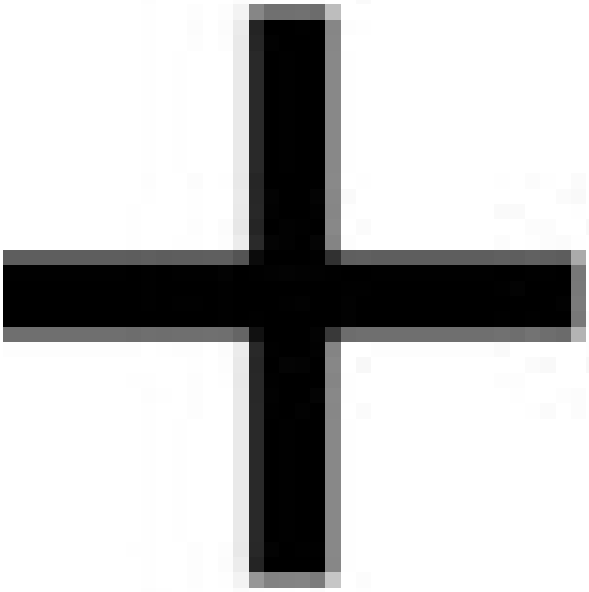
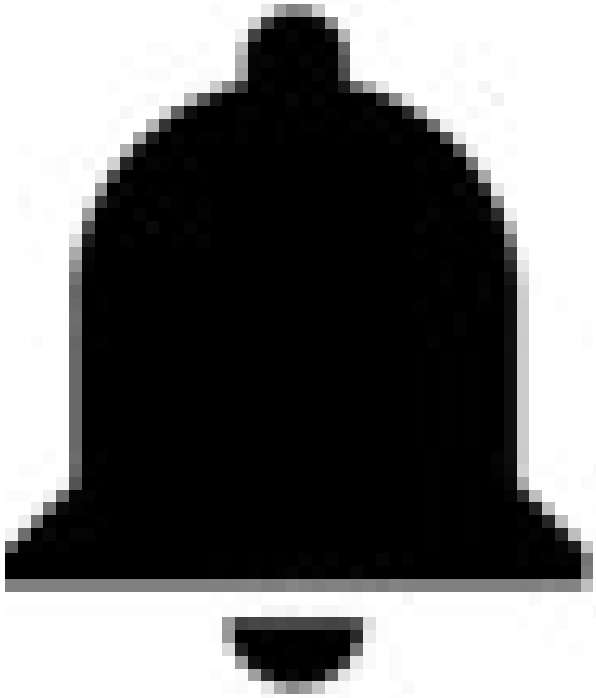
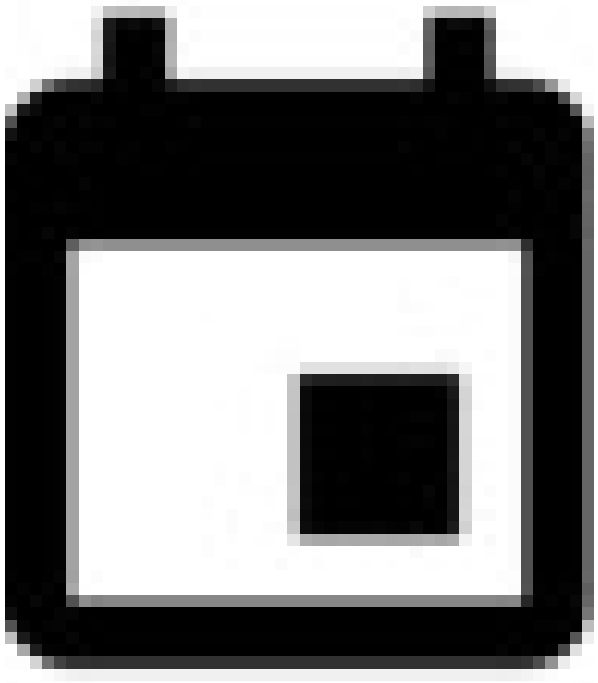


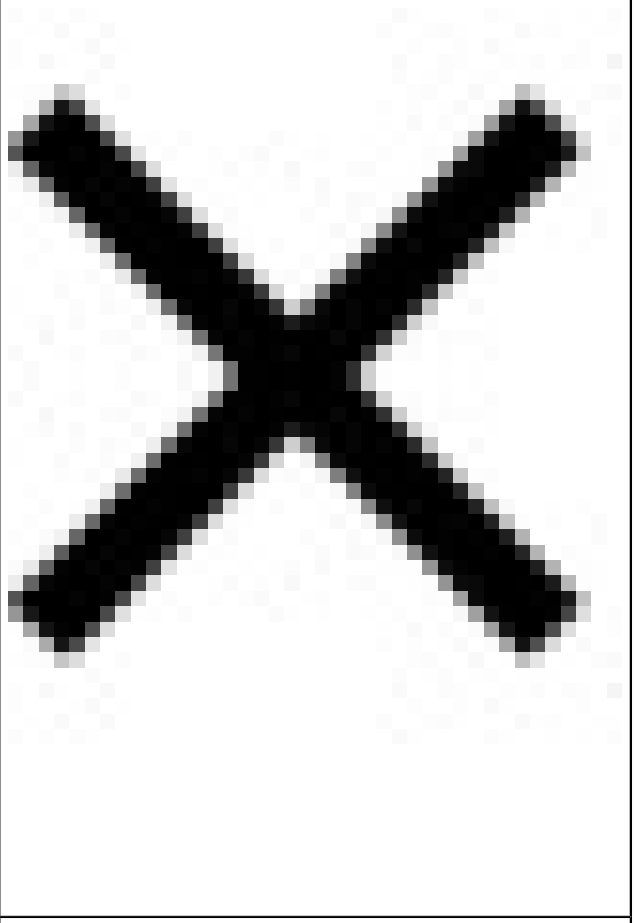
Table 284: Default icons for common actions on Android

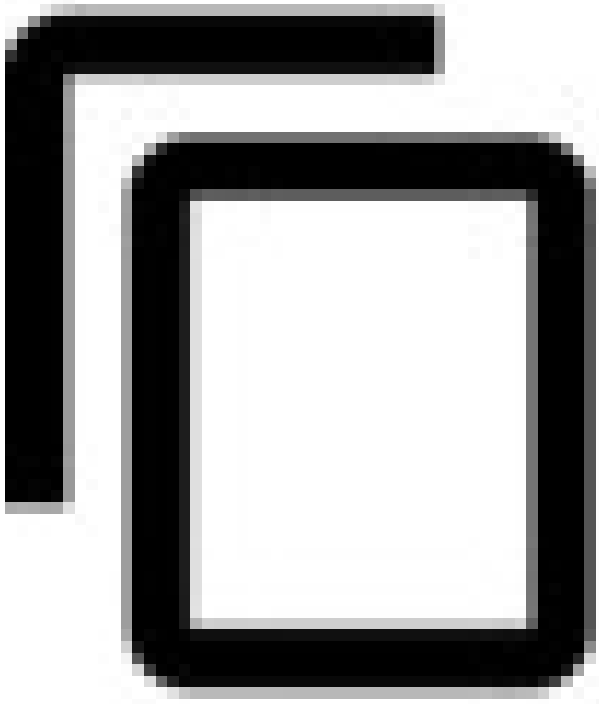
Action name	Icon
`about`	
`accept`	
`append`	

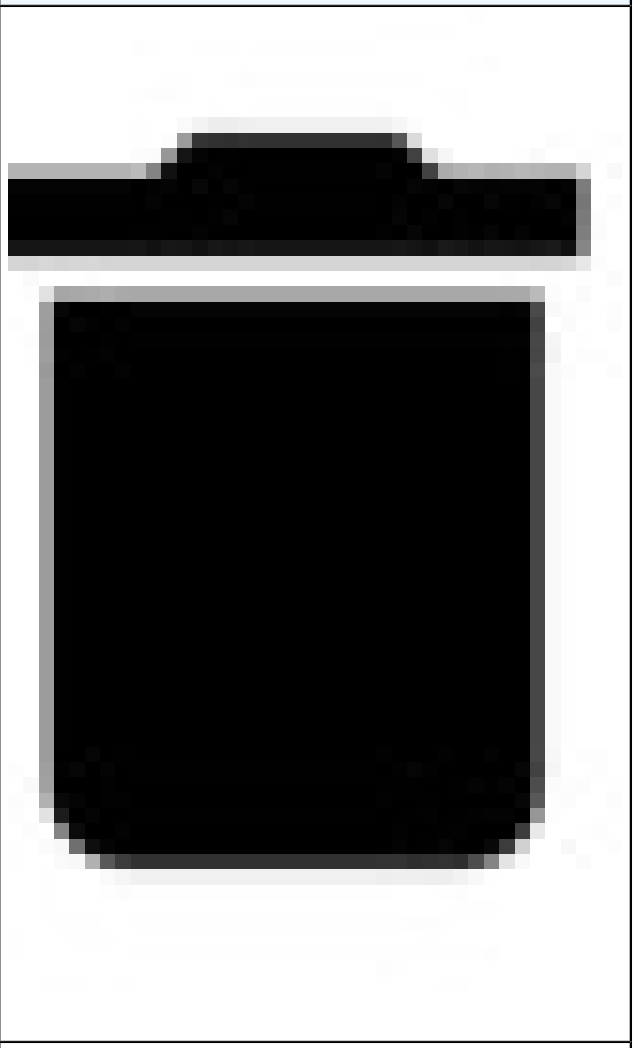
Action name	Icon
	
`attention`	

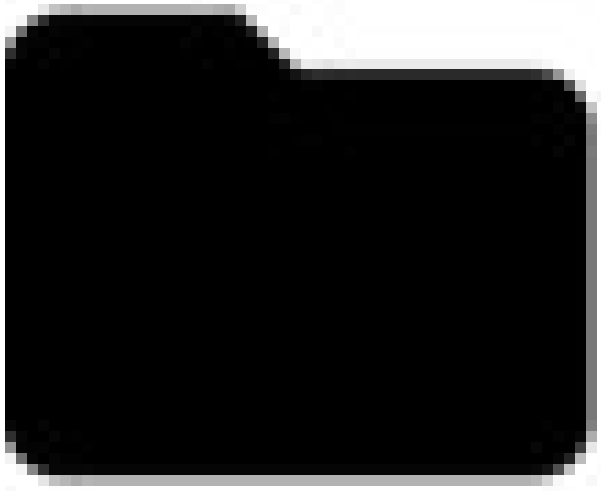
Action name	Icon
	
`bell`	
`calendar`	

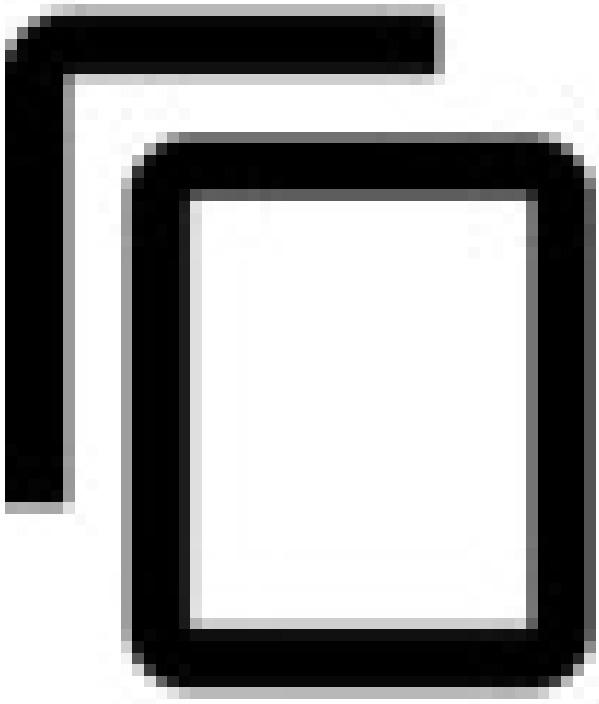
Action name	Icon
	
`camera`	
`cancel`	

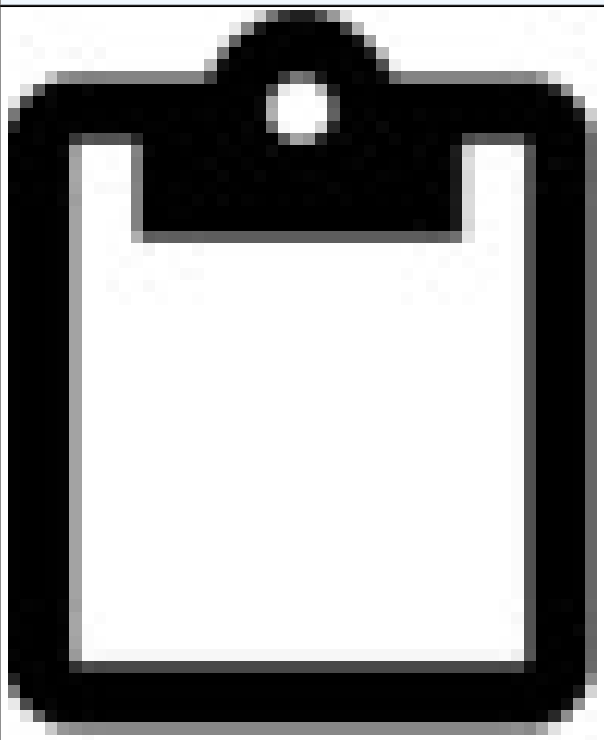
Action name	Icon
	
copy	

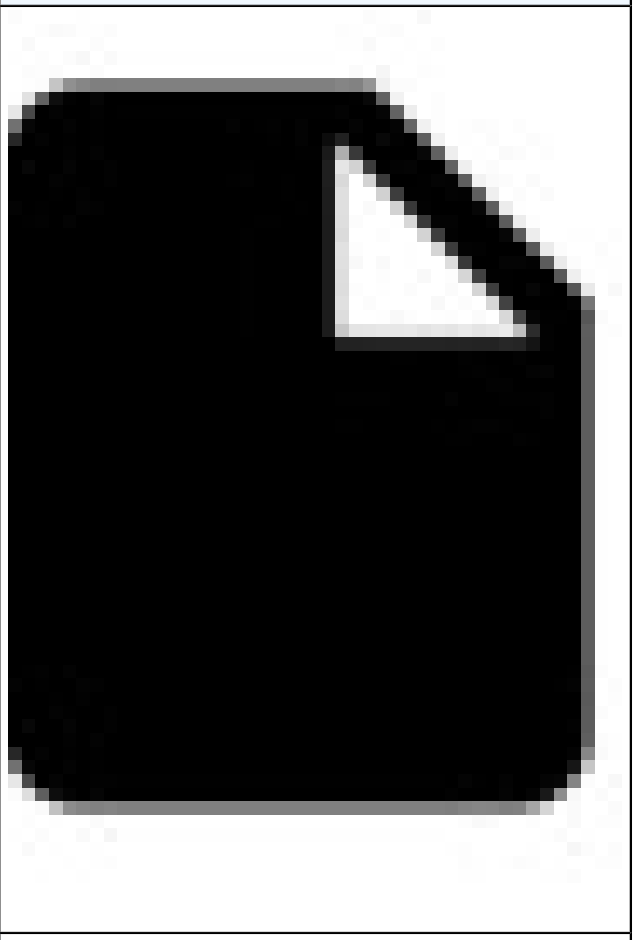
Action name	Icon
	
`cut`	
`delete`	

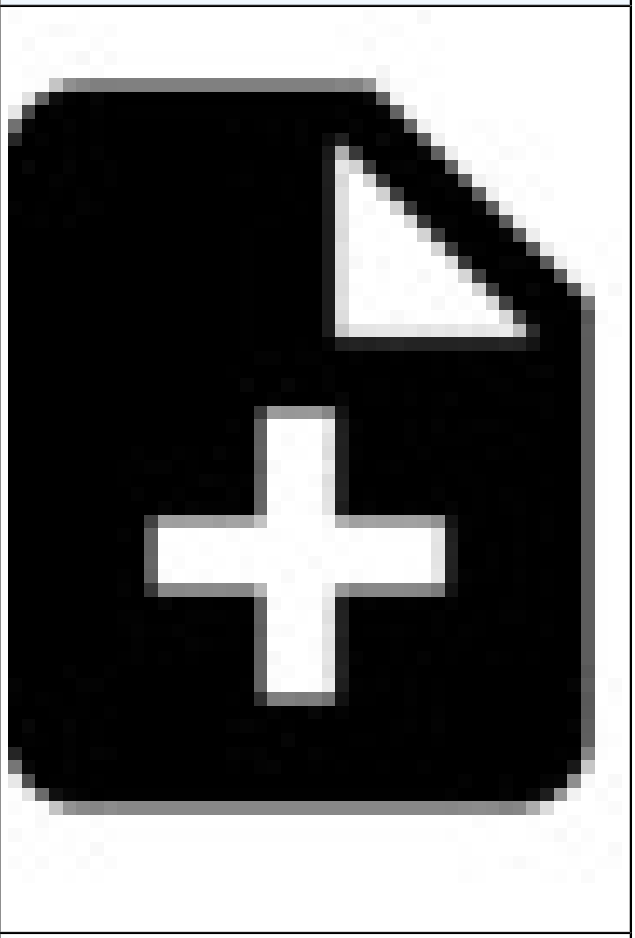
Action name	Icon
	
`diropen`	

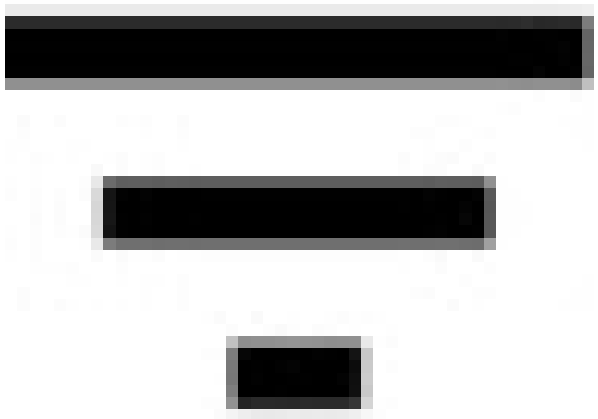
Action name	Icon
	
edit	
editcopy	

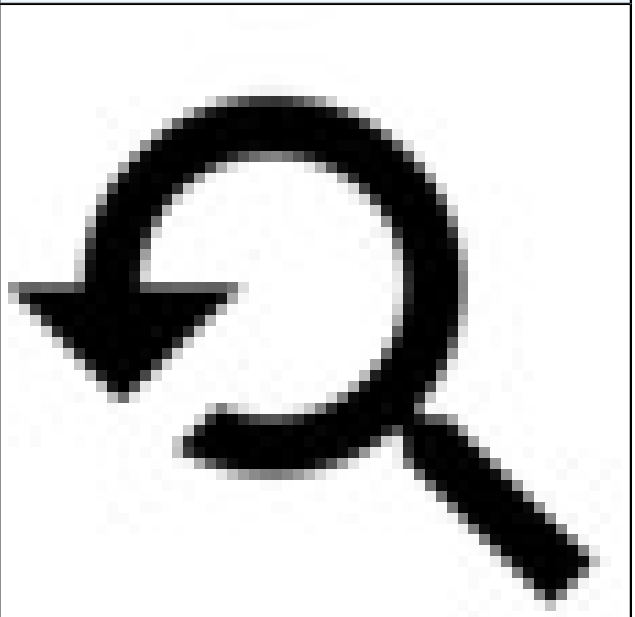
Action name	Icon
	
editcut	
editpaste	

Action name	Icon
	
exit	
file	

Action name	Icon
	
`filenew`	

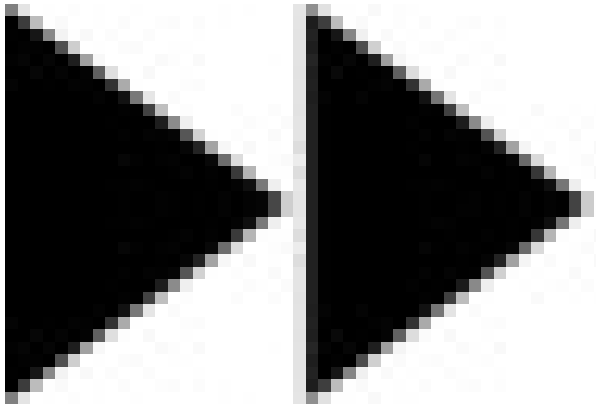
Action name	Icon
	
`filter`	

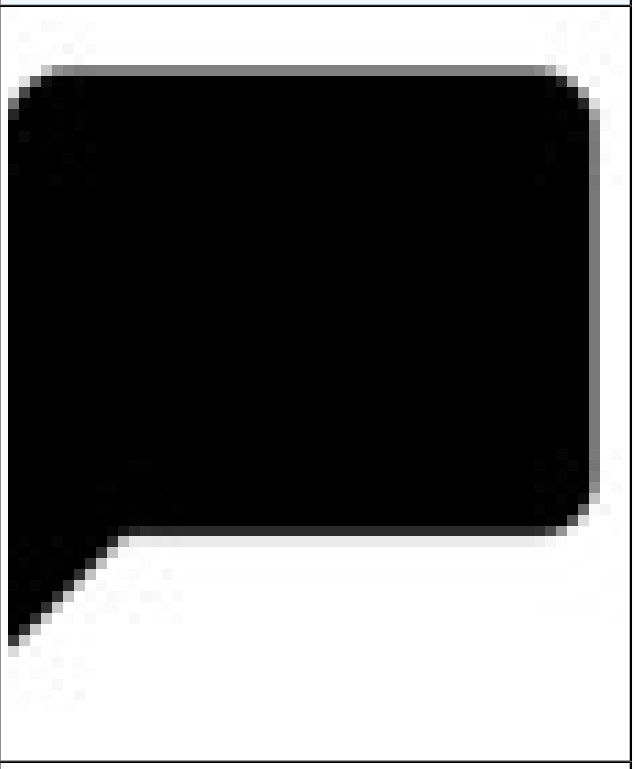
Action name	Icon
	
`find`	
`findnext`	

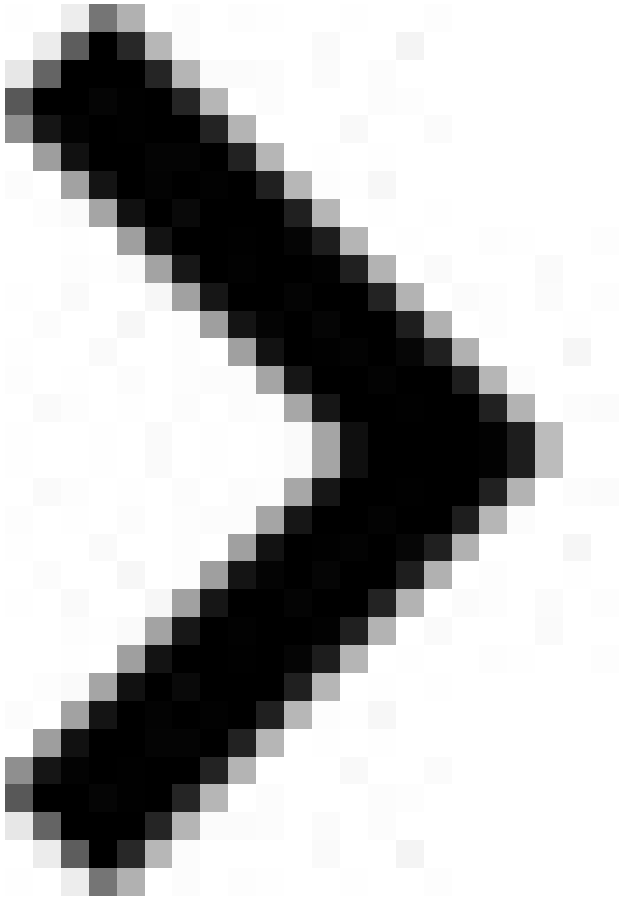
Action name	Icon
	
`first`	

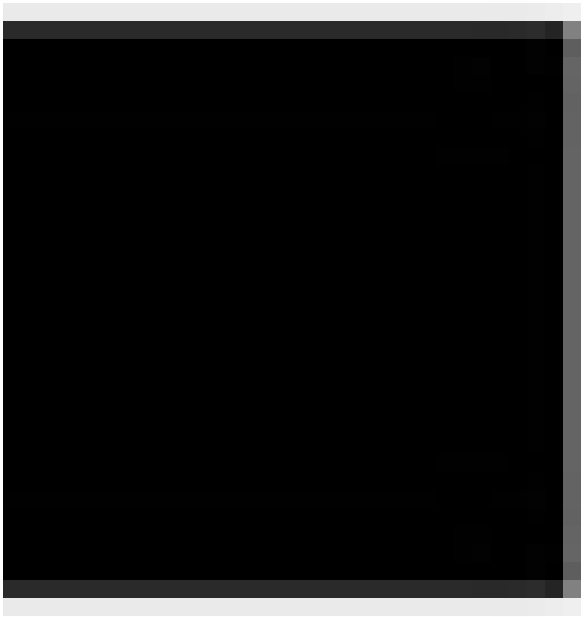
Action name	Icon
`firstrow`	

Action name	Icon
forwind	

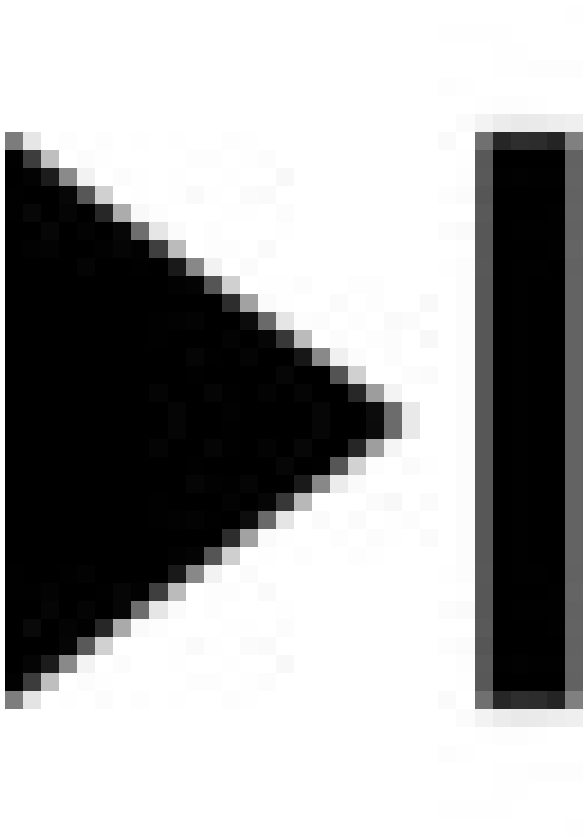
Action name	Icon
	
help	
hint	

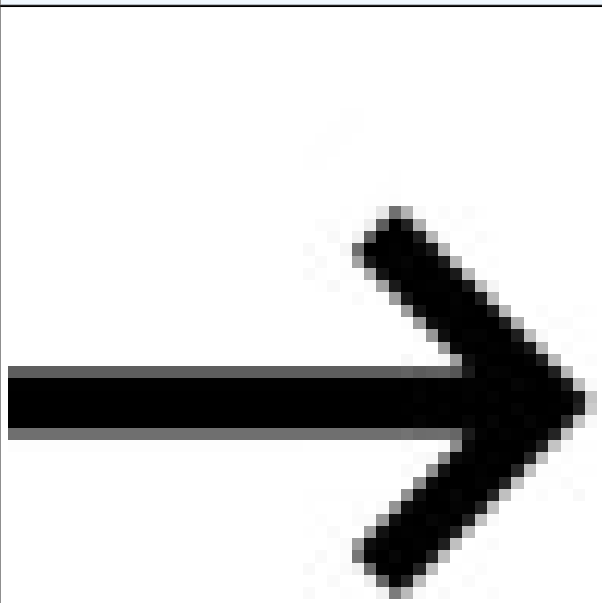
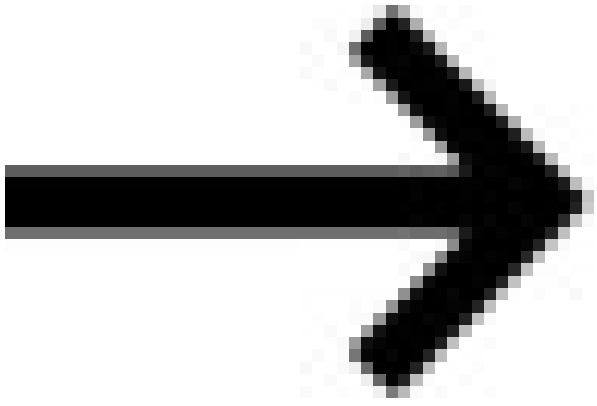
Action name	Icon
	
`insert`	

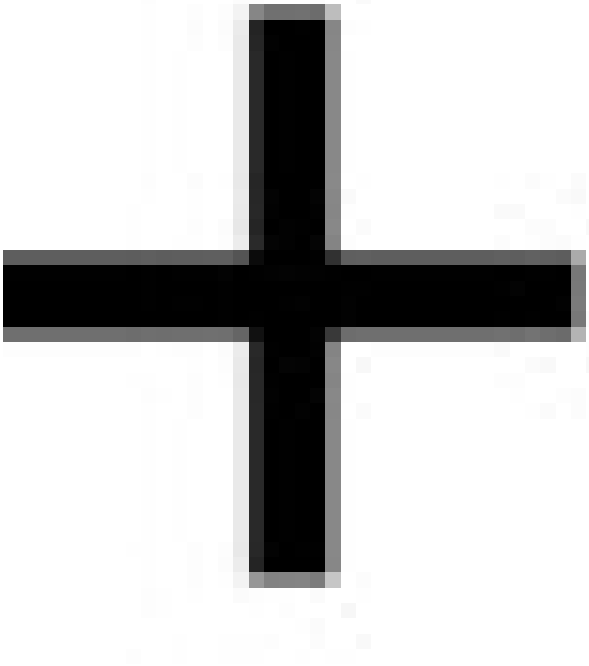
Action name	Icon
	

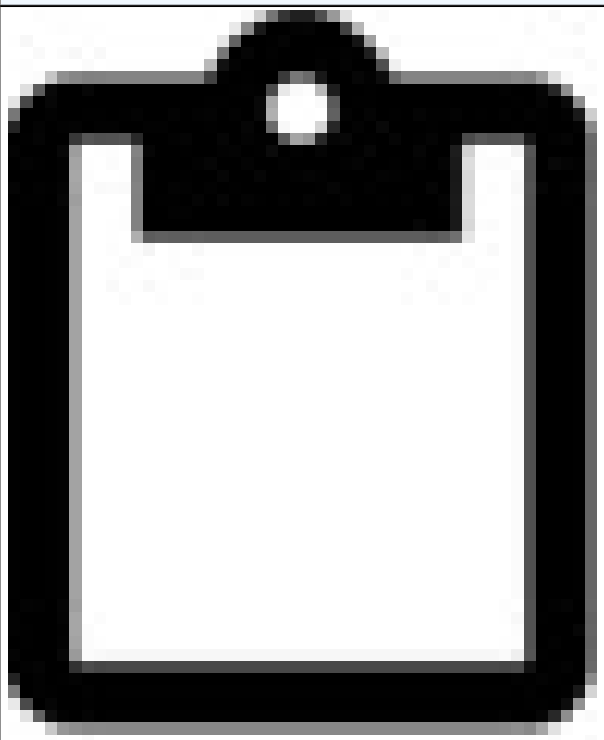
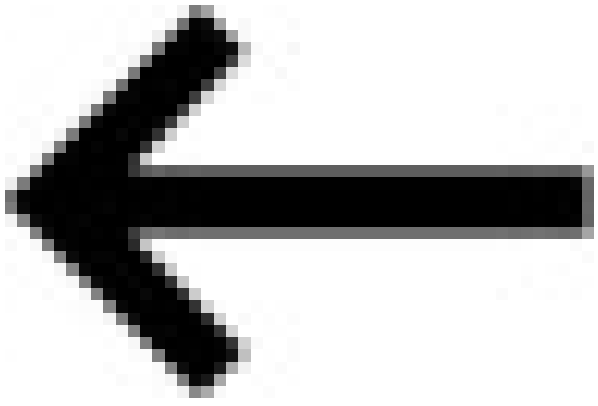
Action name	Icon
`interrupt`	
`last`	

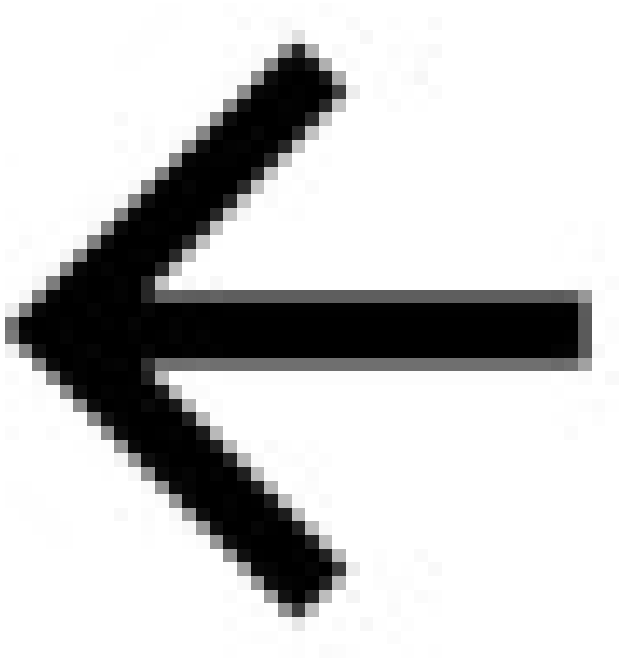
Action name	Icon
lastrow	

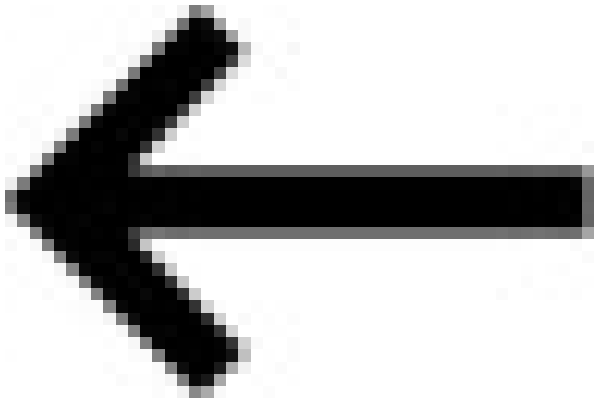
Action name	Icon
`next`	

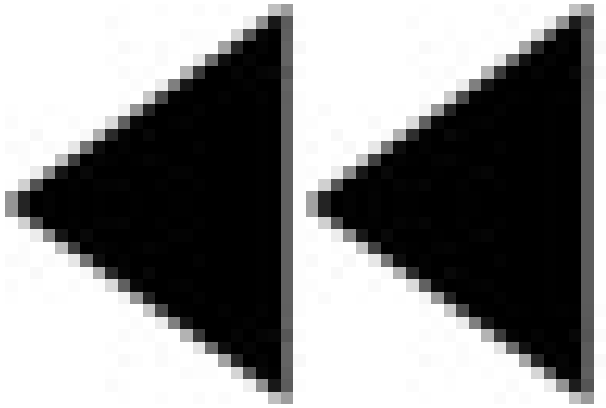
Action name	Icon
	
nextrow	

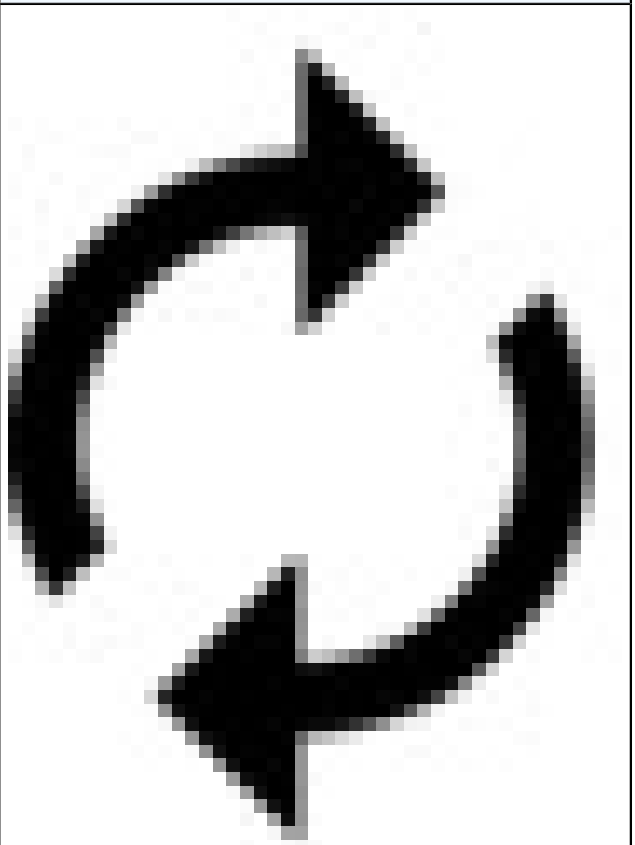
Action name	Icon
new	
paste	

Action name	Icon
	
`prev`	

Action name	Icon
previous	
prevrow	

Action name	Icon
	
`query`	
`rewind`	

Action name	Icon
	
`update`	

Action name	Icon
	

Configuring actions
Action attributes related to decoration, keyboard shortcuts and behavior can be defined with *action attributes*.

Action attributes define attributes for actions, including decoration such as text, icon, comment, as well as keyboard accelerator (ctrl-?, function keys), and also semantics such as current field validation control when an action is fired.

The action attributes can be defined at different levels, through action defaults, form item attributes and action handler attributes:

1. Common action attributes can be centralized in a global action defaults file with the `.4ad` extension,
2. Form-specific action attributes can be defined in the `ACTION DEFAULTS` section of a form definition file,
3. Dialog-specific action attributes can be defined in programs with the `ATTRIBUTES()` clause of `ON ACTION` handlers.
4. Form-item specific action view attributes (decoration only) can be defined directly at the item level (labels, icons, comments).

Action attributes do not only define action view decoration: It is possible to define the semantics of an action, for example by using the `VALIDATE` action default attribute. Functional attributes take effect for a given action when the dialog implementing the action handler becomes active.

Action attributes are particularly important to render the default action view (when there is no explicit action view defined in the form). This is typically the case when not form is associated to the dialog.

Action attributes can be defined with action defaults: Common action defaults are defined in a global action defaults (`.4ad`) file, while form specific actions are define withing the `ACTION DEFAULTS` section of form files.

If a dialog is not attached to a specific form such as an independent `MENU`, define the action attributes with the `ATTRIBUTES` clause on `ON ACTION` handlers, to render the default view and configure the action semantics. Attributes defined by `ON ACTION` *`action-name`* `ATTRIBUTES()` will only be applied to the default action view: The elements in the forms do not get decoration attributes defined by dialog action handlers.

The final decoration and functional attribute values are set in this order of precedence:

1. The attribute defined in the action view element definition itself (local form element decoration).
2. The attribute defined in the `ATTRIBUTES` clause of an `ON ACTION` handler.
3. The attribute defined for the action in the `ACTION DEFAULTS` section of the current form.
4. The attribute defined for the action in the global action defaults file (`.4ad`).

Note that the syntax to define action attributes depends on the context where the action attributes are defined:

- In the `.4ad` file, the syntax follows XML standards, as defined in Action default attributes reference (.4ad) on page 797.
- In the `.per` files, the syntax follows the form specification file attributes, as defined in ACTION DEFAULTS section on page 903.
- In the `.4gl` files (in dialog action handlers), the syntax follows the language syntax, as defined in ON ACTION block on page 1056.

Example

Consider the following parts of code related to the same action definition, namely "print":

1. A `BUTTON` item defined in the form specification file:

```
ATTRIBUTES
 BUTTON b1: print, TEXT="Print item";
END
```

2. A dialog instruction with code defining the `ON ACTION` handler with an `ATTRIBUTES` clause:

```
DIALOG ...
   ...
   ON ACTION print
      ATTRIBUTES( ROWBOUND, IMAGE = "printer_2" )
      ...
```

3. The form `ACTION DEFAULTS` section defining:

```
form.per:
ACTION DEFAULTS
  ACTION print (IMAGE="printer_1",
                COMMENT="Print the order",
                ACCELERATORNAME=Control-P,
                CONTEXTMENU=NO)
END
```

4. A global .4ad action defaults file defining:

```
<ActionDefaultList>
  <ActionDefault name="print" text="Print" image="smiley" />
</ActionDefaultList>
```

When the dialog executes, the "print" action will get the following functional attributes:

- `acceleratorName = "control-p"` - from the form `ACTION DEFAULTS` section
- `rowBound = "yes"` - from the dialog `ON ACTION` handler
- `contextMenu = "no"` - from the form `ACTION DEFAULTS` section

The form button (i.e. the action view) will get the following decoration attribute values:

- `text = "Print item"` - from the `BUTTON` form item
- `image = "printer_2"` - from the dialog `ON ACTION` handler
- `comment = "Print the order"` - from the form `ACTION DEFAULTS` section

Action attributes context usage

Action attributes are used to configure functional and decoration properties of actions. The table below lists the possible action atttributes and indicates in what context they can be defined.

Table 285: Action attributes definitions

Attribute	Context			
	Form action view	**Dialog action handler**	**Form action defaults section**	**Global action defaults file (.4ad)**
`ACCELERATOR` See ACCELERATOR action attribute on page 1323.	No	Yes	Yes	Yes
`ACCELERATOR2` See ACCELERATOR2 action attribute on page 1324.	No	No	Yes	Yes
`ACCELERATOR3` See ACCELERATOR3 action attribute on page 1325.	No	No	Yes	Yes
`ACCELERATOR4` See ACCELERATOR4 action attribute on page 1325.	No	No	Yes	Yes
`COMMENT` See COMMENT action attribute on page 1325.	Yes	Yes	Yes	Yes
`CONTEXTMENU` See CONTEXTMENU action attribute on page 1326.	No	Yes	Yes	Yes
`DEFAULTVIEW` See DEFAULTVIEW action attribute on page 1327.	No	Yes	Yes	Yes
`DISCLOSUREINDICATOR`	No	Yes (only for `MENU`)	No	No

Attribute	Context			
	Form action view	**Dialog action handler**	**Form action defaults section**	**Global action defaults file (.4ad)**
See DISCLOSUREINDICATOR action attribute on page 1328.				
`IMAGE` See IMAGE action attribute on page 1328.	Yes	Yes	Yes	Yes
`ROWBOUND` See ROWBOUND action attribute on page 1329.	No	Yes (only for list dialogs)	No	No
`TEXT` See TEXT action attribute on page 1330.	Yes	Yes	Yes	Yes
`VALIDATE` See VALIDATE action attribute on page 1331.	No	Yes (only for input dialogs)	Yes	Yes

Using attributes of action defaults

Purpose of action defaults

Action defaults allow to define default attributes for common action. These defaults can be overwritten with form item attributes, or with dialog action handler attributes (only for default action views).

Centralize action attributes with action defaults, to avoid specifying them in all the source files that define the same action view and action handler. For example, you can specify the default text, image and keyboard accelerator for elements like push buttons, toolbar items, topmenu options.

Common action defaults are typically defined in a global action defaults (`.4ad`) file, while form specific actions are configured with form action defaults in the `ACTION DEFAULTS` section of the `.per` form specification file.

Global action defaults file

Global action defaults are defined in an XML file with the `4ad` extension. By default, the runtime system searches for a file named `default.4ad` in the current directory. If the file does not exist, it searches in the directories defined by the FGLRESOURCEPATH (or DBPATH) environment variable. If no file was found using the environment variable(s), standard action default settings are loaded from the `FGLDIR/lib/default.4ad` file.

> **Important: Global action defaults must be defined in a unique file; you cannot combine several 4ad files.**

If needed, override the default search by loading a specific global action defaults file with the `ui.Interface.loadActionDefaults()` method.

It is possible to use localized strings in action default attributes such as `TEXT` and `COMMENT`, by using `LStr` XML elements:

```
<ActionDefaultList>
```

```
  <ActionDefault name="yes" text="Yes">
     <LStr text="common.yes"/>
  </ActionDefault>
  ...
```

Form specific action defaults

Action defaults can be defined at the form level in the `ACTION DEFAULTS` section. When action defaults are defined in the form file, action views get the attributes defined locally for this form:

```
ACTION DEFAULTS
  ACTION print (TEXT="Print",
                IMAGE="printer",
                COMMENT="Print the current record",
                ACCELERATOR=CONTROL-P)
END
```

Form action defaults can also be defined in a `.4ad` file to be loaded dynamically with the `ui.Form.loadActionDefaults()` method. This method is typically used in form initializers to decorate several application forms without defining an `ACTION DEFAULTS` section in each `.per` file.

It is possible to use localized strings in action default attributes such as `TEXT` and `COMMENT`:

```
ACTION print (TEXT=%"common.print")
```

Action defaults are applied only once

Decoration attributes (like `TEXT`, `IMAGE`) of an action view will automatically be set with the value defined in the action defaults to all new action views of a new created form, if there is no explicitly value specified in the element definition for that attribute. Decoration action default attributes are applied only once, to newly created form elements: Dynamic changes are not reapplied to action views. For example, if you first load a toolbar, then you load a global action defaults file, the attributes of the toolbar items will not be updated with the last loaded action defaults. If you dynamically create action views (like `TopMenu` or `ToolBar`), the action defaults are not applied, so you must set all decoration attributes by hand.

Action defaults and sub-dialog actions

The action default attributes to be applied are selected according to name of the action. In some situations, the action view can be bound to an action handler by specifying a sub-dialog and/or field name prefix. For those views, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name `custlist.append`, it will get the action defaults defined for the `append` action.

Functional attributes

Functional attributes (like `VALIDATE`, `ACCELERATOR`) can only be defined in action defaults, or in `ON ACTION` dialog action handlers with the `ATTRIBUTES` clause. Functional attributes take effect for a given action when the dialog becomes active.

Dialog action handler attributes

Action attributes can be specified at the dialog instruction level for default action views. These action attributes will overwrite the attributes defined in action defaults.

To define dialog-level action attributes for an action, add the `ATTRIBUTES()` clause to `ON ACTION`, with a comma-separated list of action default attributes:

```
ON ACTION print
   ATTRIBUTES (TEXT = "Print",
               COMMENT = "Print the current record",
```

```
                IMAGE = "printer",
                VALIDATE = NO)
```

It is possible to use localized strings in action attributes such as TEXT and COMMENT:

```
ON ACTION print
   ATTRIBUTES (TEXT = %"common.print.label",
               COMMENT = %"common.print.comment",
               ... )
```

Dialog-level action attributes are typically used when the dialog is not related to a specific form, for example with independent MENU dialogs.

If the current form defines explicit action views (buttons in layout, toolbar buttons, topmenu items) with the same name as the ON ACTION handler defining action attributes with the ATTRIBUTES( ) clause, the explicit action views will not get the action attributes defined by the ON ACTION.

Text attribute shows default action view

When creating actions with ON KEY (or COMMAND KEY without a command name in a MENU), the default action view (i.e. button in action frame) is invisible. However, if you define a text action attribute for the corresponding key action, the default action view is made visible.

You can also control the visibility of the default action view with the DEFAULTVIEW action attribute.

Note that it is also possible to set key labels with form attributes (KEY) or with function calls (FGL_SET_KEYLABEL( )), this feature is supported for backward compatibility. Use action default text attributes in new developments.

Defining keyboard accelerators

When using the ON ACTION clause in a dialog instruction, action defaults accelerators are applied in both GUI and TUI mode. For backward compatibility, this is not done in TUI mode when using the ON KEY clause.

The traditional ON KEY clause in a dialog like INPUT implicitly defines the acceleratorName attribute for the action, and the corresponding action default accelerator will be ignored. For example, when you define an ON KEY(F10) block, the first accelerator will be "F10", even if an action default defines an accelerator "F5" for the action "F10". However, you can set other accelerators with the acceleratorName2, acceleratorName3 and acceleratorName4 attributes in action defaults.

> **Important: In TUI mode, actions created with ON KEY do not get accelerators of action defaults; Only actions defined with ON ACTION will get accelerators of Action Defaults.**

In menus, the behavior is a bit different, see the COMMAND and COMMAND KEY clause in MENU.

If no accelerator is specified in action defaults for a predefined action, the runtime system sets one or more default accelerators according to the user interface mode. For example, the accept action will get the Return and Enter keys in GUI mode, but in TUI mode, the Escape key would be used.

If you want to force an action to have no accelerator, you can use none as the accelerator name.

Action attributes list
ACCELERATOR action attribute
The ACCELERATOR is an action attribute defining the primary accelerator key for an action.

Syntax

Syntax 1 (Dialog action handlers)

```
ACCELERATOR = "key"
```

Syntax 2 (`ACTION DEFAULTS` section in form files)

```
ACCELERATOR = key
```

Syntax 3 (Global .4ad action defaults file)

```
acceleratorName = "key"
```

1. *key* defines the accelerator key.

Usage

The `ACCELERATOR` attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

Note that in dialog-specific action attributes, the `ACCELERATOR` must be specified as a string expression.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global `.4ad` file, in the `ACTION DEFAULTS` section of form files, or as dialog action attribute.

Example

```
-- As action handler attribute
ON ACTION print ATTRIBUTES(ACCELERATOR="control-p")

-- As action default
ACTION DEFAULTS
  ACTION print (ACCELERATOR=control-p)
END

-- In a global action defaults file
<ActionDefault name="print" acceleratorName="control-p" ... />
```

ACCELERATOR2 action attribute

The `ACCELERATOR2` is an action attribute defining the secondary accelerator key for an action.

Syntax

Syntax 1 (Dialog action handlers): N/A

Syntax 2 (`ACTION DEFAULTS` section in form files)

```
ACCELERATOR2 = key
```

Syntax 3 (Global .4ad action defaults file)

```
acceleratorName2 = "key"
```

1. *key* defines the accelerator key.

Usage

The `ACCELERATOR2` attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

> **Important:** This attribute is provided for specific cases, consider using only one accelerator per action.

ACCELERATOR3 action attribute
The ACCELERATOR3 is an action attribute defining the third accelerator key for an action.

Syntax

Syntax 1 (Dialog action handlers): N/A

Syntax 2 (ACTION DEFAULTS section in form files)

```
ACCELERATOR3 = key
```

Syntax 3 (Global .4ad action defaults file)

```
acceleratorName3 = "key"
```

1. *key* defines the accelerator key.

Usage

The ACCELERATOR3 attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

> **Important:** This attribute is provided for specific cases, consider using only one accelerator per action.

ACCELERATOR4 action attribute
The ACCELERATOR4 is an action attribute defining the fourth accelerator key for an action.

Syntax

Syntax 1 (Dialog action handlers): N/A

Syntax 2 (ACTION DEFAULTS section in form files)

```
ACCELERATOR4 = key
```

Syntax 3 (Global .4ad action defaults file)

```
acceleratorName4 = "key"
```

1. *key* defines the accelerator key.

Usage

The ACCELERATOR4 attribute defines the keyboard combination that can be pressed by the user to send an action to the program.

> **Important:** This attribute is provided for specific cases, consider using only one accelerator per action.

COMMENT action attribute
The COMMENT attribute defines hint for the user about the action.

Syntax

Syntax 1 (Dialog action handlers and form action defaults)

```
COMMENT = [%]"string"
```

Syntax 2 (Global .4ad action defaults file)

```
comment = "string"
(with optional LStr node for localized strings)
```

1. *string* is the text to display, with the % prefix it is a localized string.

Usage

Use the COMMENT attribute to define a description for the action. This text will typically be displayed as a hint for the corresponding action view.

Consider using localized strings with the %"*string-id*" syntax, if you plan to internationalize your application.

This action attribute can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, as dialog action attribute, or as action view attribute.

Example

```
-- As action handler attribute
ON ACTION print ATTRIBUTES(COMMENT="Prints current record")

-- As action default
ACTION DEFAULTS
  ACTION print (COMMENT="Print current order information")
END

-- In a form buttom, using a localized string id
BUTTON b1: print, COMMENT=%"actions.print.comment";

-- In a global action defaults file with a localized string id
<ActionDefault name="zoom" comment="Opens a zoom window" ... >
   <LStr comment="actions.zoom.comment" />
</ActionDefault>
```

CONTEXTMENU action attribute

The CONTEXTMENU attribute defines whether a context menu option must be displayed for an action.

Syntax

Syntax 1 (Dialog action handlers and form action defaults)

```
CONTEXTMENU = [ AUTO | YES | NO ]
```

Syntax 2 (Global .4ad action defaults file)

```
contextMenu = [ "yes" | "no" | "auto" ]
```

Usage

CONTEXTMENU is an action attribute defining whether the context menu option must be displayed for an action.

1. NO indicates that no context menu option must be displayed for this action.
2. YES indicates that a context menu option must always be displayed for this action, if the action is visible.
3. AUTO means that the context menu option is displayed if no explicit action view is used for that action and the action is visible.

The default is YES.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, or as dialog action attribute.

Example

```
-- As action handler attribute
ON ACTION zoom ATTRIBUTES(CONTEXTMENU=YES)

-- As action default
ACTION DEFAULTS
  ACTION zoom (CONTEXTMENU=YES)
END

-- In a global action defaults file
<ActionDefault name="zoom" contextMenu="yes" ... />
```

DEFAULTVIEW action attribute

The DEFAULTVIEW attribute defines if a default view (a button) must be displayed for a given action.

Syntax

Syntax 1 (Dialog action handlers and form action defaults)

```
DEFAULTVIEW = [ AUTO | YES | NO ]
```

Syntax 2 (Global .4ad action defaults file)

```
defaultView = [ "yes" | "no" | "auto" ]
```

Usage

DEFAULTVIEW is an action attribute defining whether the default action view (a button) must be displayed for an action.

- NO indicates that no default action view must be displayed for this action.
- YES indicates that a default action view must always be displayed for this action, if the action is visible.
- AUTO means that a default action view is displayed if no explicit action view is used for that action and the action is visible.

The default is AUTO.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, or as dialog action attribute.

Example

```
-- As action handler attribute
ON ACTION zoom ATTRIBUTES(DEFAULTVIEW=YES)

-- As action default
ACTION DEFAULTS
  ACTION zoom (DEFAULTVIEW=YES)
END

-- In a global action defaults file
```

```
<ActionDefault name="zoom" defaultView="yes" ... />
```

DISCLOSUREINDICATOR action attribute
The DISCLOSUREINDICATOR attribute a drill-down decoration to an action.

Syntax

(only in MENU action handlers)

```
DISCLOSUREINDICATOR
```

Usage

DISCLOSUREINDICATOR is an action attribute defining whether a disclosure indicator must be shown for the default view (a button) of an action.

> **Important:** This feature is only for mobile platforms.

A disclosure indicator gives a visual hint to the user, to show that the selection of the action will drill down in the application screens.

The DISCLOSUREINDICATOR attribute is typically used in a MENU instruction, for options that open a sub-menu.

The rendering of a disclosure indicator depends from the front-end platform standards. On iOS devices, buttons will show a typical > icon on the right.

This attribute can only be specified in a MENU dialog, as action attribute in the ATTRIBUTES( ) clause of ON ACTION handlers, and applies to the actions defined by the current dialog in the current window.

Note however, that form buttons can get a DISCLOSUREINDICATOR attribute, as an action view decoration.

Example

```
MENU ...
   ...
   ON ACTION details ATTRIBUTES(DISCLOSUREINDICATOR)
      CALL show_customer_details(cust_rec.cust_no)
   ...
```

IMAGE action attribute
The IMAGE attribute defines the image resource to be displayed for the action.

Syntax

Syntax 1 (Dialog action handlers and form action defaults)

```
IMAGE = "resource"
```

Syntax 2 (Global .4ad action defaults file)

```
image = "resource"
```

1. *resource* defines the file name, path or URL to the image source.

Usage:

The IMAGE attribute is used to define the image resource for the action view such a BUTTON , BUTTONEDIT or a TOOLBAR button.

For more details about image resource specification, see Providing the image resource on page 784.

This action attribute can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, as dialog action attribute, or as action view attribute.

Example

```
-- As action handler attribute
ON ACTION print ATTRIBUTES(IMAGE="printer")

-- As action default
ACTION DEFAULTS
  ACTION print (IMAGE="printer")
END

-- In a form buttonedit or button
BUTTONEDIT f001 = FORMONLY.field01, IMAGE = "zoom";
BUTTON b01: open_file, IMAGE = "buttons/fileopen";
BUTTON b02: accept, IMAGE = "http://myserver/images/accept.png";
```

ROWBOUND action attribute

The ROWBOUND attribute defines if the action is related to the row context of a record list.

Syntax

(only in action handlers of record list dialog)

```
ROWBOUND
```

Usage

The ROWBOUND is typically used in a DISPLAY ARRAY or INPUT ARRAY dialog action handler, when the action depends from the row context. The actions marked with this attribute will be automatically enabled/disabled according the current row existence, and rendered in a special way according to the front-end platform standards.

Important: This feature is only for mobile platforms.

The ROWBOUND attribute was mainly introduced for mobile applications, when using a TABLE container to get a list view: Actions marked with this attribute will be rendered in a native manner on the mobile device.

If a default action view is displayed for the action, it will be automatically hidden when no current row context is available.

This attribute can only be specified in a list handling dialog, as action attribute in the ATTRIBUTES( ) clause of ON ACTION handlers, and applies to the actions defined by the current dialog in the current window.

Default actions such as the delete action when using an ON DELETE modification trigger will automatically get the ROWBOUND attribute, to be available only when at least one row exists in the list. Therefore, the ROWBOUND attribute cannot be specified for such DISPLAY ARRAY modification triggers.

Example

```
DISPLAY ARRAY ...
   ...
```

```
   ON ACTION print ATTRIBUTES(ROWBOUND)
      CALL print_customer_info(arr_curr())
   ...
```

TEXT action attribute
The TEXT attribute defines the label associated to the action.

Syntax

Syntax 1 (Dialog action handlers and form action defaults)

```
TEXT = [%]"string"
```

Syntax 2 (Global .4ad action defaults file)

```
text = "string"
(with optional LStr node for localized strings)
```

1. *string* defines the label for the action, with the % prefix it is a localized string.

Usage

The TEXT attribute is used to define the label associated to an action, for example for a CHECKBOX form field or a BUTTON action view.

Consider using localized strings with the %"*string-id*" syntax, if you plan to internationalize your application.

This action attribute can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, as dialog action attribute, or as action view attribute.

Example

```
-- As action handler attribute
ON ACTION print ATTRIBUTES(TEXT="Print")

-- As form action default
ACTION DEFAULTS
  ACTION print (TEXT="Print")
END

-- As a CHECKBOX label
CHECKBOX cb01 = FORMONLY.checkbox01,
                TEXT="OK" ... ;

-- As a BUTTON label, using a localized string id
BUTTON b1: print, TEXT=%"actions.print.label";

-- In a global action defaults file with a localized string id
<ActionDefault name="zoom" text="Zoom" ... >
   <LStr text="actions.zoom.label" />
</ActionDefault>
```

VALIDATE action attribute
The VALIDATE action attribute defines the data validation level for a given action.

Syntax

Syntax 1 (Dialog action handlers and form action defaults)

```
VALIDATE = NO
```

Syntax 2 (Global .4ad action defaults file)

```
validate = "no"
```

Usage

When the VALIDATE action attribute is set to NO, it indicates that no data validation must occur for this action. However, current input buffer contains the text modified by the user before triggering the action.

This attribute applies to the actions defined by the current dialog in the current window. It can be specified as action default attribute in a global .4ad file, in the ACTION DEFAULTS section of form files, or as dialog action attribute.

Example

```
-- As action handler attribute
ON ACTION undo ATTRIBUTES(VALIDATE=NO)

-- As action default
ACTION DEFAULTS
  ACTION undo (VALIDATE=NO)
END

-- In a global action defaults file
<ActionDefault name="undo" validate="nos" ... />
```

Data validation at action invocation
The validate action default attribute controls field validation when an action is fired.

When using the UNBUFFERED mode of interactive instructions such asINPUT or DIALOG, if the user triggers an action, the current field data is checked and loaded in the target variable bound to the form field. For example, if the user types a wrong date (or only a part of a date) in a field using a DATE variable and then clicks on a button to invoke an action, the runtime system will display an invalid input error and will not execute the ON ACTION block corresponding to the button.

To prevent data validation for some actions, use the validate action default attribute. This attribute instructs the runtime not to copy the input buffer text into the program variable (requiring input buffer text to match the target data type).

```
ACTION DEFAULTS
   ...
   ACTION zoom ( ... VALIDATE = NO ... )
   ...
END
```

This is especially needed in DIALOG instructions; in singular dialogs like INPUT, predefined actions like cancel do not validate the current field value when UNBUFFERED mode is used.

The validate action default attribute can be set in the global action default file, or at the form level, with the VALIDATE attribute in a line of the ACTION DEFAULTS section.

Enabling and disabling actions

By default, dialog actions are enabled, however an action should be disabled when not allowed in the current context.

Dialog actions are enabled to let the user invoke the action handler (`ON ACTION`/`COMMAND`) by clicking on the corresponding action view (button) or by pressing its accelerator key. In most situations, actions remain active during the whole dialog execution. However, to follow GUI standards, actions must be disabled when not allowed in the current context. For example, a print action should be disabled if no record is currently shown in the form. After a database query, when the form is filled with a given record, the print action can be activated.

Depending on the front-end ergonomics, the visual result of disabling an action can be different. On desktop front-ends, the action views (buttons) are typically grayed, indicating that the action is there but cannot be triggered. On other front-ends such as some mobile devices, the action view might be hidden, for layout reasons (there is not much space on a mobile device screen).

During a dialog instruction, enable or disable an action with the `setActionActive()` method of the `ui.Dialog` built-in class. This method takes the name of the action (in lowercase letters) and a boolean expression (`0` or `FALSE`, `1` or `TRUE`) as arguments.

```
  BEFORE INPUT
    CALL DIALOG.setActionActive( "zoom", FALSE )
```

Consider centralizing action activation / deactivation in a setup function specific to the dialog, passing the `DIALOG` object as the parameter. Centralizing the action activation defines the rules in a single location:

```
FUNCTION cust_dialog_setup(d)
  DEFINE d ui.Dialog
  DEFINE can_modify BOOLEAN
  LET can_modify = (cust_rec.is_new OR user_info.is_admin)
  CALL d.setActionActive("update", can_modify)
  CALL d.setActionActive("delete", can_modify)
  ...
END FUNCTION
```

Some predefined dialog actions such as insert / append / delete of `INPUT ARRAY` are automatically enabled/disabled according to the context. For example, if the maximum number of rows (`MAXCOUNT`) is reached in an `INPUT ARRAY`, insert and append actions are disabled.

When the action activation depends on the focus being in a specific field, consider using the `INFIELD` clause of `ON ACTION` to automatically disable an action if the focus leaves the specified field.

Inside a `DIALOG` block, actions can be defined a different levels, and may need to be identified with the sub-dialog prefix, when you invoke the `ui.Dialog.setActionActive()` method outside of the context of the sub-dialog. In the next example, the `check_row` action must be prefixed by the `s_ord` sub-dialog name, because `setActionActive()` is called from the `INPUT BY NAME` sub-dialog context, to disable an action from the `DISPLAY ARRAY` sub-dialog:

```
DIALOG ATTRIBUTES(UNBUFFERED)
   DISPLAY ARRAY a_ord TO s_ord.*
      -- sub-dialog-level action
      ON ACTION check_row
         ...
   END DISPLAY
   ...
   INPUT BY NAME rec.* ...
      ON CHANGE consolidation
          -- Must use sub-dialog name to identify the check_row action:
          CALL DIALOG.setActionActive( "s_ord.check_row", FALSE )
      ...
   END INPUT
```

```
END DIALOG
```

Hiding and showing default action views
If needed, default action views can be hidden or shown.

When an action is rendered with a default action view (for example, by a button on the action frame of a destop front-end, or in the top action panel on a mobile front-end), it is sometimes required to hide the action button when the operation is not possible and there is not much space on the screen.

> **Important:** Hiding an action will only make the default action view invisible, if there is a keyboard accelerator associated to the action, it can still fire the action. Consider disabling the action completely with `setActionActive()`.

During a dialog instruction, shown or hide an action with the `setActionHidden()` method of the `ui.Dialog` built-in class. This method takes the name of the action (in lowercase letters) and an integer boolean expression (`0` or `FALSE`, `1` or `TRUE`) as arguments.

```
  BEFORE INPUT
    CALL DIALOG.setActionHidden( "zoom", 1 )
```

Consider centralizing action visibility control in a setup function specific to the dialog, passing the `DIALOG` object as the parameter. Centralizing the action activation defines the rules in a single location:

```
FUNCTION cust_dialog_setup(d)
  DEFINE d ui.Dialog
  DEFINE can_modify BOOLEAN
  LET can_modify = (cust_rec.is_new OR user_info.is_admin)
  CALL d.setActionActive("update", can_modify)
  CALL d.setActionHidden("update", IIF(can_modify,0,1))
  CALL d.setActionActive("delete", can_modify)
  CALL d.setActionHidden("delete", IIF(can_modify,0,1))
  ...
END FUNCTION
```

Pay attention to multi-level action definitions inside a `DIALOG` block: Inside a `DIALOG` block, actions must be hidden/shown with the `ui.Dialog.setActionHidden()` method by specifying a simple action name:

```
DIALOG ATTRIBUTES(UNBUFFERED)
     ...
   BEFORE DIALOG
     CALL DIALOG.setActionHidden( "print", 1 )
     ...
   ON ACTION query
     -- query the database and fill the record
     ...
     CALL DIALOG.setActionHidden( "print", (cust_id IS NULL) )
     ...
END DIALOG
```

Sub-dialog actions in procedural DIALOG blocks
This topic describes how action are differentiated with handlers defined in a procedural DIALOG block.

We distinguish *dialog actions* from *sub-dialog actions*: When the `ON ACTION` handler is defined at the same level as a `BEFORE DIALOG` control block, it is a dialog action, and the action name is a simple identifier as in singular interactive instructions:

```
action-name
```

When the `ON ACTION` handler is defined inside a sub-dialog, or if the action is an implicit action such as `insert` in `INPUT ARRAY`, it is a sub-dialog action, and the action name gets the name of the sub-dialog as the prefix to identify the sub-dialog action with a unique name:

```
sub-dialog-name.action-name
```

The `INPUT ARRAY` and `DISPLAY ARRAY` sub-dialogs are implicitly identified with the screen-record name defined in the form. For `INPUT` and `CONSTRUCT` sub-dialogs, the sub-dialog identifier can be specified with the `NAME` attribute.

The next example defines two 'check' action in different sub-dialog contexts, and a 'close' action at the dialog level:

```
DIALOG
  INPUT BY NAME ... ATTRIBUTES (NAME = "cust")
    ON ACTION check                   -- sub-dialog action "cust.check"
    ...
  END INPUT
  DISPLAY ARRAY arr_orders TO sr_ord.*
    ...
    ON ACTION check                   -- sub-dialog action "sr_ord.check"
    ...
  END DISPLAY
  BEFORE DIALOG
    ...
  ON ACTION close                      -- dialog action "close"
    ...
END DIALOG
```

By using the sub-dialog identifier in form definition files, you can bind action views to specific sub-dialog actions. Action views bound to sub-dialog actions with qualified sub-dialog action names will always be active, even if the focus is not in the sub-dialog of the action. You typically use fully-qualified sub-dialog actions names for buttons in the form body or in topmenu options. However, it does not make much sense to use this technique for toolbar buttons, where buttons must be enabled/disabled according to the context.

```
TOOLBAR
   ...
   ITEM append
   ...
END

TOPMENU
   ...
   GROUP orders (TEXT="Orders")
     COMMAND sr_ord.append
   ...
END

LAYOUT
GRID
{
   ...
   [b002  ]
}
END
END

ATTRIBUTES
BUTTON b002: sr_ord.append;
END
```

If you bind an action view with a simple action name (without the sub-dialog prefix), the action view will be attached to any sub-dialog action with the matching name. This is especially useful for common actions such as the implicit insert / append / delete actions created by `INPUT ARRAY`, when the dialog handles multiple editable lists. Bind toolbar buttons to these actions without the sub-dialog prefix; the buttons will apply to the current list that has the focus. The action views bound to sub-dialog actions without the sub-dialog qualifier will automatically be enabled or disabled when entering or leaving the group of fields controlled by the sub-dialog (i.e. typical navigation buttons in the toolbar will be disabled if the focus is not in a list).

If a sub-dialog action is invoked when the focus is not in the sub-dialog of the action, the focus will automatically be given to the first field of the sub-dialog, before executing the user code defined in the `ON ACTION` clause. This will trigger the same validation rules and control blocks as if the user had selected the first field of the sub-dialog by hand.

When using `DIALOG.setActionActive()` (or any method that takes an action name as parameter), you can specify the action name with or without a sub-dialog identifier. If you qualify the action with the sub-dialog identifier, the sub-dialog action is clearly identified. If you don't specify a sub-dialog prefix, the action will be identified based on the focus context - when the focus is in the sub-dialog of the action, non-qualified action names identify the local sub-dialog action; otherwise, they identify a dialog action if one exists with the same name. Disabling an action by the program with `setActionActive()`, will take precedence over the built-in activation rules (i.e. if the action is disabled by the program, the action will not be activated when entering the sub-dialog).

For action views bound to sub-dialog actions with qualifiers, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name "`custlist.append`", it will get the action defaults defined for the "`append`" action.

Field-specific actions (INFIELD clause)
Using the `INFIELD` clause of `ON ACTION` provides automatic action activation when a field gets the focus.

The `ON ACTION` interaction block of `INPUT`, `CONSTRUCT` and `INPUT ARRAY` (as singular dialogs or sub-dialogs in `DIALOG` instruction), can be specified with the `INFIELD` *`field-name`* clause. With this clause, the action will only be active when the focus is in one of the fields. The same action name can be used for several fields.

```
INPUT ARRAY custarr WITHOUT DEFAULTS FROM sr_cust.*
  ON ACTION zoom INFIELD cust_city
    LET custarr[arr_curr()].cust_city = zoom_city()
  ON ACTION zoom INFIELD cust_state
    LET custarr[arr_curr()].cust_state = zoom_state()
END INPUT
```

Actions defined with the `INFIELD` *`field-name`* clause can be identified with the field name as prefix:

```
field-name.action-name
```

Bind action views with field name prefix to identify the action specifically to a field, or use the action name only. Without the field name prefix, the action view is enabled and disabled automatically according to the current field. When binding the action view with the fully-qualified name including the field name prefix, the action view will always be active, and the focus will jump to the field if the action is fired.

Actions defined in sub-dialogs of the `DIALOG` instruction get the name of the sub-dialog as prefix. If `ON ACTION` *`action-name`* `INFIELD` *`field-name`* is used in a sub-dialog, the action object name is prefixed with the name of the sub-dialog, followed by the name of the field. The fully-qualified action name will be:

```
sub-dialog-name.field-name.action-name
```

When the field-specific action is invoked (for example by a button of the toolbar bound with the fully-qualified action name) and if the field does not have the focus, the runtime system first selects that field before executing the code of the `ON ACTION INFIELD` block. The field selection forces data validation and `AFTER FIELD` of the current field, followed by `BEFORE FIELD` of the target field associated to the action.

It's still possible to enable and disable field-specific action objects by the program using the `DIALOG.setActionActive()` method. When specifying a fully-qualified action name with the field name prefix, that field-specific action will be enabled or disabled. When disabled by the `setActionActive()` method, the corresponding action views will always be disabled, even if the field has the focus. If you do not specify a fully-qualified name in the method call, and if several actions are defined with the same action name in different sub-dialogs and/or using the `INFIELD` clause, the method will identify the action according to the current focus context. For example, if you define `ON ACTION zoom INFIELD cust_city` and `ON ACTION zoom INFIELD cust_addr`, when the focus is in `cust_city`, a call to `DIALOG.setActionActive("zoom", FALSE)` will disable the action specific to the `cust_city` field.

Fields can be enabled or disabled dynamically with the `DIALOG.setFieldActive()` method. If an `ON ACTION INFIELD` is declared on a field and if you enable/disable the field dynamically, then the field-specific action (and corresponding action views in the form) will be enabled or disabled accordingly.

For action views bound to field actions with qualifiers, the action defaults defined with the corresponding action name will be used to set the attributes with the default values. In other words, the prefix will be ignored. For example, if an action view is defined with the name "`cust_addr.check`", it will get the action defaults defined for the "`check`" action.

Multilevel action conflicts

Actions can be defined at two levels in a singular dialog, and three levels in the context of a `DIALOG` block:

1. Dialog level
2. Sub-dialog level (procedural `DIALOG` only)
3. Field level (`ON ACTION` with `INFIELD` clause)

It is not good practice to use the same action name at different levels of a dialog: This makes action view bindings and action handling (i.e. enabling / disabling) very complex, because there are many possible combinations. Therefore, when using the same action name at different dialog levels, the fglcomp compiler will raise a warning -8409. However, it is legal to use the same action name for a given level of action handlers in a sub-dialogs or for field-actions. For example, using the "zoom" action name for multiple `ON ACTION INFIELD` handlers is a common practice.

When binding action views with full qualified names, the `ON ACTION` handler is clearly identified, and the corresponding user code will be executed. However, when you do not specify the complete prefix of a sub-dialog or field action, the runtime system searches for the best `ON ACTION` handler to be executed, according to the current focus context.

Take for example a `DIALOG` instruction defining three `ON ACTION print` handlers at the dialog, sub-dialog and field level:

```
DIALOG
  INPUT BY NAME ... ATTRIBUTES (NAME = "cust")
    ...
    ON ACTION print INFIELD cust_name -- field-level action (1)
      ...
    ON ACTION print                   -- sub-dialog-level action (2)
      ...
  END INPUT
  ...
  ON ACTION print                     -- dialog-level action (3)
    ...
END DIALOG
```

The action views of the form will behave as follows:

- Action views bound with the name "`print`" will always be active, and invoke the `ON ACTION print` handler corresponding to the current focus context:
  - (1) is invoked if the focus is in the `cust_name` field.
  - (2) is invoked if the focus is in the `cust` sub-dialog, but not in `cust_name` field.
  - (3) is invoked if the focus is in another sub-dialog as `cust` sub-dialog.
- Action views bound with the name "`cust.print`" will always be active, even if the focus is not the `cust` sub-dialog, and invoke the `ON ACTION print` handler according to the current focus context:
  - (1) is invoked if the focus is in the `cust_name` field.
  - (2) is invoked if the focus is in the `cust` sub-dialog, but not in `cust_name` field.
- Action views bound with the name "`cust.cust_name.print`" will always be active, and invoke the `ON ACTION print INFIELD cust_name` handler after giving the focus to the `cust_name` field.

If the first field of a sub-dialog defines an `ON ACTION INFIELD` with the same action name as a sub-dialog action, and the focus is not in that sub-dialog when the user selects an action view bound with the name *sub-dialog-name.action-name*, the runtime system gives the focus to the first field of the sub-dialog. This field becomes the current field, and the runtime system executes the field-specific action handler instead of the sub-dialog action handler.

To avoid mistakes and complex combinations, you should use specific action names for each dialog level.

Action display in the contextual menu

The `CONTEXTMENU` action default attribute allows to control action visibility in the contextual menu.

Some front-ends can display a *contextual menu*, with all the active actions that are possible in the current form. Displaying all actions might not be adapted to your needs. To control if an action must be displayed in the context menu, set the `CONTEXTMENU` attribute in action defaults. Values for `CONTEXTMENU` can be `YES`, `NO` and `AUTO`.

```
ACTION DEFAULTS
   ...
   ACTION insert ( ... CONTEXTMENU = YES ... )
   ACTION append ( ... CONTEXTMENU = YES ... )
   ACTION delete ( ... CONTEXTMENU = YES ... )
   ...
   ACTION validate_order ( ... CONTEXTMENU = NO ... )
   ...
END
```

Implementing the close action

The close action is a predefined action dedicated to close graphical windows (for example, with the X cross button).

In graphical applications, windows can be closed by the user, for example by pressing Alt+F4 or by clicking the cross button in the upper-left corner of the window. A predefined action is dedicated to this specific event, named "close".

When the end user closes a graphical window, the program gets a close action.

Note that the default action view (i.e. button in the action frame) of the close action is hidden.

The close action in DIALOG dialogs

When executing a `DIALOG` instruction, the close action executes the `ON ACTION close` block, if defined. Otherwise, the close action is mapped to the cancel action if an `ON ACTION cancel` handler is defined.

If neither `ON ACTION close`, nor `ON ACTION cancel` are defined, nothing will happen if the user tries to close the window with the X cross button or an `ALT+F4` keystroke.

The INT_FLAG register will not be set in the context of DIALOG.

The close action in form input singular dialogs

When an ON ACTION close handler is defined in an INPUT, INPUT ARRAY, CONSTRUCT, DISPLAY ARRAY or PROMPT interactive instruction, the handler code will be executed if the close action is fired.

If no explicit ON ACTION close handler is defined, the close action acts the same as the cancel predefined action. So by default when the user clicks the X cross button in a window, the interactive instruction stops and INT_FLAG is set to 1.

If there is an explicit ON ACTION cancel block defined, INT_FLAG is set to 1 and the user code under ON ACTION cancel will be executed.

If the CANCEL=FALSE option is set, no cancel and no close action will be created, and you must write an ON ACTION close handler to proceed with the close action. In this case, the INT_FLAG register will not be set when the close action is invoked.

The close action in MENU dialogs

When an ON ACTION close handler is defined in a MENU statement, the handler code will be executed if the close action is fired.

If no explicit ON ACTION close action handler is defined, the code of the COMMAND KEY(INTERRUPT) or ON ACTION cancel will be executed, if defined. if neither COMMAND KEY(INTERRUPT) nor ON ACTION cancel are defined, nothing happens and the program stays in the MENU instruction. Regarding the close action, the value of INT_FLAG is undefined in a MENU instruction.

The close action on mobile devices

When displaying on a mobile device, the close action is rendered differently according to the type of mobile platform:

- On Android™, the close action is mapped to the [Back] button (it is not rendered in the action panel)
- On iOS, there is no [Back] button concept and the close action is rendered as a regular action.

For more details, see Rendering default action views on mobile on page 1279.

Example

You typically implement a close action handler to open a confirmation dialog box as in following example:

```
INPUT BY NAME cust_rec.*
  ...
  ON ACTION close
    IF msg_box_yn("Are you sure you want to close this window?")
 == "y" THEN
      EXIT INPUT
    END IF
  ...
END INPUT
```

Predefined actions

Genero predefines some action names for common operations of interactive instructions.

Predefined actions are different from user-defined action, in the sense that the name of a predefined action is reserved, and the action may has an ON ACTION handler, while user-defined actions have a specific name, and must be implemented with an ON ACTION handler.

There are three types of predefined actions:

- *Automatic actions*: actions that are automatically created and handled by the program dialog, like `accept`, `cancel`, `insert`.
- *Special actions*: actions with a special usage, that can be invoked asynchronously or automatically by the front-end, like `interrupt`, `dialogtouched`.
- *Local actions*: actions that are handled on the front end side, without program interaction, such as `editcopy`.

Default decoration attributes and keyboard shortcuts are defined with action defaults, like for user-defined actions.

Automatic and local actions with same name

Some predefined actions exist as both automatic actions and as local actions (for example, editcopy). The automatic actions are created according to the dialog context. If an automatic action has to be defined and if a local action exists with the same name, the automatic action takes precedence over the local action.

For example, if the dialog context requires a editcopy runtime action, the local editcopy action will not be handled by the front end. Identical action names are used for automatic and local action to bind with the same action view. For example, the same toolbar button created with the editcopy name will trigger the automatic action or the local action, according to the context.

Overwriting predefined actions with ON ACTION

If you define your own `ON ACTION` handler with the name of a predefined action, the default action processing is bypassed and the program code is executed instead.

The next code example defines an `ON ACTION` clause with the `accept` predefined action name:

```
INPUT BY NAME customer.*
  ON ACTION accept
    ...
END INPUT
```

In this case, the default behavior of the automatic accept action is not performed; the user code is executed instead.

Local actions can be overwritten in the same manner, however, this is not recommended (use your own action names).

Predefined actions enabled according to context

Some predefined actions (such as insert, append and delete in `INPUT ARRAY`) are enabled and disabled automatically by the dialog according to the context (for example, when a static array used by the `INPUT ARRAY` is full, the insert and append actions get disabled).

Even when overwriting such actions with your own action handler, the runtime system will continue to enable and disabled the actions automatically.

You should not overwrite predefined actions.

Binding action views to predefined actions

As for user-defined actions, if you design forms with action views using predefined action names, they will automatically attach themselves to the actions of the interactive instructions.

It is also possible to define default images, texts, comments and accelerator keys in the action defaults resource file for the predefined actions.

List of predefined actions

Table 286: Automatic actions (automatically created by dialogs)

Action Name	Description	ON ACTION block is required	Context
`accept`	Validates the current interactive instruction (singular dialogs only)	can overwrite	(1)
`cancel`	Cancels the current interactive instruction (singular dialogs only)	can overwrite	(1)
`close`	Triggers a cancel key in the current interactive instruction (by default)	can overwrite	(7)
`insert`	Inserts a new row before current row	can overwrite	(2)
`append`	Appends a new row at the end of the list	can overwrite	(2)
`delete`	Deletes the current row	can overwrite	(2)
`find`	Opens the fglfind dialog window to let the user enter a search value, and seeks to the row matching the value	can overwrite	(4)
`findnext`	Seeks to the next row matching the value entered during the fglfind dialog	can overwrite	(4)
`nextrow`	Moves to the next row (only if list using one flat screen record)	can overwrite	(8)
`prevrow`	Moves to the previous row (only if list using one flat screen record)	can overwrite	(8)
`firstrow`	Moves to the first row (only if list using one flat screen record)	can overwrite	(8)
`lastrow`	Moves to the last row (only if list using one flat screen record)	can overwrite	(8)
`help`	Shows the help topic defined by the `HELP` clause	can overwrite	(1)
`editcopy`	Copy selected rows (or current row if MRS is off) to the clipboard	can overwrite	(9)

Table 287: Special actions (special behavior)

Special Action Name	Description	ON ACTION block is required	Context
`browser_back`	Sent when the user hits the back button in a web browser (web front-end only).	yes	(7)
`browser_forward`	Sent when the user hits the forward button in a web browser (web front-end only).	yes	(7)
`dialogtouched`	Sent by the front end each time the user modifies the value of a field. For more details, see Immediate detection of user changes on page 1267.	yes	(7)
`interrupt`	Sends an interruption request to the program when processing. For more details, see User interruption handling on page 1252.	no	(5)
`windowresized`	On Mobile devices, this action is sent when changing the orientation of the device. On other front-ends, it is sent when the current active window is resized. For more details, see Adapting to viewport changes on page 1003.	yes	(6)
`notificationpushed`	On Mobile devices, this action is fired when receiving a push notification message. See getRemoteNotifications on page 1930	yes	(6)

Table 288: Local actions (handled by the front end)

Local Action Name	Description	ON ACTION block is required	Context
`editcopy`	Copies the current selected text to the clipboard	can overwrite	(7)

Local Action Name	Description	ON ACTION block is required	Context
`editcut`	Copies the current selected text to the clipboard and removes the text from the current input widget	can overwrite	(7)
`editpaste`	Pastes the clipboard content to the current input widget	can overwrite	(7)
`nextfield`	Moves to the next field in the form	can overwrite	(3)
`prevfield`	Moves to the previous field in the form	can overwrite	(3)
`nextrow`	Moves to the next row in the list	can overwrite	(4)
`prevrow`	Moves to the previous row in the list	can overwrite	(4)
`firstrow`	Moves to the first row in the list	can overwrite	(4)
`lastrow`	Moves to the last row in the list	can overwrite	(4)
`nextpage`	Moves to the next page in the list	can overwrite	(4)
`prevpage`	Moves to the previous page in the list	can overwrite	(4)
`nexttab`	Moves to the next page in the folder	can overwrite	(6)
`prevtab`	Moves to the previous page in the folder	can overwrite	(6)

Context column descriptions

1. `CONSTRUCT`, `INPUT`, `PROMPT`, `INPUT ARRAY` and `DISPLAY ARRAY`.
2. `INPUT ARRAY` only.
3. `CONSTRUCT`, `INPUT` and `INPUT ARRAY`.
4. `INPUT ARRAY` and `DISPLAY ARRAY`.
5. Only possible when no interactive instruction is active.
6. Possible in any kind of interactive instruction (`MENU` included).
7. `DIALOG`, `CONSTRUCT`, `INPUT`, `PROMPT`, `INPUT ARRAY` and `DISPLAY ARRAY`.
8. `INPUT ARRAY` and `DISPLAY ARRAY` on flat screen-record.
9. `DISPLAY ARRAY` only.

Keyboard accelerator names

Virtual keys

Virtual keys are the key names that can be used in program instructions such as `ON KEY` and `COMMAND KEY`.

An `ON KEY` block defines one to four different action objects that will be identified by the key name in lowercase (`ON KEY(F5,F6) = creates Action f5 + Action f6`). Each action object will get an `acceleratorName` attribute assigned. In GUI mode, Action defaults are applied for `ON KEY` actions by using the name of the key. You can define secondary accelerator keys, as well as default decoration attributes like button text and image, by using the key name as action identifier. The action name is always in lowercase letters.

Check carefully the `ON KEY CONTROL-?` statements because they may result in having duplicate accelerators for multiple actions due to the accelerators defined by action defaults. Additionally, `ON KEY` statements used with `ESC`, `TAB`, `UP`, `DOWN`, `LEFT`, `RIGHT`, `HELP`, `NEXT`, `PREVIOUS`, `INSERT`, `CONTROL-M`, `CONTROL-X`, `CONTROL-V`, `CONTROL-C` and `CONTROL-A` should be avoided for use in GUI programs, because it's very likely to clash with default accelerators defined in the Action Defaults.

By default, `ON KEY` actions are not decorated with a default button in the action frame (i.e. default action view). You can show the default button by configuring a `text` attribute with the action defaults.

Table 289: Names of keys to be referenced in programs

Key Name	Description
`ACCEPT`	The validation key.
`INTERRUPT`	The interruption key.
`ESC` or `ESCAPE`	The ESC key (not recommended, use `ACCEPT` instead).
`TAB`	The TAB key (not recommended).
`Control-`*`char`*	A control key where *char* can be any character except A, D, H, I, J, K, L, M, R, or X.
`F1` through `F255`	A function key.
`DELETE`	The key used to delete a new row in an array.
`INSERT`	The key used to insert a new row in an array.
`HELP`	The help key.
`LEFT`	The left arrow key.
`RIGHT`	The right arrow key.
`DOWN`	The down arrow key.
`UP`	The up arrow key.
`PREVIOUS` or `PREVPAGE`	The previous page key.
`NEXT` or `NEXTPAGE`	The next page key.

Accelerator keys

Accelerators keys are attributes defining the keyboard shortcuts for actions.

Keyboard accelerators can be defined at several level in the form files or in action defaults. You can define up to four accelerator keys for the same action in action defaults, by setting the `acceleratorName`, `acceleratorName2`, `acceleratorName3` and `acceleratorName4` attributes.

If no accelerators are defined in the action defaults, the runtime system sets default accelerators for predefined actions, according to the user interface mode. For example, the accept action will get the `Return` and `Enter` keys in GUI mode, but gets the `Escape` key in TUI mode.

Accelerators can also be defined in on the program in the attribute list of the `ON ACTION` interaction block.

If one of the user-defined actions uses an accelerator that would normally be used for a predefined action, the runtime system does not set that accelerator for the predefined action. For example (in GUI mode), if you define an `ON ACTION quit` with an action default using the accelerator "`Escape`", the "cancel" predefined action will not get the "`Escape`" default accelerator. In this case, user settings take precedence over defaults.

Text edition and navigation accelerators such as `Home` and `End` are usually local to the widget. According to the context, such accelerators might be eaten by the graphical widget and will not invoke the action bound to the corresponding accelerator defined in the action defaults. For example, even if the action defaults for the "firstrow" action defines the `Home` accelerator, when using an `INPUT ARRAY`, the Home key will jump to the beginning of the edit field, not the first row of the list.

If you want to force an action to have no accelerator, specify "`none`" as the accelerator name.

This table lists all the keyboard accelerator names:

Table 290: Keyboard accelerator names

Accelerator Name	Description
`none`	Special name indicating the runtime system must not set any default accelerator for the action.
`0-9`	Decimal digits from 0 to 9
`A-Z`	Letters from A to Z
`F1-F35`	The functions keys
`BackSpace`	The BACKSPACE key (do not confuse with DELETE key)
`Delete`	The DELETE key (navigation keyboard group)
`Down`	The DOWN key (arrow keyboard group)
`End`	The END key (navigation keyboard group)
`Enter`	The ENTER key (numeric keypad, see Note)
`Escape`	The ESCAPE key
`Home`	The HOME key (navigation keyboard group)
`Insert`	The INSERT key (navigation keyboard group)
`Left`	The LEFT key (arrow keyboard group)
`Minus`	The MINUS sign key (-)
`Next`	The NEXT PAGE key (navigation keyboard group)
`Prior`	The PRIOR PAGE key (navigation keyboard group)
`Return`	The RETURN key (alphanumeric keypad, see Note)
`Right`	The RIGHT key (arrow keyboard group)
`Space`	The SPACEBAR key

Accelerator Name	Description
`Tab`	The TABULATION Key
`Up`	The UP key (arrow keyboard group)

> **Note:** The "`Enter`" key represents the ENTER key available on the numeric keypad of standard keyboards, while "`Return`" represents the RETURN key of the alphanumeric keyboard. By default, the "accept" validation action is configured to accept both "`Enter`" and "`Return`" keys.

Accelerator key modifiers

All of the key names listed in the previous table can be combined with modifiers representing the Ctrl, Shift and Alt keys.

The names to be used for the key modifiers are "`Control-`", "`Shift-`", and "`Alt-`", to be added as prefix in accelerator name.

For example:

```
Control-P
Shift-Alt-F12
Control-Shift-Alt-Z
```

Table views

Describes how to program dialogs controlling record lists.

- Understanding tables views on page 1345
- Defining tables in the layout on page 1346
- Binding tables to arrays in dialogs on page 1349
- Controlling the total number of rows on page 1350
- Handling the current row on page 1352
- Controlling table rendering on page 1350
- Displaying column images on page 1354
- Defining actions on list columns with images on page 1355
- Built-in table features on page 1355
- Summary lines in tables on page 1360
- Defining the action for a row choice on page 1360
- Actions bound to the current row on page 1361
- Using tables on mobile devices on page 1362
- Populating a DISPLAY ARRAY on page 1372
- INPUT ARRAY row modifications on page 1377
- INPUT ARRAY temporary rows on page 1378
- DISPLAY ARRAY modification triggers on page 1380
- Cell color attributes on page 1380
- Multiple row selection on page 1381
- Examples on page 1383
  - Example 1: Simple list view on page 1383

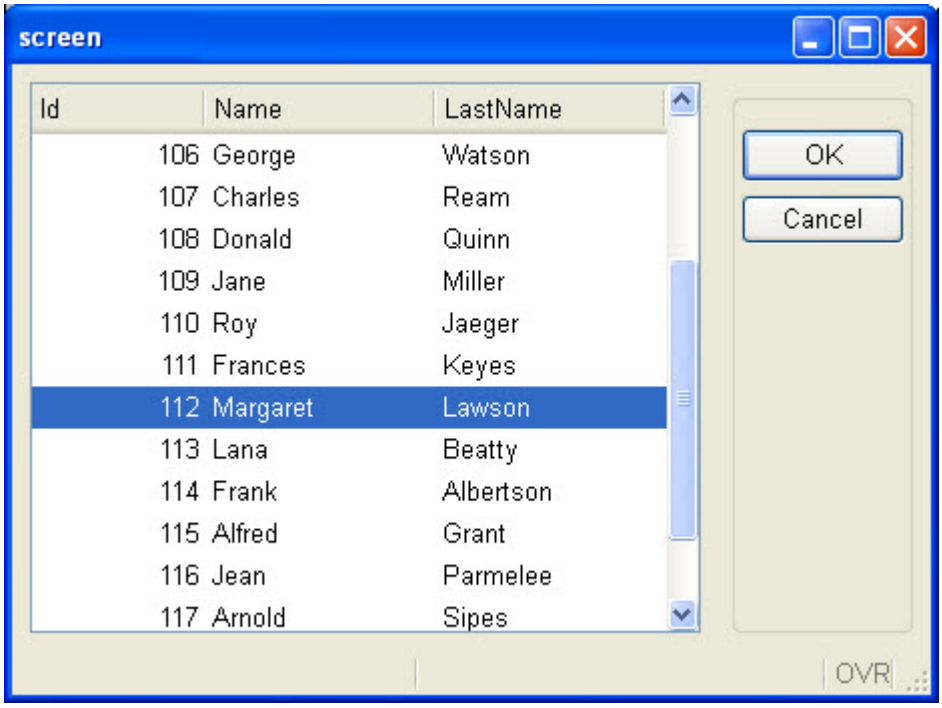
Understanding tables views

Table views define the graphical element to display a list of records.

The end user can navigate in the list to select a row or edit rows, according to the dialog controlling the table.

If the front-end platform standards allow it, the user can resize the table, sort rows, move/resize/hide columns, make multiple-row selections, search rows by criterion, and more.

Figure 82: Form with Table View (desktop front-end)



Tables views are controlled by a `DISPLAY ARRAY` or `INPUT ARRAY` instruction using a form screen-array bound to a `TABLE` container.

You can customize the rendering and the behavior of table views with form attributes in the TABLE container, and in the program using the dialog implementation.

Defining tables in the layout

Define table views in the `LAYOUT` section of the form definition file.

Desiging table views

When using a grid-based layout, the table rows and columns are defined within an area delimited by curly braces. Columns are defined with item tags and form fields. Every column tag must be properly aligned. You typically use a pipe character to separate the column tags.

A table definition using the `TABLE` layout item:

```
TABLE
{
[c1     |c2              |c3              ]
[c1     |c2              |c3              ]
[c1     |c2              |c3              ]
}
END
```

Alternatively, you can define a `<TABLE >` layout tags inside a `GRID` container, beside other layout tags:

```
GRID
{
<GROUP g1                                 >
```

```
[f1            ]
[f2                                    ]
[                                      ]
<                                      >
<TABLE t1                              >
[c1     |c2             |c3            ]
[c1     |c2             |c3            ]
[c1     |c2             |c3            ]
<                                      >
}
END
```

> **Important:** Avoid Tab characters (ASCII 9) inside the curly-brace delimited area. If used, Tab characters are replaced with 8 blanks at compilation with `fglform`.

The position of the item tags is detected by the form compiler to build the table. Column item types (widget to be used) and behavior are defined with form items in the `ATTRIBUTES` section:

```
ATTRIBUTES
EDIT c1 = customer.cust_id;
EDIT c2 = customer.cust_name;
EDIT c3 = customer.cust_address;
END
```

When using a stack-based layout, table views are defined with the `TABLE` stack item inside a `STACK` container. In this case, position/size and behavior are defined at a single place:

```
LAYOUT
  STACK
    TABLE t1(UNMOVABLECOLUMNS)
      EDIT customer.cust_id;
      EDIT customer.cust_name;
      EDIT customer.cust_address;
    END
  END
END
```

Controlling the size of the table

In a grid-based container, the default width and height of a table are defined by the columns and the number of lines used in the table layout respectively. In a stack-based container,

You can overwrite the default table by specifying the `WIDTH` and `HEIGHT` attributes.

```
TABLE t1 ( WIDTH = 5 COLUMNS, HEIGHT = 10 LINES )
```

Defining column titles

The `TABLE` layout item definition can contain column titles as well as the tag identifiers for each column's form fields. The `fglform` form compiler can associate column titles in the table layout with the form field columns if they are aligned properly.

> **Note:** At least two spaces are required between column titles.

```
TABLE
{
 Title1  Title2           Title3
[c1     |c2             |c3            ]
[c1     |c2             |c3            ]
[c1     |c2             |c3            ]
```

```
}
END
```

Alternatively, you can set the column titles of a table container by using the `TITLE` attribute in the definition of the form fields. This allows you to use localized strings for the column titles.

```
TABLE
{
[c1  |c2          |c3         ]
[c1  |c2          |c3         ]
[c1  |c2          |c3         ]
}
END
...
ATTRIBUTES
EDIT c1 = customer.cust_id, TITLE=%"label.cust_id";
EDIT c2 = customer.cust_name, TITLE=%"label.cust_name";
EDIT c3 = customer.cust_address, TITLE=%"label.cust_address";
END
```

Similarly, in a stack item `TABLE` container, columns can get a `TITLE` attribute:

```
LAYOUT
  STACK
    TABLE t1(UNMOVABLECOLUMNS)
      EDIT customer.cust_id, TITLE=%"label.cust_id";
      EDIT customer.cust_name, TITLE=%"label.cust_name";
      EDIT customer.cust_address, TITLE=%"label.cust_address";
    END
  END
END
```

Height of table rows

The height of table rows can be defined with a grid-based layout by adding empty tags underneath column tags (this makes sense only when using widgets that can get a height such as `TEXTEDIT` or `IMAGE`).

```
LAYOUT
TABLE
{
[c1  |c2                    ]
[    |                      ]
[    |                      ]
}
END
END
ATTRIBUTES
EDIT c1=FORMONLY.key;
TEXTEDIT c2=FORMONLY.thetext;
END
...
```

In the above example, the second column is defined as a `TEXTEDIT` item type, that can get a height as a number of grid cells. The height is defined by the number of item tags of the table row in the layout section (height=3 in our example)

Binding tables to arrays in dialogs

Identifying list views in program dialogs

In list dialogs such as the `INPUT ARRAY` or `DISPLAY ARRAY`, the screen array identifies the record list element in the current form to be bound to the program array used by the dialog.

In the next example, the `INPUT ARRAY` uses the `custlist` screen array of the form, and binds the `custarr` program ARRAY with:

```
INPUT ARRAY custarr FROM custlist.*
```

The screen array members will be associated to the program array record members by position. The order and number of the screen array elements matters, because these are bound by position to the members the program array. The position of the `TABLE` columns, however, can differ from the members of the screen array and program array.

To omit columns in the `TABLE` layout, yet include them in the definition of the screen array, and define the columns as `PHANTOM` fields in the form definition file.

The program array can be defined from the database table definition with the `DEFINE LIKE` instruction:

```
DEFINE custarr DYNAMIC ARRAY OF RECORD LIKE customer.*
```

Note that the array is usually defined with a flat list of members with `ARRAY OF RECORD / END RECORD`. However, the array can be structured with sub-records and still be used with a list dialog. This is especially useful when you need to define arrays from database tables, and additional information needs to be managed at runtime (for example to hold image resource for each row, to be displayed with the `IMAGECOLUMN` attribute):

```
SCHEMA shop
DEFINE a_items DYNAMIC ARRAY OF RECORD
                   item_data RECORD LIKE items.*,
                   it_image STRING,
                   it_count INTEGER
               END RECORD
...
DISPLAY ARRAY a_items TO sr.*
   ...
```

Defining screen arrays in grid-based layout TABLEs

When using a grid-based layout, the `TABLE` container is bound to a screen array defined in the `INSTRUCTION` section, by the name of the form fields used in the screen array definition.

The column data type and additional column properties are defined in the `ATTRIBUTES` section as form fields:

```
LAYOUT
...
TABLE
{
[c1    |c2     |c3             ]
[c1    |c2     |c3             ]
[c1    |c2     |c3             ]
[c1    |c2     |c3             ]
}
END
...

ATTRIBUTES
```

```
EDIT c1 = customer.cust_num;
EDIT c2 = customer.cust_name,
EDIT c3 = customer.cust_cdate;
...
```

Each form field of the table must be grouped in the `INSTRUCTIONS` section in a `SCREEN RECORD` definition.

```
SCREEN RECORD custlist( cust_num, cust_name, cust_cdate );
```

Defining screen arrays in stack-based layout TABLEs

When using a stack-based layout, the `TABLE` stack item gets a identifier, which defines the screen array to be used in programs:

```
LAYOUT
  STACK
    TABLE custlist (STYLE="regular")
       EDIT customer.cust_num;
       EDIT customer.cust_name,
       EDIT customer.cust_cdate;
    END
  END
END
```

This identified is mandatory for `TABLE` stack items.

Controlling table rendering

Table rendering can be controlled by the use of presentation styles and table attributes.

Current row rendering

By default, the current row in a `TABLE` is highlighted in display mode (`DISPLAY ARRAY`) but not in input mode (`INPUT ARRAY`, `CONSTRUCT`). You can set decoration attributes of a table with a presentation style of the `Table` class.

Table resize control

By default, tables can be resized in height. Use the `WANTFIXEDPAGESIZE` form file attribute to deny table resizing.

Current row visibility after dialog execution

When the dialog controlling the table has finished, the current row may be deselected, depending on the `KEEP CURRENT ROW` dialog attribute.

Controlling the total number of rows

Methods are provided to set and get the number of rows in a read-only or editable list of records.

> **Note:** The `DISPLAY ARRAY` and `INPUT ARRAY` dialogs can use dynamic or static arrays. Static arrays are supported for backward compatibility, consider using dynamic arrays for new development.

Set the number of rows when using a static array

When using a static array in DISPLAY ARRAY or INPUT ARRAY, you must specify the actual number of rows with the SET_COUNT() built-in function or with the COUNT dialog attribute. Both of them are only taken into account when the interactive instruction starts.

```
DEFINE arr ARRAY[100] OF ...
... (fill the array with x rows)
CALL set_count(x)
DISPLAY ARRAY arr TO sa.*
   ...
END DISPLAY
```

When using multiple list subdialogs in a DIALOG block, the SET_COUNT() built-in function is unusable, as it defines the total number of rows for all lists. The only way to define the number of rows when using a static array in multiple dialogs is to use the COUNT attribute.

Consider using dynamic arrays instead of static arrays.

Set the number of rows when using a dynamic array

When using a dynamic array in DISPLAY ARRAY or INPUT ARRAY, the total number of rows is automatically defined by the array variable (array.getLength()).

```
DEFINE arr DYNAMIC ARRAY OF ...
... (fill the array with x rows)
DISPLAY ARRAY arr TO sa.*
   ...
END DISPLAY
```

However, special consideration has to be taken when using the paged mode of DISPLAY ARRAY. In this mode, the dynamic array only holds a page of the complete row set shown to the user: In paged mode, you must specify the total number of rows with the ui.Dialog.setArrayLength() method.

Get the number of rows in a list

To get the current number of rows in a DISPLAY ARRAY or INPUT ARRAY, use either the ui.Dialog.getArrayLength() or the ARR_COUNT() function.

The getArrayLength() method can be used inside or outside the context of the list dialog, as it takes the screen array as parameter to identify the list dialog. For example, when implementing a DIALOG block with two DISPLAY ARRAY subdialogs, you can query the number of rows of a list in the code block of another list controller:

```
DIALOG ...
   DISPLAY ARRAY arr1 TO sa1.*
      ON ACTION check
         IF DIALOG.getArrayLength("sa2")] > 1 THEN
            ...
         END IF
   END DISPLAY
   DISPLAY ARRAY arr2 TO sa2.*
   END DISPLAY
END DIALOG
```

The ARR_COUNT() function must be used in the context of the DISPLAY ARRAY or INPUT ARRAY dialog, or just after executing such dialog. For example, it can be used just after an INPUT ARRAY dialog, to get the number of rows left in the list:

```
INPUT ARRAY arr FROM sa.*
   ...
```

```
END INPUT
IF NOT int_flag THEN
   FOR i=1 TO arr_count()
     ...
   END FOR
END IF
```

The ARR_COUNT( ) function returns the number of rows for the last executed dialog, until a new list dialog is started.

Handling the current row
Query and control the current row in a read-only or editable list of records.

Get the current row

To query the current row of a list, use either the ui.Dialog.getCurrentRow( ) method or the ARR_CURR( ) built-in function, according to the context.

The getCurrentRow( ) method can be used inside or outside the context of the DISPLAY ARRAY or INPUT ARRAY dialog. The method takes the name of the screen array as the argument to identify the list. For example, when implementing a DIALOG block with two DISPLAY ARRAY subdialogs, you can query the current row of a list in the code block of the other list controller:

```
DIALOG ...
   DISPLAY ARRAY arr1 TO sa1.*
      ON ACTION check
         IF arr2[DIALOG.getCurrentRow("sa2")].value > 0 THEN
            ...
         END IF
   END DISPLAY
   DISPLAY ARRAY arr2 TO sa2.*
   END DISPLAY
END DIALOG
```

The ARR_CURR( ) function must be used in the context of the current DISPLAY ARRAY or INPUT ARRAY dialog, or just after executing such a dialog. For example, when implementing modification triggers in a DISPLAY ARRAY dialog, the current row and the current screen line can be queried respectively with the ARR_CURR( ) and SCR_LINE( ) functions:

```
DISPLAY ARRAY arr TO sa.*
    ON UPDATE
       INPUT arr[arr_curr()].* WITHOUT DEFAULTS FROM sa[scr_line()].* ;
END DISPLAY
```

The ARR_CURR( ) function returns the current row index for the last executed dialog, until a new list dialog is started.

Set the current row

To set the current row in a list controlled by a DISPLAY ARRAY or INPUT ARRAY, use the ui.Dialog.setCurrentRow( ) method. This method takes the name of the screen array and the new row index as parameters:

```
DISPLAY ARRAY p_items TO sa.*
   ...
   ON ACTION next_empty
      LET row = findEmptyRow(p_items)
      CALL DIALOG.setCurrentRow("sa", row)
   ...
END DISPLAY
```

Calling the `DIALOG.setCurrentRow()` method will not execute control blocks such as `BEFORE ROW` and `AFTER ROW`, and will not set the focus. If you want to set the focus to the list, you must use the `NEXT FIELD` instruction. This works with `DISPLAY ARRAY` as well as with `INPUT ARRAY`.

> **Tip:** Use this method with care. Let the dialog handle normal navigation automatically, and jump to a specific row only in the context of an `ON ACTION` block.

The `FGL_SET_ARR_CURR()` function can also be used. This function must be called in the context of the current list having the focus.

> **Note:** `FGL_SET_ARR_CURR()` triggers control blocks such as `BEFORE ROW`, while `DIALOG.setCurrentRow()` does not trigger any control blocks.

Converting visual index to/from program array index

When the end user sorts rows in a table, the program array index (`arr_curr()`) may differ from the visual row index (the row position as seen by the user).

The `ui.Dialog` class provides methods to convert between these contexts:

The ui.Dialog.arrayToVisualIndex on page 1796 method converts a program array index to a visual index. It can be used, for example, to display a typical list position message (Row: *current-row* / *total-rows*). The current row (*arr_curr()*/*getCurrentRow()*) is a program array index that must be converted to a visual index. Note that you need to display such messages in the `BEFORE ROW` trigger and `ON SORT` trigger:

```
FUNCTION disp_row(d,n)
   DEFINE d ui.DIALOG, n STRING
   MESSAGE SFMT("Row: %1/%2",
                d.arrayToVisualIndex(n,d.getCurrentRow(n)),
                d.getArrayLength(n))
END FUNCTION
...
   DISPLAY ARRAY arr TO sr.*
       ...
       BEFORE ROW
           CALL disp_row(DIALOG,"sr")
       ON SORT
           CALL disp_row(DIALOG,"sr")
       ...
   END DISPLAY
```

The ui.Dialog.visualToArrayIndex on page 1815 method converts a visual index to a program array index. It can be used for example to ask the user for a row position (visual index), and make that row current by using `DIALOG.setCurrentRow()` after converting to the program array index:

```
DEFINE i INTEGER
...
DISPLAY ARRAY arr TO sr.*
   ...
   ON ACTION move_to
      PROMPT "Enter row index:" FOR i
      CALL DIALOG.setCurrentRow( "sr", DIALOG.visualToArrayIndex("sr", i))
   ...
END DISPLAY
```

Displaying column images

You can use PHANTOM fields and the IMAGECOLUMN attribute to display images in a column, to the left of the column value.

To display an image on the left of the column value in table views, define a PHANTOM field to hold the image name, and bind it to a parent column with the IMAGECOLUMN attribute.

```
LAYOUT
TABLE
{
[c1            |c2          ]
[c1            |c2          ]
[c1            |c2          ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.file_icon;
EDIT c1 = FORMONLY.file_name, IMAGECOLUMN=file_icon;
EDIT c2 = FORMONLY.file_size;
...
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

The program code can then display the specified image with each row.

```
DEFINE arr DYNAMIC ARRAY OF RECORD
     file_icon STRING,
     file_name STRING,
     file_size INTEGER
  END RECORD
...
FOR x=1 TO max_files
    CASE file_type(arr[x].file_name)
    WHEN "file" LET arr[x].file_icon = "file"
    WHEN "dir"  LET arr[x].file_icon = "folder"
    END CASE
END FOR
...
DISPLAY ARRAY arr TO sr.*
   ...
END DISPLAY
```

When images come from the database, these are typically fetched into BYTE variables. If the BYTE variable is located in a file (LOCATE IN FILE), it can be bound to the IMAGECOLUMN field: The runtime system will automatically display the image data. Note however that each BYTE element of the array must be located in a distinct file. This can be done as follows:

```
DEFINE arr DYNAMIC ARRAY OF RECORD
           pic_num INTEGER,
           pic_data BYTE,
           pic_when DATETIME YEAR TO SECOND
       END RECORD
...
DECLARE c1 CURSOR FOR SELECT * FROM mypics
LET i=1
LOCATE arr[i].pic_data IN FILE
FOREACH c1 INTO arr[i].*
    LOCATE arr[i:=i+1].pic_data IN FILE
END FOREACH
```

```
CALL arr.deleteElement(i)
...
```

Defining actions on list columns with images
Columns in tables displaying images can trigger action events, when the user selects the image.

TABLE and TREE containers can define columns as IMAGE field, to display pictures or icons. By default, these table cells are not clickable. When you define an ACTION attribute for a table column defined as IMAGE, the action event will fire when the image is selected (with a mouse click, for example). Note that this note apply to the IMAGECOLUMN concept, which is rather a column decoration.

> **Important:** When selecting an image, the current row may change as when selecting a new row in the table.

The following example defines a TABLE with two IMAGE columns, and attaches the update and delete actions:

```
LAYOUT
TABLE
{
[c1   |c2                |i1|i2]
[c1   |c2                |i1|i2]
[c1   |c2                |i1|i2]
}
END
END
ATTRIBUTES
EDIT c1 = FORMONLY.id, TITLE="Id", NOENTRY;
EDIT c2 = FORMONLY.name, TITLE="Name";
IMAGE i1 = FORMONLY.i_modify, ACTION=update;
IMAGE i2 = FORMONLY.i_delete, ACTION=delete;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

In the program code, use a dialog instruction to implement the action handlers for the image actions. For example, you can define a DISPLAY ARRAY with ON UPDATE and ON DELETE list modification triggers that will respectively create the update and delete actions:

```
DISPLAY ARRAY arr TO sr.*
    ON UPDATE
       -- user code
    ON DELETE
       -- use code
END DISPLAY
```

Built-in table features
Several implicit list handling features are provided by table views.
Columns layout
By default, a user can position, hide, show, and resize columns in TABLE and TREE containers.

> **Important:** This feature is not supported on mobile platforms.

Resizing columns

By default, columns can be resized. On desktop front-ends, the user can drag the right edge of a column header to increase or decrease the width of the column.

To deny column resizing for all columns in a table, add the UNSIZABLECOLUMNS attribute to the TABLE or TREE container.

To deny column resizing for an individual column, add the UNSIZABLE attribute to the form field definition for that column.

Hiding/showing columns

By default, the user can control the visibility of columns. On desktop front-ends, a user right-clicks on a column header to get a contextual menu that allows to show/hide columns.

To deny the column visibility option for all columns in a table, add the UNHIDABLECOLUMNS attribute to the TABLE or TREE container.

To deny the column visibility option for an individual column, add the UNHIDABLE attribute to the form field definition for that column.

To hide a column initially but allow column visibility, set the HIDDEN attribute with the value USER in the form field definition for that column. This hides the column by default, and lets the user show the column if needed.

Changing column positions

By default, columns can be moved around. On desktop front-ends, a user can rearrange columns by dragging the column header to a different position.

To deny this option, add the UNMOVABLECOLUMNS attribute to the TABLE or TREE container.

To deny this option for an individual column, add the UNMOVABLE attribute to the form field definition for that column.

List ordering

List controllers implement a built-in sort. This feature can be disabled if not required.

When a DISPLAY ARRAY or INPUT ARRAY block is combined with a TABLE container, the row sorting feature is implicitly available. Row sorting is supported on TREE containers with DISPLAY ARRAY dialogs only.

> **Important:** This feature is not supported on mobile platforms.

To sort rows in a list, the user must click on a column header of the table. Clicking on a table column header triggers a GUI event that instructs the runtime system to reorder the rows displayed in the list container.

In fact, the rows are only sorted from a visual point of view; the data rows in the program array (the model) are left untouched. Therefore, when rows are sorted, the visual position of the current row might be different from the current row index in the program array.

To sort rows, the runtime system uses the standard collation order of the system, following the current locale settings. As result, the rows might be ordered a bit differently than when using the database server to sort rows (with an ORDER BY clause of the SELECT statement), since database servers can define their own collation sequences to sort character data.

The built-in sort is enabled by default. To prevent sorting in a TABLE or TREE containers, defined the UNSORTABLECOLUNMS attribute at the list container level, or set the UNSORTABLE attribute at the column/field level. As rows can be created and modified during an INPUT ARRAY instruction, you may want to use the UNSORTABLECOLUMNS attribute for tables controlled by INPUT ARRAY.

To execute code after a sort was performed, use the ON SORT interaction block in the dialog, for example to display the current row position with ui.Dialog.arrayToVisualIndex on page 1796.

The sorting feature is disabled when using the paged mode of DISPLAY ARRAY, because not all result set rows are known by the runtime system in this mode. However, it is possible to detect a sort request from the user with the ON SORT trigger. You can then re-execute the SQL query with a new sort order. For more details, see Populating a DISPLAY ARRAY on page 1372.

When an application window is closed, the selected sort column and order is stored by the front-end in the user settings database of the system (for example, on Windows™ platforms it's the registry database). The sort will be automatically re-applied the next time the window is created. This way, the rows will appear sorted when the program restarts. The saved sort column and order is specific to each list container.

Find function
List controllers implement a built-in find. This feature can be disabled if not required.

The `DISPLAY ARRAY` and `INPUT ARRAY` block blocks support the built-in find feature by default.

> **Important:** This feature is not supported on mobile platforms.

This feature works with any list container (`TABLE`, `TREE`, `SCROLLGRID`).

The built-in find creates the implicit "find" and "findnext" actions. These actions can be decorated, enabled and disabled as regular actions.

When the user triggers the "find" action (default accelerator is Ctrl-F), the dialog opens a popup window to let the user enter a search value. On validation with the OK button, the dialog starts to search a row where a field value matches the value entered in the find dialog. The "find" action starts the search from the current row. After a first search, the user can trigger the "findnext" action (default accelerator is Ctrl-G), in order to continue the search in the rest of the record list, without opening the find dialog again (the current search value will be reused).

By default, any table column is scanned, but the user can select a specific column in the find dialog box, as long as a column title is available. Case-sensitive or insensitive search as well as wraparound options are also available.

The value entered in the find dialog is compared to all fields of visible columns, except columns of the type `TEXT` or `BYTE`. The comparison is based on the formatted value. For example, a `MONEY` column will display values formatted with the currency symbol. To match values in that column, the user must enter exactly the same value (i.e. with the currency symbol and the correct decimal separator). When using `COMBOBOX` fields, the find searches in the visible values of combobox items.

Only text widgets displaying values are searched. Columns using widgets such as images, radio-groups, checkboxes are not searched. Further, the find function ignores `PHANTOM` fields, hidden fields and fields defined with the `INVISIBLE` attribute.

Only rows in memory can be searched. When using the paged-mode (`ON FILL BUFFER`), the built-in search is disabled. When implementing dynamic tree views , the built-in find will only search the tree nodes available in the program array.

If the dialog defines an explicit `ON ACTION find` or `ON ACTION findnext`, the default built-in find is disabled.

Keyboard seek
The keyboard seek feature allows a user to find a row in a read-only list, by typing characters.

During a `DISPLAY ARRAY`, when the user types alphabetic characters on the keyboard, the runtime system will automatically seek to the next row having a character field that contains a value starting with the typed characters. The seek search restarts from the current row when the user types a new characters on the keyboard.

> **Important:** This feature is not supported on mobile platforms.

This feature works with any list container (`TABLE`, `TREE`, `SCROLLGRID`).

Numeric, date/time and large data (`TEXT`/`BYTE`) columns are ignored. Only character columns are searched, fields using widgets like image, radio-group or checkbox are ignored. Further, the seek function ignores `PHANTOM` fields, hidden fields and fields defined with the `INVISIBLE` attribute.

The user can rapidly type several characters on the keyboard, to search for a value that starts with the typed characters. After a given timeout (less than a second), the seek buffer is cleared and a new search filter can be taken into account.

The seek search is case-insensitive.

If no row could be found from the typed characters, the [Not found] error -8105 will be displayed automatically.

If an alphabetic character is used as action accelerator, the built-in seek feature is disabled, because the accelerator must fire the corresponding action.

Only rows in memory can be searched. When using page-mode (`ON FILL BUFFER`), the built-in seek is disabled. When implementing dynamic tree views, the built-in seek will only search the tree nodes available in the program array.

By default, any character column of the list is scanned. But if the list gets sorted, the runtime system considers that the sort column is the most important and searches only in that column.

Reduce filter
The reduce filter allows a user to reduce the row set in a read-only list according to a filter.

When using a `DISPLAY ARRAY` with a `TABLE` container, and if the front-end supports filter search facility, the user can enter a criterion in that search field, to show only the rows matching the content of the filter.

> **Important:** This feature is only for mobile platforms.

The filter search is case-insensitive.

The value entered in the filter field is compared to all fields of visible columns, except columns of the type `TEXT` or `BYTE`. The comparison is based on the formatted value. For example, a `MONEY` column will display values formatted with the currency symbol. To match values in that column, the user must enter exactly the same value (i.e. with the currency symbol and the correct decimal separator). When using `COMBOBOX` fields, the find searches in the visible values of combobox items.

Only text widgets displaying values are searched. Columns using widgets such as images, radio-groups or checkboxes are not searched. The filter function ignores `PHANTOM` fields, hidden fields and fields defined with the `INVISIBLE` attribute.

Only rows in memory can be searched. When using page-mode (`ON FILL BUFFER`), the built-in filter is disabled. When implementing dynamic tree views, the built-in filter will only search the tree nodes available in the program array.

If the rows are filtered (i.e. some value is present in the search field), any non-rowbound action is disabled. On iOS, the action bar is replaced by the search bar.

The list filter is typically used on mobile devices for full-screen list views.

Figure 83: iOS list view with filter field

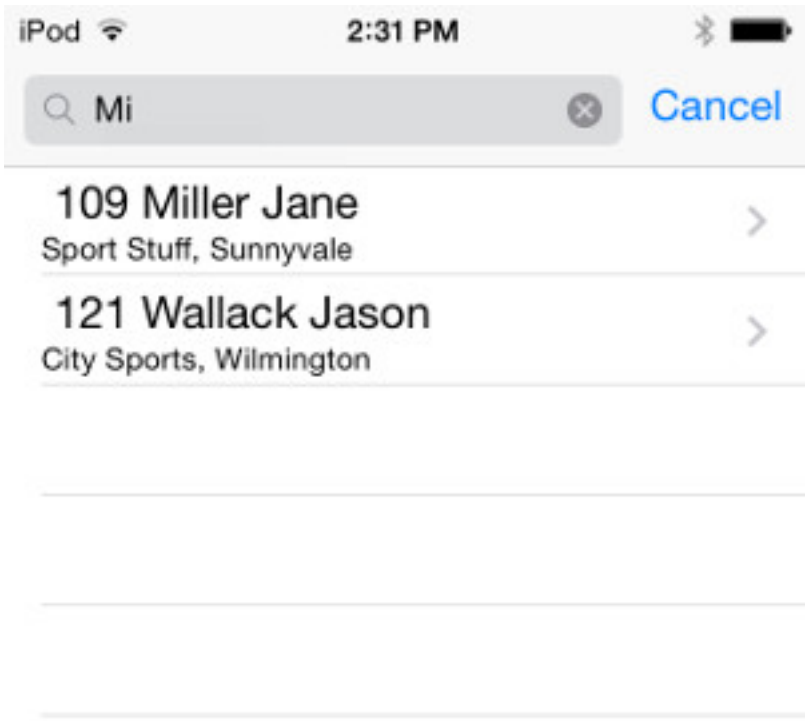
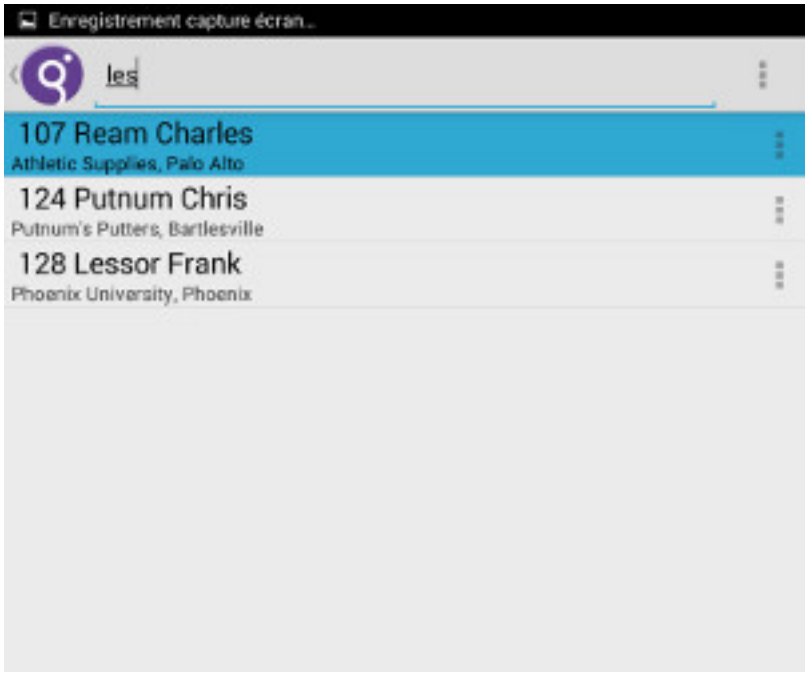


Figure 84: Android™ list view with filter field



Summary lines in tables

Table views can display a summary line, to show aggregate values for columns.

To get a summary line in a table, define AGGREGATE fields at the bottom of the TABLE container, with the corresponding form item definition in the INSTRUCTION section.

> **Important:** This feature is not supported on mobile platforms.

Define the type of the aggregate field with the AGGREGATETYPE attribute: The aggregate value can be automatically computed, or set by program.

To get a global label for the summary line, specify the AGGREGATETEXT attribute at the TABLE level. This aggregate label will appear on the left in the summary line, if no aggregate text is defined at the aggregate field level.

To decorate the summary line, use presentation style attributes such as summaryLineAlwaysAtBottom.

The next example defines a "total" aggregate field for the third column of the table:

```
TABLE (AGGREGATETEXT="Total")
{
 [c1  |c2          |c3         ]
 [c1  |c2          |c3         ]
                   [total      ]
}
END
...
ATTRIBUTES
...
AGGREGATE total = FORMONLY.total, AGGREGATETYPE=PROGRAM;
...
```

For details, see AGGREGATE item definition on page 933.

Defining the action for a row choice

The row choice in the DISPLAY ARRAY dialog can be associated with a dedicated action.

When using a DISPLAY ARRAY dialog to control a table view with a graphical front-end, by default, a double-click on a row (for a desktop client), or a tap on a row (for mobile clients) has the following behavior:

- On a desktop front-end, by default, a mouse double-click changes the current row, and fires the "accept" action if available. If the default accept action is fired, the dialog will end, except if the accept action has been disabled or was over-written by a ON ACTION accept handler. This default behavior fits most of the record list of a desktop application, where the main purpose is to let the user choose a row from the list.
- On a mobile devices, by default, a tap on a row changes the current row only. This corresponds to a single mouse click on a desktop front-end, and therefore does not fire the "accept" action by default. If a tap must fire the accept action, define the DOUBLECLICK attribute.

In order to detect the physical event when the user chooses a row with a double-click on desktop clients and tap on mobile clients, define the DOUBLECLICK attribute of DISPLAY ARRAY dialogs to fire an action handler block (ON ACTION *double-click-action-name*):

```
DISPLAY ARRAY arr TO sr.*
        ATTRIBUTES(UNBUFFERED, DOUBLECLICK=select)
   ON ACTION select
      MESSAGE "myselect:", arr_curr()
END DISPLAY
```

If the DOUBLECLICK attribute is defined, it will only configure the action for the double-click or tap physical event: By default, the accept action is still available, and the [Ok] button or the [Return] key will still fire

the accept action and leave the dialog. To avoid the default accept action, add ACCEPT=FALSE to the DISPLAY ARRAY attribute list:

```
DISPLAY ARRAY arr TO sr.*
        ATTRIBUTES(UNBUFFERED, DOUBLECLICK=select, ACCEPT=FALSE)
   ON ACTION select
      MESSAGE "myselect:", arr_curr()
END DISPLAY
```

Note that if the selected row is not the current row, any defined AFTER ROW and BEFORE ROW control blocks execute before the ON ACTION block. The code blocks execute in the following order:

1. AFTER ROW (for the previous current row)
2. BEFORE ROW (for the new current row)
3. ON ACTION *double-click-action*

When defining a DOUBLECLICK action, you declare an explicit action view, and no default action view will be displayed for this action (except if you explicitely force it with DEFAULTVIEW=YES).

The double-click action can also be defined as TABLE/TREE attribute in form files. DOUBLECLICK in DISPLAY ARRAY attributes has a higher precedence as DOUBLECLICK in the form file. For more details, see DOUBLECLICK attribute on page 962.

Actions bound to the current row

Actions can be configured with the ROWBOUND attribute to depend from the current row.

When using a DISPLAY ARRAY or INPUT ARRAY dialog to control a table view, actions can get the ROWBOUND attribute in order to make the action only available when there is a current row in the list.

> **Important:** This feature is only for mobile platforms.

The ROWBOUND attribute must only be used with TABLE and TREE containers (it does not make sense for SCROLLGRID and static lists in GRID containers).

This attribute is generally used in mobile applications, when a list view requires actions to be decorated in a row-specific way. For example, on Android™ devices, the actions with the ROWBOUND attribute will be available by selecting the three-dot button on the right of each list view cell.

In the next example, the DISPLAY ARRAY dialog implements three actions:

- The "refresh" action is not "rowbound", and will always be available (i.e. active/visible), even if the list is empty.
- The "check" action is rowbound, and will only be available if there is a (current) row in the list.
- The "delete" action created by the ON DELETE modification trigger is implicitely "rowbounded".

```
DISPLAY ARRAY a_orders TO sr.* ATTRIBUTES(UNBUFFERED)
   ...
   ON ACTION refresh -- not rowbound
      CALL fetch_orders()
   ON ACTION check ATTRIBUTES(ROWBOUND)
      CALL check_order(arr_curr())
   ON DELETE -- implicitly rowbound
      CALL delete_order(arr_curr())
   ...
END DISPLAY
```

Using tables on mobile devices
Table views render in a specific way on mobile devices, in order to take advantage of mobile device ergonomics.
Unsupported table features
Some table / list view features are not supported on mobile devices.

The list view features not supported on mobile devices include:

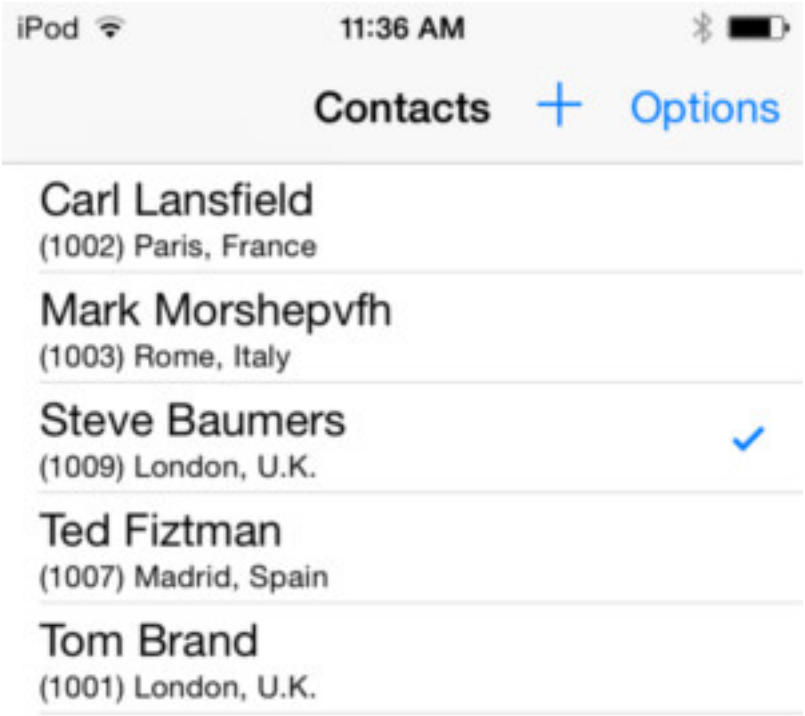
- Multiple row selection on page 1381
- Summary lines in tables on page 1360
- List ordering on page 1356
- Find function on page 1357
- Keyboard seek on page 1357
- Columns layout on page 1355
- Drag & drop on page 1411

Note also that there are no column headers/titles in mobile list views.

Two-column display
On mobile devices, a TABLE container displays as a list view with the first two columns' content.

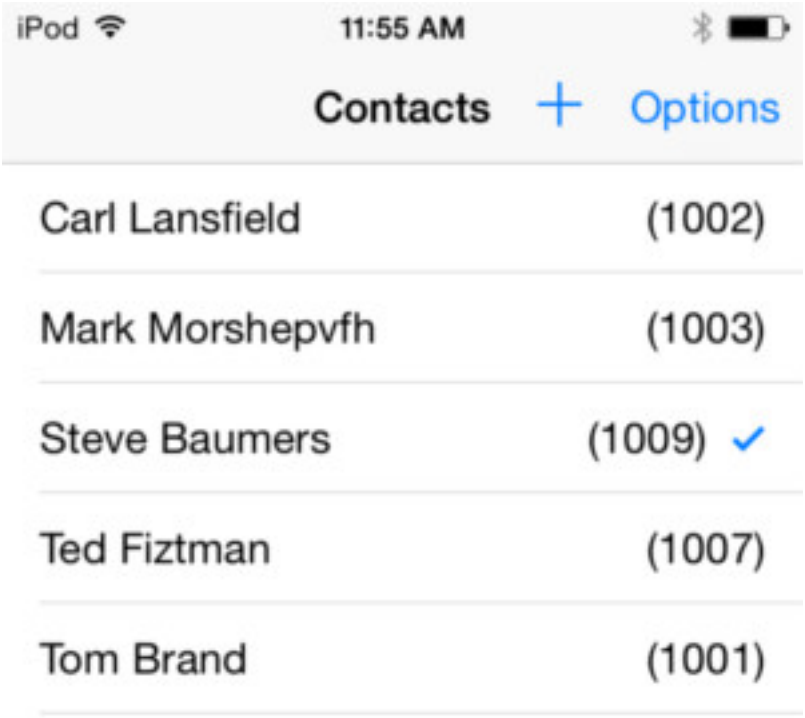
While a TABLE container and the corresponding list controller (DISPLAY ARRAY) can define multiple columns, only the first two columns are rendered on a mobile device. The first column defines the main information to be shown for the row (such as a customer name), while the second column contains additional information (such as a comment, date, address or phone number).

Figure 85: iOS list view with two-column default rendering



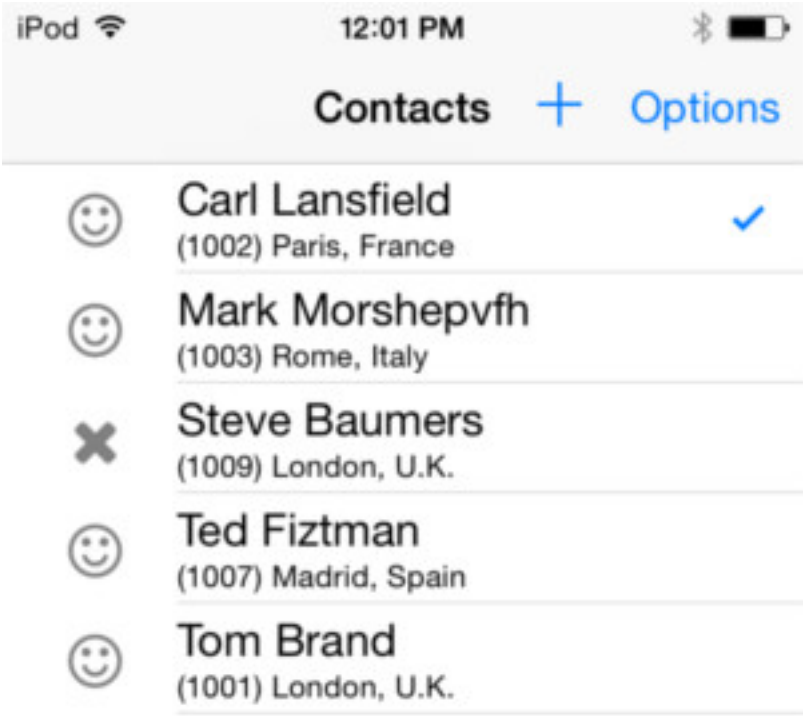
If the second column contains numeric data or has the `JUSTIFY=RIGHT` attribute, both columns display on a single line with the first column left-aligned and the second column right-aligned.

Figure 86: iOS list view with side-by-side rendering



A list view on a mobile device can include an image for each row. To display an image, associate a `PHANTOM` column to the `IMAGECOLUMN` attribute of the first column definition. For more details about images in lists, see Displaying column images on page 1354.

Figure 87: iOS list view with row images



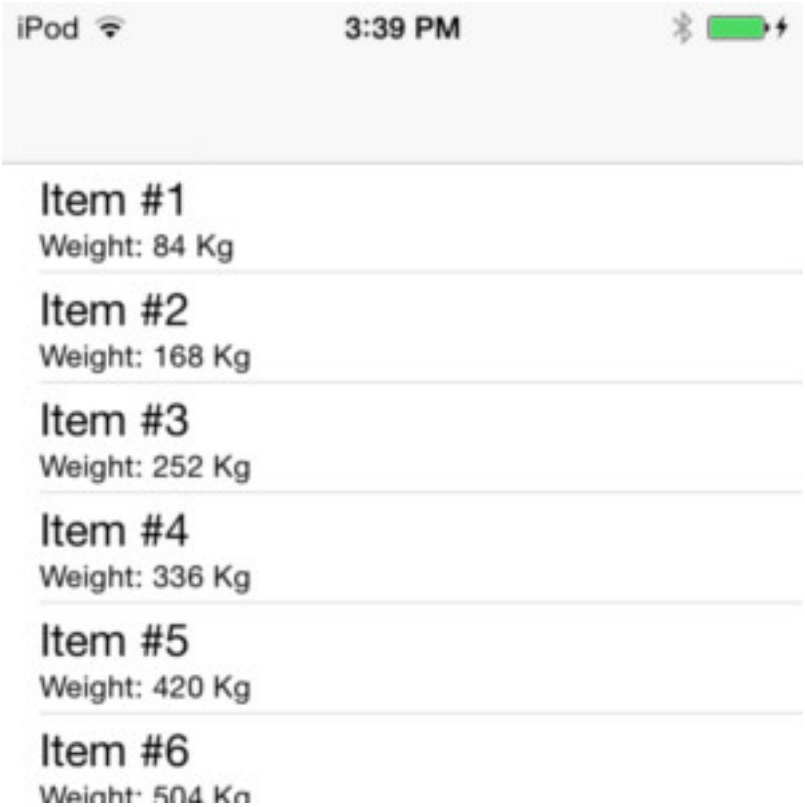
Full and Embedded list views
On mobile devices, table views are displayed as either full screen lists or embedded lists, according to the layout definition.

Full list view

A *full list view* displays when the table is the only element in a form.

```
LAYOUT
TABLE
{
[c1   |c2        ]
}
END
END
```

Figure 88: iOS full list view rendering



Embedded list view

An *embedded list view* displays when the table is mixed with other form elements. All rows of the table are shown. Scrolling is not possible.

> **Tip:** With an embedded list view, consider limiting the number of rows in the program array.

In this example, the table is inside a `GRID` container:

```
LAYOUT
GRID
{
<GROUP g1             >
 Id:   [f1           ]
 Name: [f2           ]
<                     >
<GROUP g2             >
 <TABLE t1          >
 [c1    |c2         ]
 [c1    |c2         ]
 [c1    |c2         ]
 <                  >
<                     >
}
END
END
```

Figure 89: iOS embedded list view rendering

The DOUBLECLICK (tap) action
On mobile devices, the `DOUBLECLICK` attribute defines the action to fire when a row is tapped.

By default, no action is fired on mobile devices when the user taps on a row. To fire a dedicated action, add the `DOUBLECLICK` attribute to the `DISPLAY ARRAY` dialog and define an `ON ACTION` action handler.

```
DISPLAY ARRAY arr TO sr.*
        ATTRIBUTES( DOUBLECLICK=row_select )
   ON ACTION row_select
      CALL process_row(arr_curr())
...
```

Alternatively, you can add a `DOUBLECLICK` attribute to your `TABLE` definition in your form file.

> **Tip:** We recommend you specify the `DOUBLECLICK` attribute with the `DISPLAY ARRAY` dialog, as it is strongly related to the `DISPLAY ARRAY` dialog.
>
> **Note:**
>
> - On Android™ devices, a long tap on a row only selects the row. The `DOUBLECLICK` action is not fired.
> - For iOS devices, consider using list view decoration options, as described in Row configuration on iOS devices on page 1369.

Rowbound actions

A rowbound action specifies an action to apply to the selected row. Rowbound actions get specific rendering and behavior on mobile devices.

Rowbound actions are action defined with the `ROWBOUND` action attribute in `ON ACTION` handlers. Rowbound actions can also be default actions that are implicitly related to the current row, such as the "delete" action.

```
DISPLAY ARRAY arr TO sr.*
   ...
   ON ACTION clear_list -- not rowbound
      ...
   ON ACTION copy_row ATTRIBUTES(ROWBOUND, TEXT="Copy row")
      ...
   ON ACTION check_row ATTRIBUTES(ROWBOUND, TEXT="Check row")
      ...
   ON DELETE -- implicitly rowbound
      ...
```

Genero Mobile for Android™ (GMA)

On Android 4 devices, when rowbound actions are defined, each row of a list view shows the three-dot indicator. Tap this icon to bring up a row context menu with options to execute the corresponding rowbound actions. Swipe the row from the right to the left to fire the delete action, it defined.

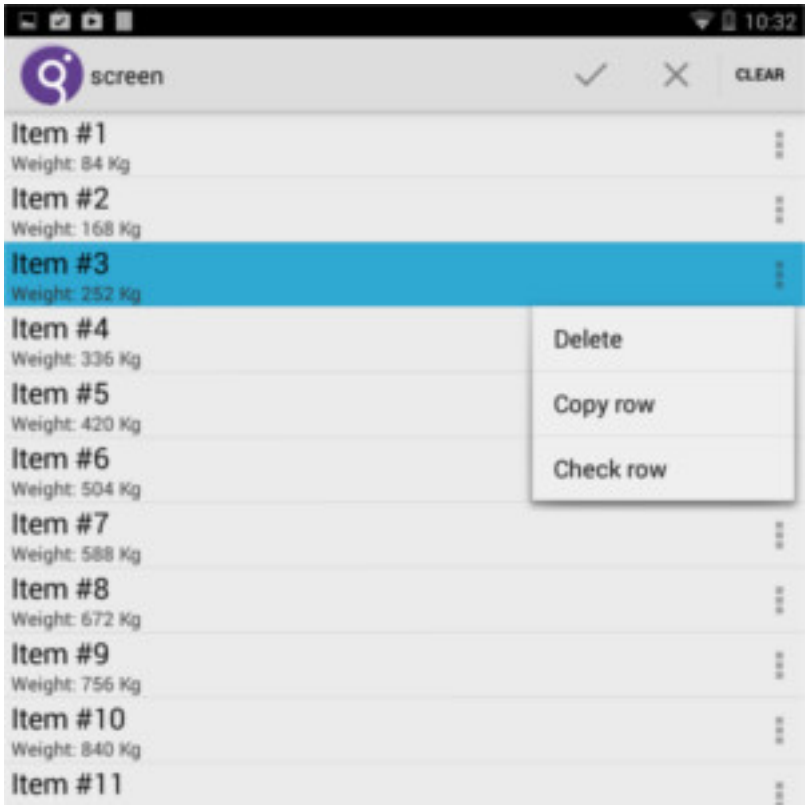


Figure 90: Android list view with rowbound actions

Genero Mobile for iOS (GMI)

On iOS 7 devices, when you swipe your finger from right to left, **More…** and/or **Delete** icons show up in the row. Tap **More...** to bring up a list of rowbound actions to execute. Tap **Delete** to fire the corresponding delete action code.

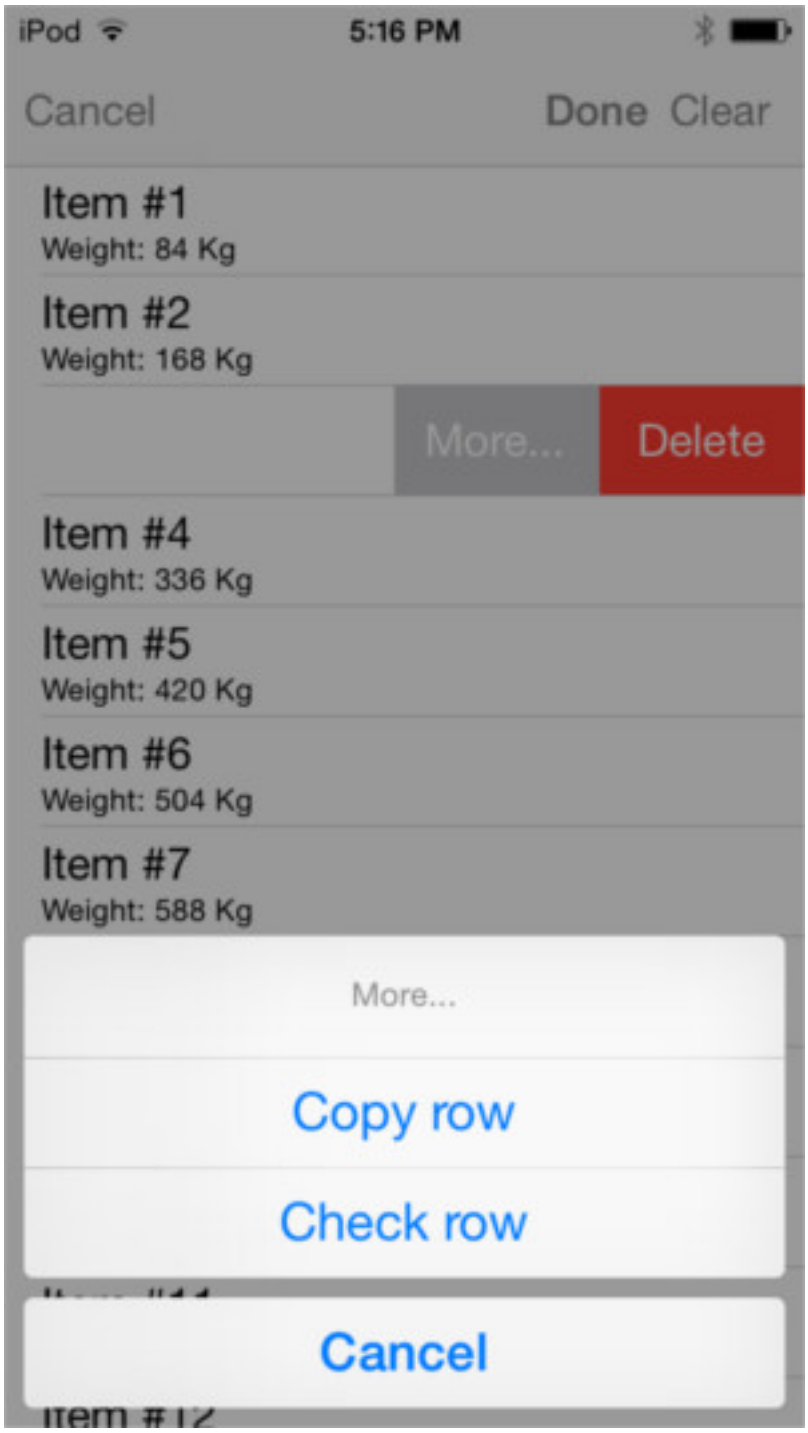


Figure 91: iOS list view with rowbound actions

Close, accept and cancel actions
The default rendering of the close, accept and cancel actions with a list view depends on the mobile device.

A `DISPLAY ARRAY` dialog implements the "close", "accept", and "cancel" actions by default. When using a full list view, these actions are default action views. The rendering of these actions vary according to the type of mobile device. The accept and cancel buttons typically show up on the top of the list view.

For more details, see Rendering default action views on mobile on page 1279.

Row configuration on iOS devices

On iOS devices, table views can be configured to use specific row decorations.

> **Note:** The features described in this topic are provided for iOS devices. The decoration attributes are ignored by Genero Mobile for Android™ (GMA)

The ACCESSORYTYPE attribute

On iOS devices, the `ACCESSORYTYPE` attribute used in the `DISPLAY ARRAY` dialog `ATTRIBUTES` clause defines the type of icon that appears at the right side of each row.

Possible values for the `ACCESSORYTYPE` attribute are:

- `CHECKMARK`
- `DETAILBUTTON`
- `DISCLOSUREINDICATOR`

For more details about the `ATTRIBUTES` syntax, see Syntax of DISPLAY ARRAY instruction on page 1076.

Checkmark

When using `ACCESSORYTYPE=CHECKMARK`, the current row gets a check mark icon on the right hand side.

This decoration is typically used to get a visual indicator for the current row, so the user knows what row will be selected when the `DISPLAY ARRAY` dialog is validated with an accept (Done) action:

```
DISPLAY ARRAY arr TO sr.*
      ATTRIBUTES( ACCESSORYTYPE=CHECKMARK )
    ...
```

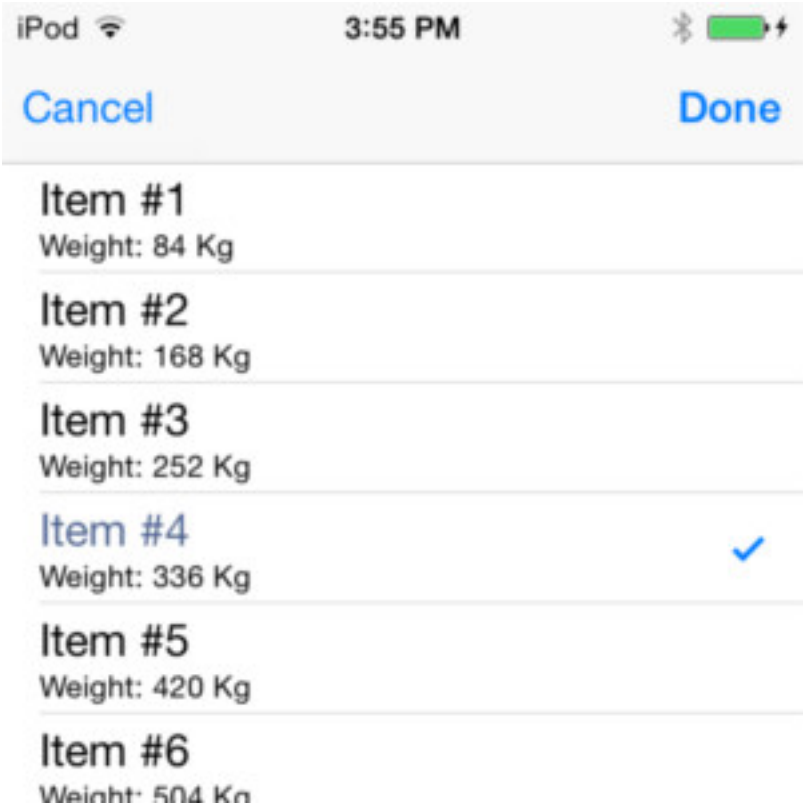


Figure 92: iOS list view with checkmark

To customize the application, define the color of the checkmark with the `iosTintColor` Window-class style attribute.

Detail button

When using `ACCESSORYTYPE=DETAILBUTTON`, each row gets a `(i)` icon on the right-hand side.

To specify what action must be fired when the user taps on the `(i)` icon, define the `DETAILACTION` in the `DISPLAY ARRAY` attributes, and its corresponding `ON ACTION` handler.

By opening a new window when in the detail action code, a tap on the icon shifts the current window from right to left, to show the new screen.

When tapping on another part of a row, by default, the row becomes then new current row. To follow typical iOS standards, you should also define a `DOUBLECLICK` with its corresponding `ON ACTION` handler, to handle current row selection with a dedicated action. If tapping on any part of a row should open a detail form, use the `DISCLOSUREINDICATOR` solution instead of `DETAILBUTTON`.

When selecting a different row, the `AFTER ROW` / `BEFORE ROW` control blocks are executed before the detail action or double-click action.

```
DISPLAY ARRAY arr TO sr.*
      ATTRIBUTES( ACCESSORYTYPE=DETAILBUTTON,
                  DETAILACTION=edit_details,
                  DOUBLECLICK=select_row )
   ...
   ON ACTION edit_details
      OPEN WINDOW w_details WITH FORM "details"
      INPUT BY NAME arr[i].*
        ...
      END INPUT
      CLOSE WINDOW w_details
   ON ACTION select_row
      ...
```

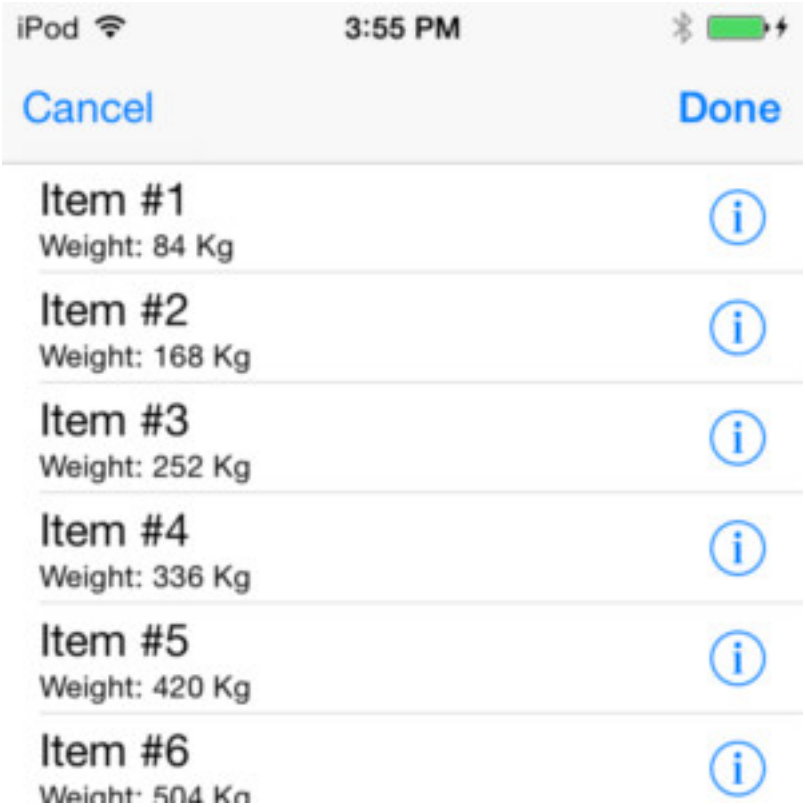


Figure 93: iOS list view with detail button

To customize the application, define the color of the disclosure indicator with the `iosTintColor` Window-class style attribute.

Disclosure indicator

When using `ACCESSORYTYPE=DISCLOSUREINDICATOR`, each row gets a > gray chevron at the right of each row. This decoration is typically used when tapping the button brings up a list of more choices related to the current row, or to open a detail form to modify the list element.

To execute code when a tapping on a row, define the `DOUBLECLICK` attribute and its corresponding `ON ACTION` handler.

By opening a new window when in the detail action code, a tap on a row shifts the current window from right to left, to show the new screen.

When selecting a different row, the `AFTER ROW` / `BEFORE ROW` control blocks are executed before the double-click action.

```
DISPLAY ARRAY arr TO sr.*
      ATTRIBUTES( ACCESSORYTYPE=DISCLOSUREINDICATOR,
                  DOUBLECLICK=row_select )
   ...
   ON ACTION row_select
      MENU "Options" ATTRIBUTES(STYLE="dialog")
         COMMAND "Refresh"
            ...
         COMMAND "Duplicate"
            ...
         COMMAND "Compress"
            ...
         COMMAND "Refresh"
            ...
```

```
        COMMAND "Synchronize"
           ...
     END MENU
...
```

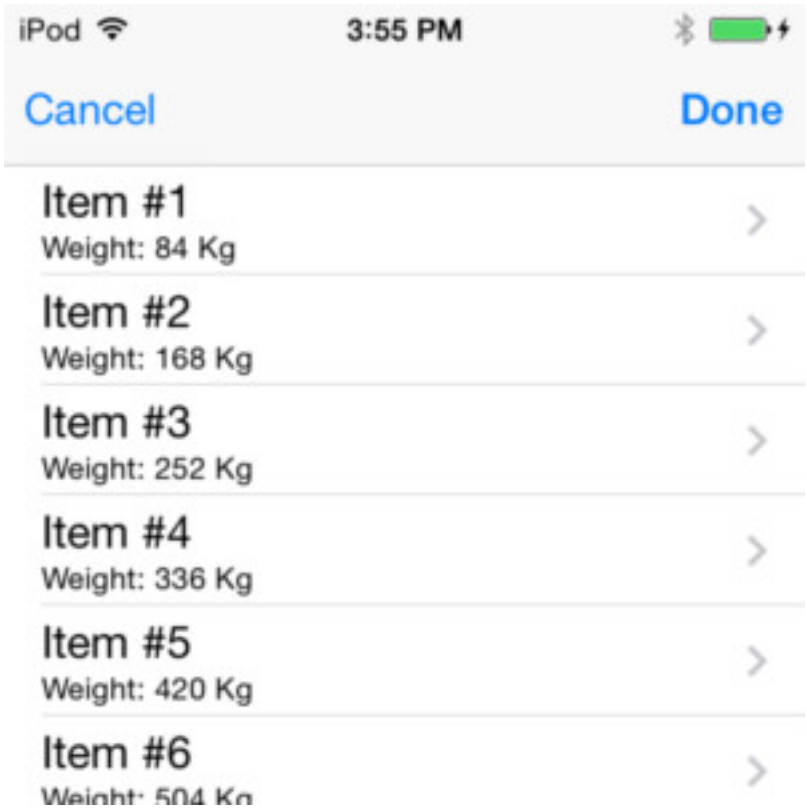


Figure 94: iOS list view with disclosure indicator

Populating a DISPLAY ARRAY
The program array must be filled with rows to populate the `DISPLAY ARRAY` dialog.

With `DISPLAY ARRAY`, either full list mode or paged mode is used to fill the form array. Consider using full list mode for short/medium result sets, and use paged mode for very large result sets.

Full list mode of DISPLAY ARRAY
In order to handle short/medium result sets, use the full list mode of `DISPLAY ARRAY`.

Understanding the full list mode

In *full list mode*, `DISPLAY ARRAY` uses a complete copy of the result set to be displayed in the form array. The full list mode is typically used for a short or medium row set (10 - 100 rows).

In full list mode, the `DISPLAY ARRAY` instruction uses a static or dynamic program array defined with a record structure corresponding to (or to a part of) a screen-array in the current form.

The program array is filled with data rows before `DISPLAY ARRAY` is executed, typically with a `FOREACH` loop when rows come from the database.

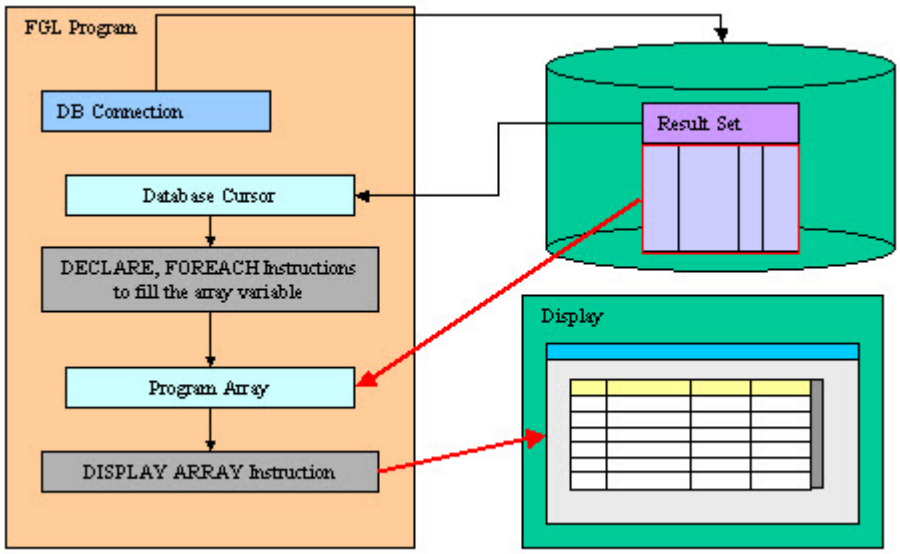


Figure 95: Full list mode in DISPLAY ARRAY diagram

Consider using a dynamic array instead of a static array: By using a dynamic array the program will only use the required memory resources, and the dialog will automatically detect the number of rows from the dynamic array (`array.getLength()`)

Full list mode example

The following example implements a `DISPLAY ARRAY` in its simpler form: A dynamic array is filled with database rows and contains the whole result set to be displayed in the table:

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
             id INTEGER,
             fname CHAR(30),
             lname CHAR(30)
         END RECORD
  DEFINE i INTEGER

  DATABASE stores

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 CURSOR FOR
          SELECT customer_num, fname, lname FROM customer
  LET i=1
  FOREACH c1 INTO arr[i].*
      LET i = i+1
  END FOREACH
  CALL arr.deleteElement(i)

  DISPLAY ARRAY arr TO sa.* ATTRIBUTES(UNBUFFERED)
      BEFORE ROW
        MESSAGE "Moved to row ", arr_curr()
  END DISPLAY

END MAIN
```

Paged mode of DISPLAY ARRAY

In order to handle very large result sets, use the paged mode of DISPLAY ARRAY.

Understanding the paged mode

The *paged mode* of DISPLAY ARRAY allows the program to display a very large number of rows, without copying all database rows into the program array.

This mode uses the ON FILL BUFFER data block to let the program populate the array with the current visible page of rows. This is a subset of the database query result set (SELECT), typically controlled by a scrollable cursor.

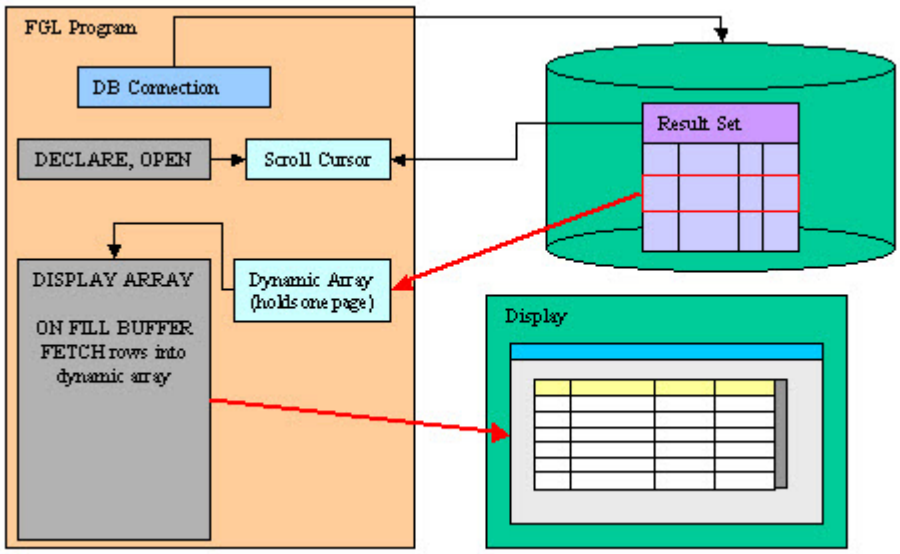


Figure 96: Paged mode diagram

> **Note:** DISPLAY ARRAY has following contraints when using the paged mode:
>
> - By default, row sorting is not allowed: Implement an ON SORT trigger to handle list sorting,
> - Multi-range selection is not supported, if the paged mode uses an undefined number of rows (COUNT=-1),
> - To fill a tree view dynamically, use the ON EXPAND / ON COLLAPSE data blocks.

Paged mode programming details

In paged mode, the dynamic array holds a page of rows, not all rows of the result set. The data rows are provided throught the ON FILL BUFFER block, by filling a dynamic array with the rows of the current page.

The ON FILL BUFFER clause is used to fill a page of rows in the dynamic array, according to a row offset and the number of rows required in the page. The row offset is defined by the FGL_DIALOG_GETBUFFERSTART() built-in function, and the number of rows to provide is defined by the FGL_DIALOG_GETBUFFERLENGTH() built-in function.

> **Note:** The ON FILL BUFFER is triggered when all the user code is executed and the dialog gets the control back, this means that the fill clause is not immediately fired when calling DIALOG.setArrayLength().

If known, specify the total number of rows with the COUNT attribute in the ATTRIBUTES clause of DISPLAY ARRAY. The total number of rows can be changed during dialog execution with the ui.Dialog.setArrayLength() method. In singular DISPLAY ARRAY instructions, you define the total number of rows of a paged mode with the SET_COUNT() built-in function or the COUNT attribute. But

these are only taken into account when the dialog starts. If the total number of rows changes during the execution of the dialog, the only way to specify the number of rows is `DIALOG.setArrayLength()`.

If the total number of rows is not known before starting the `DISPLAY ARRAY` dialog, set `COUNT=-1`. The dialog will then query for rows until the end of the result set is reached. The end of the result set is detect when the number of rows provided in `ON FILL BUFFER` are less then the number of rows asked by the dialog, or if you reset the total number of rows to a value higher value as -1 with the `ui.Dialog.setArrayLength()` method. Note that the dialog cannot support multi-row selection when the total number of rows is undefined.

It is not possible to use treeview decoration when the dialog uses the paged mode: For treeviews, the dialog needs the complete set of open nodes with parent/child relations to handle the tree view display. With the paged mode only a short window of the dataset is known by the dialog. If you use a tree view with a paged mode `DISPLAY ARRAY`, the program will raise an error at runtime.

A typical paged `DISPLAY ARRAY` implementation consists of a scroll cursor providing the list of records to be displayed. Scroll cursors use a static result set. If you want to display fresh data, you can implement an advanced paged mode by using a scroll cursor that provides the primary keys of the referenced result set, plus a prepared cursor to fetch rows on demand in the `ON FILL BUFFER` clause. In this case you may need to check whether a row still exists when fetching a record with the second cursor.

Paged mode basic example

The following example shows a `DISPLAY ARRAY` implementation using a scroll cursor to fill pages of records in `ON FILL BUFFER`, specifying an undefined number of rows (`COUNT=-1`).

```
MAIN
  DEFINE arr DYNAMIC ARRAY OF RECORD
             id INTEGER,
             fname CHAR(30),
             lname CHAR(30)
         END RECORD
  DEFINE cnt, ofs, len, row, i INTEGER

  DATABASE stores

  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1

  DECLARE c1 SCROLL CURSOR FOR
          SELECT customer_num, fname, lname FROM customer
  OPEN c1

  DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=-1)
    ON FILL BUFFER
       CALL arr.clear()
       LET ofs = fgl_dialog_getBufferStart()
       LET len = fgl_dialog_getBufferLength()
       LET row = ofs
       FOR i=1 TO len
          FETCH ABSOLUTE row c1 INTO arr[i].*
          IF SQLCA.SQLCODE!=0 THEN
            CALL DIALOG.setArrayLength("sa",row-1)
            EXIT FOR
          END IF
          LET row = row + 1
       END FOR
    ON ACTION ten_first_rows_only
       CALL DIALOG.setArrayLength("sa", 10)
  END DISPLAY

END MAIN
```

Paged mode with sorting feature

To implement row sorting in a `DISPLAY ARRAY` using paged mode, use the `ON SORT` trigger to detect a sort request, get the sort information with the ui.Dialog.getSortKey on page 1802 / ui.Dialog.getSortReverse on page 1802 methods, and re-execute the SQL query to sort rows accordingly with an `ORDER BY` clause. The `ON SORT` trigger will be fired before the `ON FILL BUFFER` trigger:

```
MAIN
  DATABASE test1
  OPEN FORM f1 FROM "custlist"
  DISPLAY FORM f1
  CALL show_list()
END MAIN

FUNCTION execute_sql(order_by)
  DEFINE order_by STRING
  DEFINE sql STRING
  IF order_by IS NULL THEN
     LET order_by = "ORDER BY fname"
  END IF
  LET sql = "SELECT customer_num, fname, lname FROM customer ", order_by
  DECLARE c1 SCROLL CURSOR FROM sql
  OPEN c1
END FUNCTION

FUNCTION show_list()
  DEFINE arr DYNAMIC ARRAY OF RECORD
             id INTEGER,
             fname VARCHAR(30),
             lname VARCHAR(30)
         END RECORD
  DEFINE cnt, ofs, len, row, i INTEGER,
         key STRING, rev BOOLEAN

  CALL execute_sql(NULL)
  DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=-1)
    ON SORT
       LET key = DIALOG.getSortKey("sa")
       LET rev = DIALOG.isSortReverse("sa")
       IF key IS NULL THEN
          CALL execute_sql( NULL )
       ELSE
          -- Assuming that form field names match table column names
          CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," ") )
       END IF
    ON FILL BUFFER
       CALL arr.clear()
       LET ofs = fgl_dialog_getBufferStart()
       LET len = fgl_dialog_getBufferLength()
       LET row = ofs
       FOR i=1 TO len
          FETCH ABSOLUTE row c1 INTO arr[i].*
          IF SQLCA.SQLCODE!=0 THEN
            CALL DIALOG.setArrayLength("sa",row-1)
            EXIT FOR
          END IF
          LET row = row + 1
       END FOR
  END DISPLAY

END FUNCTION
```

Note that with the above example, the current row remains at the same position: When the table is sorted, the set of rows provided in the `ON FILL BUFFER` may not include the database row that was the current row before the sort.

To track the current row, store the primary key value of the current row before re-executing the query. After query execution, scan the cursor result set and perform a `DIALOG.setCurrentRow()` when the primary key of the current row is found. The current row might be outside the row set provided in `ON FILL BUFFER`. In order to make setCurrentRow() work properly, you have to count the total number of rows before the `DISPLAY ARRAY`:

```
...
DEFINE cnt, ofs, len, row, i INTEGER,
       key STRING, rev BOOLEAN,
       row_count, curr_id, last_id INTEGER

...

SELECT COUNT(*) INTO row_count FROM customer

CALL execute_sql(NULL)
DISPLAY ARRAY arr TO sa.* ATTRIBUTES(COUNT=row_count)
  ON SORT
     LET row = DIALOG.getCurrentRow("sa")
     FETCH ABSOLUTE row c1 INTO last_id
     LET key = DIALOG.getSortKey("sa")
     LET rev = DIALOG.isSortReverse("sa")
     IF key IS NULL THEN
        CALL execute_sql( NULL )
     ELSE
        -- Assuming that form field names match table column names
        CALL execute_sql( "ORDER BY " || key || IIF(rev," DESC"," ") )
     END IF
     LET row=1
     WHILE TRUE
         FETCH c1 INTO curr_id
         IF SQLCA.SQLCODE==100 THEN
            ERROR "Last current row disappeared from result set!"
            EXIT PROGRAM 1
         END IF
         IF curr_id == last_id THEN
            CALL DIALOG.setCurrentRow("sa",row)
            EXIT WHILE
         END IF
         LET row = row+1
     END WHILE
  ON FILL BUFFER
     ...
```

INPUT ARRAY row modifications
Controlling row creation and deletion in an editable record list.

The `INPUT ARRAY` instruction handles record list edition. This controller allows the user to directly edit existing rows and to create or remove rows with implicit actions.

The following implicit actions are created by default by the `INPUT ARRAY` dialog:

- `insert`: creates a new row before the current row. If there are no rows in the list, the action adds a new row.
- `append`: creates a new row after the last row of the list.
- `delete`: deletes the current row.

To prevent INPUT ARRAY to create the implicit "insert", "append" and "delete" actions, set respectively the INSERT ROW, APPEND ROW, or DELETE ROW control attributes to FALSE. To fully deny row addition, set also the AUTO APPEND attribute to FALSE.

```
...
  INPUT ARRAY p_items FROM sa.*
    -- Allow only row append and delete implicit actions.
    ATTRIBUTES(AUTO APPEND=FALSE,
               INSERT ROW=FALSE)
      ...
  END INPUT
...
```

Specific control blocks are available to take control when a row is created or deleted:

- BEFORE INSERT and AFTER INSERT control blocks can be used to control row creation. Cancel a row creation with CANCEL INSERT in BEFORE INSERT or AFTER INSERT blocks.
- BEFORE DELETE and AFTER DELETE control blocks can be used to control row deletion. Cancel row deletion with the CANCEL DELETE instruction in BEFORE DELETE.

Dynamic arrays and the ui.Dialog class provide methods such as array.deleteElement() or ui.Dialog.appendRow() to modify the list. When using these methods, the predefined triggers such as BEFORE DELETE or BEFORE INSERT are not executed. While it is safe to use these methods within a DISPLAY ARRAY, you must take care when using an INPUT ARRAY. For example, you should not call such methods in triggers like BEFORE ROW, AFTER INSERT, BEFORE DELETE.

Users can append temporary rows by moving to the end of the list, or when executing the append action. Appending temporary rows is a different from inserting a row; an appended row is considered temporary until the user modifies a field while an inserted row remains in the list even if the user does not modify a field.

By default, when the last row is removed by a delete action, the INPUT ARRAY instruction will automatically create a new temporary row at the same position. The visual effect of this behavior can be misinterpreted - if no data was entered in the last row, you can't see any difference. However, the last row is really deleted and a new row is created, and the BEFORE DELETE / AFTER DELETE / AFTER ROW / BEFORE ROW / BEFORE INSERT control block sequence is executed. In order to deny to avoid the creation of a new temporary row when the last row is deleted, set AUTO APPEND = FALSE attribute.

The insert, append or delete actions will be automatically disabled according to the context: If the INPUT ARRAY is using a static array that becomes full, or if the MAXCOUNT attribute is reached, both insert and append actions will be disabled. The delete action is automatically disabled when AUTO APPEND = FALSE and there are no more rows in the array.

INPUT ARRAY temporary rows
Temporary rows can be created at the end of an editable record list.

In record list controlled by an INPUT ARRAY, the user can create a new *temporary row* at the end of the list: The new row is called "temporary" because it will be automatically removed if the user leaves the row without entering data. If data is entered by the user or by program (setting the touched flag), the temporary row becomes permanent.

A temporary row is promoted to a permanent row under certain conditions described in this topic. We distinguish also *explicit* temporary row creation from *automatic* temporary row creation.

Temporary row creation is different from adding new rows with the DIALOG.appendRow() method; When appending a row by program, the row is considered permanent and remains in the list even if the user did not enter data in fields.

Conditions to make a temporary row permanent

The temporary row is made permanent, when moving down to the next new temporary row, or if the modification flag of one of the fields is set. The modification flag of a field is typically set when the user enters data in the form field and tabs to another field (or validates the dialog), but this modification flag can also be set by program, with a `DISPLAY TO / BY NAME` instruction or with the `DIALOG.setFieldTouched()` method. When the modification is set by program, `NOENTRY` fields are ignored, however, fields dynamically disabled by `DIALOG.setFieldActive()` are taken into account.

Explicit temporary row creation

Explicit temporary row creation takes place when the user decided to append a new row explicitly with the append action. If the list is empty, an insert action will have the same effect as an append action (i.e. a temporary row will be created at position 1).

Automatic temporary row creation

By default, automatic temporary row creation takes place when:

- The user tries to move below the last row, with a Down keystroke or with the mouse.
- The user presses the Tab key when in the last field of the last row.
- The list has the focus and the last row of the list is deleted by an implicit delete action.
- The list has the focus and the last row of the list is deleted by program with `DIALOG.deleteRow()` or `DIALOG.deleteAllRows()`.
- When the `INPUT ARRAY` is in a `DIALOG` block, the list has no rows and gets the focus (A new temporary row is created to let the user enter data immediately)

Avoiding temporary row creation

Temporary row creation is useful because, in most cases, `INPUT ARRAY` is used to edit existing rows and append new rows at the end of the list. However, you might want to deny row addition or at least avoid the automatic temporary row creation when the last row is deleted or when an empty list gets the focus.

To avoid explicit temporary row creation, prevent `INPUT ARRAY` to defined the implicit append action by setting the `APPEND ROW` attribute to `FALSE` in the `ATTRIBUTE` clause:

```
...
   INPUT ARRAY p_items FROM sa.* ATTRIBUTES(APPEND ROW=FALSE)
   ...
   END INPUT
...
```

Even if `APPEND ROW` /`INSERT ROW` attributes are set to `FALSE`, automatic temporary row can still occur when the user deletes the last row of the list or if the list is empty when the `INPUT ARRAY` is entered. Without automatic temporary row creation, an `INPUT ARRAY` instruction would have no rows to edit if the array is empty. To avoid automatic temporary row creation is such cases, set the `AUTO APPEND` attribute to `FALSE`:

```
...
   INPUT ARRAY p_items FROM sa.* ATTRIBUTES(AUTO APPEND=FALSE)
   ...
   END INPUT
...
```

To fully deny row addition, set both `APPEND ROW` and `AUTO APPEND` to `FALSE`.

If both `APPEND ROW` and `INSERT ROW` attributes are set to `FALSE`, the dialog will deny explicit temporary row creation but also automatic temporary row creation, as if `AUTO APPEND = FALSE` would be used.

Row creation control blocks for temporary rows

In order to control row creation, use the `BEFORE INSERT` and `AFTER INSERT` control blocks. The `BEFORE INSERT` trigger is invoked after a new row was inserted or appended, just before the user gets control to enter data in fields. Regarding temporary rows, the `AFTER INSERT` block is invoked if data has been entered and you leave the new row (for example, when the focus moves to another row or leaves the current list), or if the dialog is validated, for example with `ACCEPT DIALOG` in case of `DIALOG` (or `ACCEPT INPUT` in case of singular `INPUT ARRAY`). No `AFTER INSERT` block is invoked if the user did not enter data: The temporary row is automatically deleted.

In the `BEFORE INSERT` control block, you can tell if a row is a temporary appended one by comparing the current row (`DIALOG.getCurrentRow()` or `ARR_CURR()`) with the total number of rows (`DIALOG.getArrayLength()` or `ARR_COUNT()`). If the current row index equals the row count, you are in a temporary row.

AFTER ROW and temporary rows

When a temporary row as automatically removed, the `AFTER ROW` block will be executed for the temporary row, but `ui.Dialog.getCurrentRow()` / `ARR_CURR()` will be one row greater than `DIALOG.getArrayLength()` / `ARR_COUNT()`. In this case, ignore the `AFTER ROW` event.

DISPLAY ARRAY modification triggers

Using dedicated interaction blocks to allow the user to modify a read-only record list.

The `DISPLAY ARRAY` block implements by default a read-only list of records. The end user can navigate in the list, but cannot modify the rows.

The traditional way to implement an editable list of record is to use `INPUT ARRAY`. However, `INPUT ARRAY` uses ergonomics that may not correspond to the end user expectations. Basically, a list controlled by an `INPUT ARRAY` is always in "edit mode": the focus is in a field and the user can modify the current field. When moving up or down in the list, the edit cursor jumps to the upper or lower cell.

Other GUI applications use a different pattern, with read-only lists that can switch to edit mode when a specific action is fired. To implement such ergonomics, use the `ON INSERT`, `ON APPEND`, `ON UPDATE`, `ON DELETE` modification triggers to control row insertion, appending, modification and deletion in a `DISPLAY ARRAY` block.

Cell color attributes

List controllers can display every cell in a specific color.

When using the `DISPLAY ARRAY` or `INPUT ARRAY`, you can assign specific colors to cells of a `TABLE` or `TREE` rows with the `DIALOG.setArrayAttributes()` or `DIALOG.setCellAttributes()` method.

Call the method in the dialog initialization clause, for example, in `BEFORE DISPLAY` for a singular `DISPLAY ARRAY` dialog.

The method takes an array as parameter. This array must have the same structure as the data array, but each element of the record must be a string. Attributes can be set for individual cells by using the TTY attributes (see method reference for possible values).

If cell attribute values are changed in during the dialog execution, use the `UNBUFFERED` mode to get automatic form synchronization. The unbuffered mode is not required if the cell attributes are defined before executing the dialog, and leave unchanged until the dialog ends.

Example

This is the `list.per` form file defining the table view:

```
LAYOUT
TABLE
{
[c1         |c2             ]
```

```
}
END
END
ATTRIBUTES
c1 = FORMONLY.key;
c2 = FORMONLY.name;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

This is the program code (main.4gl):

```
MAIN
    DEFINE arr DYNAMIC ARRAY OF RECORD
                key INTEGER,
                name VARCHAR(100)
            END RECORD
    DEFINE att DYNAMIC ARRAY OF RECORD
                key STRING,
                name STRING
            END RECORD
    DEFINE I INT

    FOR i=1 TO 10
        LET arr[i].key = i
        LET arr[i].name = "Item "||i
        LET att[i].key = "red reverse"
        LET att[i].name = IIF(i MOD 2,"blue","green")
    END FOR

    OPEN FORM f1 FROM "list"
    DISPLAY FORM f1

    DISPLAY ARRAY arr TO sr.* ATTRIBUTES(UNBUFFERED)
        BEFORE DISPLAY
           CALL DIALOG.setCellAttributes(att)
       ON ACTION att_modify_cell
         LET att[2].key = "red reverse"
       ON ACTION att_clear_cell
         LET att[2].key = NULL
    END DISPLAY

END MAIN
```

Multiple row selection

Multiple row selection allows the end user to select several rows in a list of records.

The `DISPLAY ARRAY` controller supports multiple row selection when the `ON SELECTION CHANGE` block is defined, or by enabling the feature with the `ui.Dialog.setSelectionMode()` method when the dialog starts. The `setSelectionMode()` method can also be used to enable or disable the multi-row selection during the dialog execution.

> **Important:** This feature is not supported on mobile platforms.

When multi-row selection is enabled, the end user can select one or several rows with the standard keyboard and mouse click combinations. When the end user selects or de-selects rows, the `ON SELECTION CHANGE` block is fired, if defined. The program can then query the `DIALOG.isRowSelected()` method to check for selected rows.

```
DISPLAY ARRAY arr TO sr.*
```

```
  ...
  ON SELECTION CHANGE
     FOR i=1 TO DIALOG.getArrayLength("sr")
         DISPLAY SFMT("Row: %1 s=%2", i, DIALOG.isRowSelected("sr", i) )
     END FOR
  ON ACTION enable_mrs
     CALL DIALOG.setSelectionMode( "sr", 1 )
  ON ACTION disable_mrs
     CALL DIALOG.setSelectionMode( "sr", 0 )
  ...
END DISPLAY
```

Multiple row selection is GUI-specific and therefore can't be used in TUI mode.

With multiple row selection, you must distinguish between two concepts: *row selection* and *current row*. In GUI mode, a selected row usually has a blue background, while the current row has a dotted focus rectangle. The current row may not be selected, or a selected row may not be the current row. When the default single-row selection is used, the current row is always selected automatically.

If the `ON SELECTION CHANGE` block is not required, use the `ui.Dialog.setSelectionMode()` method to enable multi-row selection for the dialog:

```
DISPLAY ARRAY arr TO sr.*
  BEFORE DISPLAY
     CALL DIALOG.setSelectionMode( "sr", 1 )
  ...
END DISPLAY
```

Note however that without the `ON SELECTION CHANGE` trigger, it is not possible to detect row selection change when staying on the current row, since no `BEFORE ROW` / `AFTER ROW` trigger is fired in this case.

Row selection flags can be changed by program for a range of rows with the `DIALOG.setSelectionRange()` method.

The `DISPLAY ARRAY` dialog implements an implicit row-copy feature: The selected rows can be dragged to another dialog or external program, or the end-user can do an "editcopy" predefined action (Ctrl-C shortcut), to copy the selected rows to the front-end clipboard. The row-copy feature works also when multiple row selection is disabled, but only the current row will be dragged or copied to the front-end clipboard.

If you delete, insert or append rows in the program array with methods such as `array.deleteElement()`, selection information is not synchronized: To sync the selection flags with the data rows, use dialog methods like `DIALOG.insertRow()` (or `DIALOG.insertNode()` for tree-views), .

Behavior of ui.Dialog class methods with multiple row selection

Table 291: Effect of ui.Dialog class on selection flags when multi-range selection is enabled

Dialog class method	Effect on multiple row selection
`appendRow()`	Selection flags of existing rows are unchanged. New row is appended at the end of the list with selection flag set to zero.
`appendNode()`	Selection flags of existing rows are unchanged. New node is appended at the end of the tree with selection flag set to zero.
`deleteAllRows()`	Selection flags of all rows are cleared.

Dialog class method	Effect on multiple row selection
deleteRow()	Selection flags of existing rows are unchanged. Selection information is synchronized (i.e., shifted up) for all rows after the deleted row.
deleteNode()	Selection flags of existing rows are unchanged. Selection information is synchronized (i.e., shifted up) for all nodes after the deleted node.
insertRow()	Selection flags of existing rows are unchanged. Selection information is synchronized (i.e., shifted down) for all rows after the new inserted row.
insertNode()	Selection flags of existing rows are unchanged. Selection information is synchronized (i.e., shifted down) for all nodes after the new inserted node.
setArrayLength()	Selection flags of existing rows are unchanged. If the new array length is larger than the previous length, selection flags of new rows are not initialized to zero.
setCurrentRow()	Selection flags of all rows are reset, and the new current row gets selected.
setSelectionMode()	When you switch off multiple row selection, the selection flags of existing rows are cleared.

Examples
Example 1: Simple list view

The form file `table.per` (grid-based layout):

```
LAYOUT
TABLE (DOUBLECLICK=myselect)
{
[c1         |c2             ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.key;
c1 = FORMONLY.name, IMAGECOLUMN=image;
PHANTOM FORMONLY.image;
c2 = FORMONLY.detail;
END
INSTRUCTIONS
SCREEN RECORD list1(FORMONLY.*);
END
```

The form file `table.per` (stack-based layout):

```
LAYOUT
STACK
 TABLE list1(DOUBLECLICK=myselect)
```

```
   PHANTOM FORMONLY.key;
   EDIT FORMONLY.name,
        IMAGECOLUMN=image, TITLE="Name";
   PHANTOM FORMONLY.image;
   EDIT FORMONLY.detail, TITLE="Detail";
 END
END
END
```

The program `main.4gl`:

```
MAIN
    DEFINE arr DYNAMIC ARRAY OF RECORD
                key INTEGER,
                name STRING,
                image STRING,
                detail STRING
            END RECORD,
            i INTEGER
    FOR i=1 TO 60
        LET arr[i].key = i
        LET arr[i].name = SFMT("Item %1", i)
        IF i MOD 2 THEN
           LET arr[i].image = "file"
        ELSE
           LET arr[i].image = "smiley"
        END IF
        LET arr[i].detail = SFMT("This is item %1", i)
    END FOR
    OPEN FORM f1 FROM "table"
    DISPLAY FORM f1
    DISPLAY ARRAY arr TO list1.* ATTRIBUTES(UNBUFFERED)
        ON ACTION myselect
           MESSAGE "myselect:", arr_curr()
    END DISPLAY
END MAIN
```

Tree views

Describes tree view programming in the language.

- Understanding tree-views on page 1385
- Defining a TREE container on page 1386
- Defining the program array for tree-views on page 1388
- Filling the program array with rows on page 1389
- Controlling a tree-view with DISPLAY ARRAY on page 1390
- Modifying the tree during dialog execution on page 1390
- Using regular DISPLAY ARRAY control blocks on page 1391
- Dynamic filling of very large trees on page 1391
- Built-in sort and tree-views on page 1391
- Multi-row selection and tree-views on page 1392
- Drag and drop in tree-views on page 1392
- Examples on page 1392
  - Example 1: Static tree view (filled before dialog starts) on page 1392
  - Example 2: Dynamic tree view (filled on demand) on page 1393

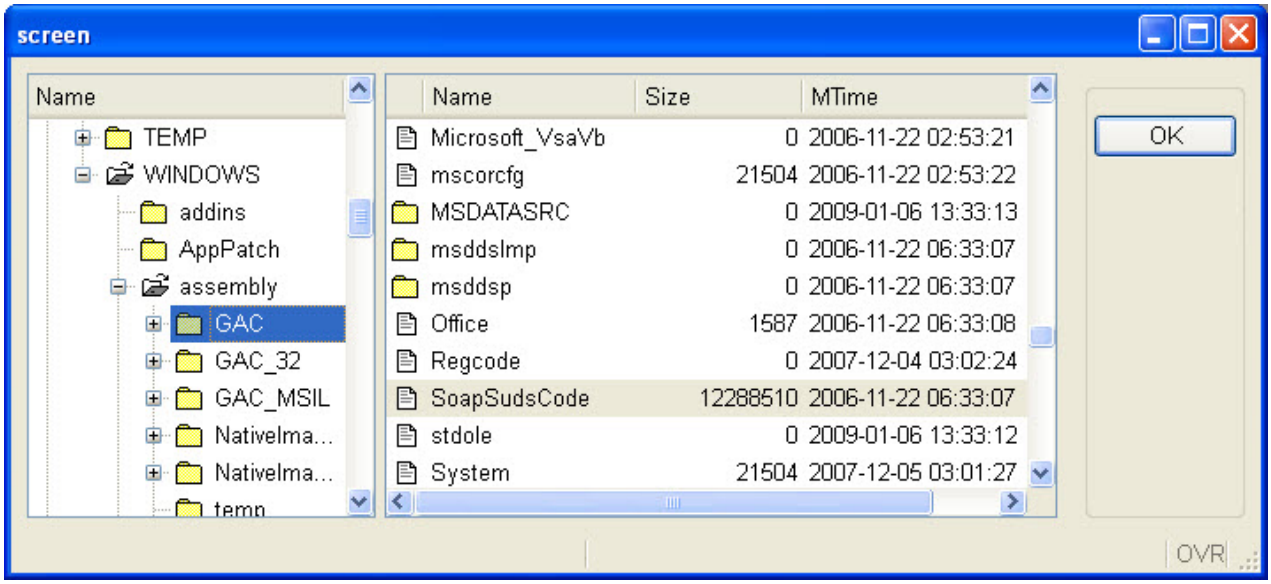
Understanding tree-views

Tree-views can be implemented with a `DISPLAY ARRAY` instruction using a form screen-array bound to a `TREE` container with tree-view specific attributes. `TREE` containers are very similar to `TABLE` containers, except that the first columns are used to display a tree of nodes on the right of the widget.

> **Important:** This feature is not supported on mobile platforms.

The next screen-shot shows a typical file browser using a tree-view. This example implements a `DIALOG` instruction with two `DISPLAY ARRAY` sub-dialogs. The first `DISPLAY ARRAY` sub-dialog controls the tree-view while the second one controls the file list on the right side.

Figure 97: Form with Tree View



The data used to display tree-view nodes must be provided in a program array and controlled by a `DISPLAY ARRAY`. It is possible to control a tree view table with a singular `DISPLAY ARRAY` or with a `DISPLAY ARRAY` sub-dialog within a `DIALOG` instruction.

A tree view model is implemented with a flat program array (i.e. a list of rows), where each row defines parent/child node identifiers to describe the structure of the tree; so, the order of the rows matters:

```
Tree structure          parent-id        child-id
Node 1                  NULL              1
     Node 1.1            1                1.1
     Node 1.2            1                1.2
         Node 1.2.1      1.2              1.2.1
         Node 1.2.2      1.2              1.2.2
         Node 1.2.3      1.2              1.2.3
     Node 1.3            1                1.3
         Node 1.3.1      1.3              1.3.1
  ...                    ...              ....
```

Depending on your need, you can fill the program array with all rows of the tree before dialog execution, or you can fill or reduce the list of nodes dynamically upon expand / collapse action events. In the second case, you must provide additional information for each row of the program array, to indicate whether the node has children. A dynamic build of the tree view allows you to implement programs displaying very large trees, for example in a bill of materials application, where thousands of elements can be assembled together.

Tree-views can display additional columns for each node, to show specific row data as in a regular table.

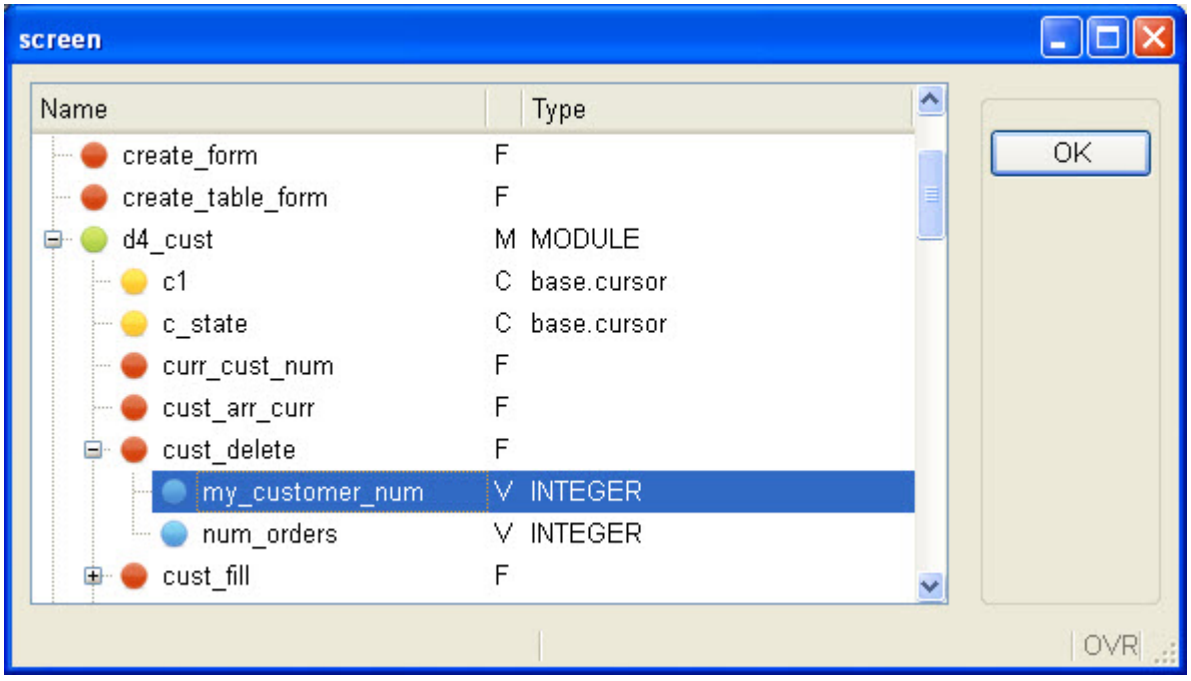


Figure 98: Tree-view with additional columns

Defining a TREE container

Create a form specification file containing a TREE container bound to a screen array. The screen array identifies the presentation elements to be used by the runtime system to display the tree-view and the additional columns.

A TREE container must be present in the LAYOUT section of the form, defining the columns of the tree-view list. The TREE container must hold at least one column defining the node texts (or names). This column will be used on the front-end side to display the tree-view widget. Additional columns can be added in the TREE container to display node information. The TREE container attributes must be declared in the ATTRIBUTES section of the form.

Secondary form fields have to be used to hold tree node information such as icon image, parent node id, current node id, expanded flag and parent flag. While these secondary fields can be defined as regular form fields and displayed in the tree-view list, we recommend that you use PHANTOM fields instead: Phantom fields can be listed in the screen-array but do not need to be part of the LAYOUT section. Phantom fields will only be used by the runtime system to build the tree of nodes.

Example of tree-view definition using a TREE container:

```
LAYOUT
TREE mytree ( PARENTIDCOLUMN=parentid, IDCOLUMN=id,
              EXPANDEDCOLUMN=expanded, ISNODECOLUMN=isnode )
{
 Tree
[name                       |desc     ]
[name                       |desc     ]
[name                       |desc     ]
[name                       |desc     ]
[name                       |desc     ]
}
END
END
ATTRIBUTES
EDIT name = FORMONLY.name, IMAGECOLUMN=image;
```

```
PHANTOM FORMONLY.image;
PHANTOM FORMONLY.parentid;
PHANTOM FORMONLY.id;
PHANTOM FORMONLY.expanded;
PHANTOM FORMONLY.isnode;
EDIT desc = FORMONLY.description;
END
INSTRUCTIONS
SCREEN RECORD sr( FORMONLY.* );
END
```

Example of tree-view definition using the <Tree> layout tag inside a `GRID` container, with a `TREE` form element to define attributes in the `ATTRIBUTES` section:

```
LAYOUT
GRID
{
<Tree tv                            >
 Tree
[name                      |desc     ]
[name                      |desc     ]
[name                      |desc     ]
[name                      |desc     ]
[name                      |desc     ]
<                                   >
}
END
END
ATTRIBUTES
TREE tv: mytree,
     PARENTIDCOLUMN=parentid, IDCOLUMN=id,
     EXPANDEDCOLUMN=expanded, ISNODECOLUMN=isnode;
EDIT name = FORMONLY.name, IMAGECOLUMN=image;
PHANTOM FORMONLY.image;
PHANTOM FORMONLY.parentid;
PHANTOM FORMONLY.id;
PHANTOM FORMONLY.expanded;
PHANTOM FORMONLY.isnode;
EDIT desc = FORMONLY.description;
END
INSTRUCTIONS
SCREEN RECORD sr( FORMONLY.* );
END
```

The first visual column ("name" in example) must be the field defining the node names, and the widget must be an `EDIT` or `LABEL`.

Several attributes are used to configure a `TREE` form element:

- The `PARENTIDCOLUMN` and `IDCOLUMN` attributes are respectively used to identify the form field containing the identifiers of the parent and child nodes, defining the structure of the tree. You must specify form field column names, not item tag identifiers (used to reference a form item in the layout section). If these attributes are not specified, the parent node id and node id field names default respectively to "*parentid*" and "*id*".
- The `EXPANDEDCOLUMN` attribute can be used to define the form field holding the flag indicating that a node is expanded (i.e. opened).
- If the `ISNODECOLUMN` attribute is used, it defines the form field indicating that a node has children, even if the program array does not contain child nodes for that parent node. This attribute must be used to implement dynamic filling of tree-views.

- The IMAGEEXPANDED, IMAGECOLLAPSED and the IMAGELEAF attributes are optional attributes defining global images for expanded, collapsed and leaf nodes. You should use these attributes if you want to display the same icons for all nodes.
- The IMAGEEXPANDED and IMAGECOLLAPSED instruct the runtime system to set a specific icon when a node gets expanded or collapsed. The IMAGELEAF attribute defines the global icon for leaf nodes. This saves the programmer from writing code to display common node images.

Tree-view definition must be completed with form fields declaration. These must be defined in the ATTRIBUTES section. The fields not used for display are declared as PHANTOM fields. The tree-view form fields must be grouped in a screen-array declared in the INSTRUCTIONS section.

The form fields required to declare a tree-view table are the following.

Table 292: Form fields required to declare a tree-view table

Description	Field type	Tree attribute to define the field	Mandatory	Default name
Text to be displayed for the node	EDIT	N/A	yes	N/A
Id of the node	PHANTOM	IDCOLUMN	yes	id
Id of the parent node	PHANTOM	PARENTIDCOLUMN	yes	parentid
Icon image for a node	PHANTOM	IMAGECOLUMN	no	N/A
Node expansion indicator	PHANTOM	EXPANDEDCOLUMN	no	no
Parent node indicator	PHANTOM	ISNODECOLUMN	no	no

The first three fields (node text, parent id and node id) are mandatory, and that the first visual (non-phantom) field listed in the screen array will be implicitly used to hold the text of tree-view nodes.

Additional fields (like the *desc* field in this example) can be defined to display details for each node in regular columns, that will appear on the right of the tree widget.

The order of the fields in the screen array of the tree-view does not matter, but it must of course match the order of the corresponding variables in the record-array of the program.

If you need to display node-specific images, define a phantom field to hold node images and attach it to the tree-view definition by using the IMAGECOLUMN attribute. Alternatively you can globally define images for all nodes with the IMAGEEXPANDED, IMAGECOLLAPSED and the IMAGELEAF attributes of the TREE form element.

Defining the program array for tree-views

In the program code, define a dynamic array of records with the DEFINE instruction. The DISPLAY ARRAY dialog will use that program array as the model for the tree-view list. A tree of nodes will be automatically built according to the data found in the program array. The front-end can then render the tree of nodes in a tree-view widget.

The members of the program array must correspond to the elements of the screen-array bound to the TREE container, by number and data types.

The name of the array members does not matter; the purpose of each member is defined by the name of the corresponding screen-array members declared in the form file. Program array members and screen-array members are bound by position.

The next code example defines a program array with a member structure corresponding to the screen-array defined in the form example of the previous section.

```
DEFINE tree_arr DYNAMIC ARRAY OF RECORD
       name STRING,       -- text to be displayed for the node
       pid STRING,        -- id of the parent node
       id STRING,         -- id of the current node
       image STRING,      -- name of the image file for the node (can be
 null)
       expanded BOOLEAN,  -- node expansion flag (TRUE/FALSE) (optional)
       isnode BOOLEAN,    -- children indicator flag (TRUE/FALSE) (optional)
       description STRING -- user field describing the node
END RECORD
```

The *name*, *pid*, *id* members are mandatory. These hold respectively the node text, parent and current node identifiers that define the structure of the tree.

The *image* member will hold the name of the little icon to be displayed for each node and leaf. You can omit this member, if you do not want to display images, or when then tree defines default images with the `IMAGEEXPANDED`, `IMAGECOLLAPSED` and the `IMAGELEAF` attributes.

The *expanded* member can be used to handle node expansion by program. You can query this member to check whether a node is expanded, or set the value to expand a specific node.

The *isnode* member can be used to indicate whether a given node has children, without filling the array with rows defining the child nodes. This information will be used by front-ends to decorate a node as a parent, even if no children are present. The program should then fill the array with child nodes when an expand action is invoked, to implement dynamic tree-views).

The program array can hold more columns (like the *description* field), which can be displayed in regular table columns as part of a node's data.

Remember the order of the program array members must match the screen-array members in the form file, but this order can be different from the column order used in the layout, with the exception of the first column defining the text of nodes (i.e. *name* field in example).

Filling the program array with rows

Once the program array is defined according to the screen-array of the tree-view table, fill the array with the tree-view definition.

You can directly fill the program array before the dialog execution. Once the dialog has started, you must use the methods `DIALOG.insertNode()`, `DIALOG.appendNode()` and `DIALOG.deleteNode()`, if you want to modify the tree, otherwise information like multi-range selection flags and cell attributes will not be synchronized.

Fill the rows in the correct order defining the structure of the tree, to reflect the parent/child relationship of the tree nodes. If a row defines a tree-view node with a parent identifier that does not exist, or if the child row is inserted under the wrong parent row, the orphan row will become a new node at the root of the tree.

In order to fill the program array with database rows defining the tree structure, you will need to write a recursive function, keeping track of the current level of the nodes to be created for a given parent.

The next example shows how to fill the array with data coming from a database table having the following structure:

```
CREATE TABLE dbtree (
     id SERIAL NOT NULL,
     parentid INTEGER NOT NULL,
     name VARCHAR(20) NOT NULL
   )
```

The difficulty with fetching a tree from a database table is in the cursor management, which can not be used recursively. A workaround for this problem is to fetch all the children of a given node at once, then call the function recursively for each of the fetched nodes:

```
TYPE tree_t RECORD
       id INTEGER,
       parentid INTEGER,
       name VARCHAR(20)
    END RECORD

DEFINE tree_arr tree_t

FUNCTION fetch_tree(pid)
    DEFINE pid, i, j, n INTEGER
    DEFINE a DYNAMIC ARRAY OF tree_t
    DEFINE t tree_t

    DECLARE cu1 CURSOR FOR SELECT * FROM dbtree WHERE parentid = pid
    LET n = 0
    FOREACH cu1 INTO t.*
       LET n = n + 1
       LET a[n].* = t.*
    END FOREACH

    FOR i = 1 TO n
        LET j = tree_arr.getLength() + 1
        LET tree_arr[j].name = a[i].name
        LET tree_arr[j].id = a[i].id
        LET tree_arr[j].parentid = a[i].parentid
        CALL fetch_tree(a[i].id)
    END FOR

END FUNCTION
```

Controlling a tree-view with DISPLAY ARRAY

After the program array has been filled, you must execute a `DISPLAY ARRAY` dialog.

The next code example implements a `DISPLAY ARRAY` binding the program array called *tree_arr* to the *sr* screen-array, attaching the dialog to the tree table defined in the form:

```
CALL fill_tree(tree_arr)
DISPLAY ARRAY tree_arr TO sr.* ATTRIBUTES(UNBUFFERED)
  BEFORE ROW
    DISPLAY "Current row is: ", DIALOG.getCurrentRow("sr")
END DISPLAY
```

It is not possible to use the `DISPLAY ARRAY` paged mode (`ON FILL BUFFER`) when the decoration is a tree view list. The dialog needs the complete set of open nodes with parent/child relation to handle the tree view display, with the paged mode only a given window of the dataset is known by the dialog. If you use a the paged mode in `DISPLAY ARRAY` with a tree view as decoration, the program will raise an error at runtime.

However, tree-views can be filled dynamically with `ON EXPAND` / `ON COLLAPSE` triggers.

Modifying the tree during dialog execution

During the `DISPLAY ARRAY` execution, it is possible to modify the content of the tree model (i.e. the program array), by inserting, adding or removing nodes by program. However, you should not directly modify the program array: You must use the dialog class methods `DIALOG.insertNode()`, `DIALOG.appendNode()` and `DIALOG.deleteNode()` to modify the tree model. By using these methods, the dialog can synchronize internal data, otherwise the tree display would be corrupted.

It is recommended to be in UNBUFFERED mode to get a front-end synchronization of the tree-view content.

Using regular DISPLAY ARRAY control blocks

If needed, you can implement traditional DISPLAY ARRAY control blocks like BEFORE ROW or AFTER ROW:

```
DISPLAY ARRAY tree_arr TO sr.* ATTRIBUTES(UNBUFFERED)
  BEFORE ROW
    DISPLAY "BEFORE ROW - Current row is: ", DIALOG.getCurrentRow("sr")
  AFTER ROW
    DISPLAY "AFTER ROW  - Current row is: ", DIALOG.getCurrentRow("sr")
END DISPLAY
```

Dynamic filling of very large trees

When a huge tree needs to be displayed, tree data filling can be optimized by creating the nodes on demand. There is no need to fill the complete program array with all possible nodes (down to the last leaf), when only the first levels/branches of the tree are displayed on the screen.

To implement a dynamically filled tree, first define an additional column in the TREE container, to indicate whether a given node has children. That field will be used to render a node with a [+] button, and let the end user click on the node to expand it, even if no child nodes are created yet.

In the DISPLAY ARRAY code, if a node is expanded (or collapsed), the dialog will invoke the ON EXPAND or ON COLLAPSE triggers, to let the program add (or remove) rows in the array, to adapt the tree data dynamically according to navigation events.

```
DEFINE row_index INTEGER
...
DISPLAY ARRAY tree_arr TO sr.* ATTRIBUTES(UNBUFFERED)
  ON EXPAND (row_index)
    DISPLAY "EXPAND   - Expanded row is: ", row_index
    -- Fill with children nodes for tree_arr[row_index]
  ON COLLAPSE (row_index)
    DISPLAY "COLLAPSE - Collapsed row is: ", row_index
    -- Remove children nodes of tree_arr[row_index]
END DISPLAY
```

The program array can be filled directly before the dialog execution, but once the dialog has started, use dialog methods such as DIALOG.insertNode() to modify the tree, otherwise information like multi-range selection flags and cell attributes will not be synchronized. This is typically the case when implementing a dynamically-filled tree with ON EXPAND/ ON COLLAPSE triggers.

Built-in sort and tree-views

By default, the built-in sort is enabled in a TREE container; when the end user clicks on column headers, the runtime system sorts the visual representation of the program array. Tree nodes are ordered by levels; the children nodes are ordered inside a given parent node.

This is a powerful built-in feature. However, in some cases, the tree structure must be static (i.e. the order of the nodes must not change) and you don't want the end user to sort the rows. To prevent the built-in sort, use the UNSORTABLECOLUMNS attribute for the TREE container definition:

```
LAYOUT
...
END
ATTRIBUTES
TREE tv: mytree, UNSORTABLECOLUMNS, ...
...
```

Multi-row selection and tree-views

Multi-row selection can be used with a `DISPLAY ARRAY` controlling a `TREE` container. However, because of the tree-view ergonomic differences with simple tables, the selection of tree nodes follows some specific rules:

1. When selecting a range of nodes, only visible nodes will get the selection flag. For example, if you select all nodes with Ctrl-A, and if the root node is collapsed, only the root node will be selected. This applies also when selecting nodes by program with the `DIALOG.setSelectionRange()`.
2. Collapsing a node will de-select all child nodes.

Drag and drop in tree-views

Drag and drop can be implemented within a `DISPLAY ARRAY` controlling a `TREE` container, with the `ON DRAG*` and `ON DROP` interactive blocks.

The nodes can be moved around in the same tree, can be dropped outside the tree or can be inserted in the tree from external sources.

Examples

Example 1: Static tree view (filled before dialog starts)

Form file "form1.per":

```
LAYOUT
GRID
{
<Tree t1                        >
 Name                    Index
[c1                      |c2    ]
[c1                      |c2    ]
[c1                      |c2    ]
[c1                      |c2    ]
}
END
END

ATTRIBUTES
LABEL c1 = FORMONLY.name;
LABEL c2 = FORMONLY.idx;
PHANTOM FORMONLY.pid;
PHANTOM FORMONLY.id;
PHANTOM FORMONLY.exp;
TREE t1: tree1
    IMAGEEXPANDED  = "open",
    IMAGECOLLAPSED = "folder",
    IMAGELEAF = "file",
    PARENTIDCOLUMN = pid,
    IDCOLUMN = id,
    EXPANDEDCOLUMN = exp;
END

INSTRUCTIONS
SCREEN RECORD sr_tree(name, pid, id, idx, exp);
END
```

Static tree `DISPLAY ARRAY`:

```
DEFINE tree DYNAMIC ARRAY OF RECORD
    name STRING,
    pid STRING,
    id STRING,
    idx INTEGER,
```

```
    expanded BOOLEAN
END RECORD

MAIN
    OPEN FORM f FROM "form1"
    DISPLAY FORM f
    CALL fill(4)
    DISPLAY ARRAY tree TO sr_tree.* ATTRIBUTES(UNBUFFERED)
    BEFORE ROW
        DISPLAY "Current row: ", arr_curr()
    END DISPLAY
END MAIN

FUNCTION fill(max_level)
    DEFINE max_level, p INTEGER
    CALL tree.clear()
    LET p = fill_tree(max_level, 1, 0, NULL)
END FUNCTION

FUNCTION fill_tree(max_level, level, p, pid)
    DEFINE max_level, level INTEGER
    DEFINE p INTEGER
    DEFINE i INTEGER
    DEFINE id, pid STRING
    DEFINE name STRING
    IF level < max_level THEN
        LET name = "Node "
    ELSE
        LET name = "Leaf "
    END IF
    FOR i = 1 TO level
        LET p = p + 1
        IF pid IS NULL THEN
            LET id = i
        ELSE
            LET id = pid || "." || i
        END IF
        LET tree[p].id = id
        LET tree[p].pid = pid
        LET tree[p].idx = p
        LET tree[p].expanded = FALSE
        LET tree[p].name = name || level || '.' || i
        IF level < max_level THEN
            LET p = fill_tree(max_level, level + 1, p, id)
        END IF
    END FOR
    RETURN p
END FUNCTION
```

Example 2: Dynamic tree view (filled on demand)

Form file "form1.per":

```
LAYOUT
GRID
{
<Tree t1                        >
 Name             Description
[c1              |c2           ]
[c1              |c2           ]
[c1              |c2           ]
[c1              |c2           ]
}
```

```
END
END

ATTRIBUTES
LABEL c1 = FORMONLY.name;
PHANTOM FORMONLY.pid;
PHANTOM FORMONLY.id;
PHANTOM FORMONLY.hasChildren;
LABEL c2 = FORMONLY.descr;
TREE t1: tree1
    IMAGEEXPANDED  = "open",
    IMAGECOLLAPSED = "folder",
    IMAGELEAF = "file",
    PARENTIDCOLUMN = pid,
    IDCOLUMN = id,
    ISNODECOLUMN = hasChildren;
END

INSTRUCTIONS
SCREEN RECORD sr_tree(FORMONLY.*);
END
```

Dynamic tree `DISPLAY ARRAY`:

```
DEFINE tree DYNAMIC ARRAY OF RECORD
    name STRING,
    pid STRING,
    id STRING,
    hasChildren BOOLEAN,
    description STRING
END RECORD

MAIN
    DEFINE id INTEGER

    OPEN FORM f FROM "form1"
    DISPLAY FORM f

    LET tree[1].pid = 0
    LET tree[1].id = 1
    LET tree[1].name = "Root"
    LET tree[1].hasChildren = TRUE
    DISPLAY ARRAY tree TO sr_tree.* ATTRIBUTES(UNBUFFERED)
    BEFORE DISPLAY
        CALL DIALOG.setSelectionMode("sr_tree",1)
    ON EXPAND(id)
        CALL expand(DIALOG,id)
    ON COLLAPSE(id)
        CALL collapse(DIALOG,id)
    END DISPLAY
END MAIN

FUNCTION collapse(d,p)
    DEFINE d ui.Dialog
    DEFINE p INTEGER
    WHILE p < tree.getLength()
        IF tree[p + 1].pid != tree[p].id THEN EXIT WHILE END IF
        CALL d.deleteNode("sr_tree", p + 1)
    END WHILE
END FUNCTION

FUNCTION expand(d,p)
    DEFINE d ui.Dialog
```

```
    DEFINE p INTEGER
    DEFINE id STRING
    DEFINE i, x INTEGER
    FOR i = 1 TO 4
        LET x = d.appendNode("sr_tree", p)
        LET id = tree[p].id || "." || i
        LET tree[x].id = id
        -- tree[x].pid is implicitly set by the appendNode() method...
        LET tree[x].name = "Node " || id
        IF i MOD 2 THEN
            LET tree[x].hasChildren = TRUE
        ELSE
            LET tree[x].hasChildren = FALSE
        END IF
        LET tree[x].description = "This is node " || tree[x].name
    END FOR
END FUNCTION
```

Split views

These topics describe split view programming in the language.

- Understanding split views on page 1395
- Creating split view windows on page 1396
- Parallel dialogs for split views on page 1397
- Refreshing a parallel dialog on page 1397
- One or two panes on page 1398
- Switching between panes on page 1398
- Navigator pane on page 1399
- Rendering an HBox as a split view on page 1402
- Examples on page 1403
  - Example 1: Single split view application on page 1403
  - Example 2: Multiple split views with navigation bar on page 1405
  - Example 3: Split view using an HBox on page 1409

Understanding split views

Split views refer to the ability to access two forms side by side on a mobile device. This feature is mainly provided for tablet devices, as most phones can only display one window/form at a time.

A split view is composed by a "*left pane*" and and "*right pane*". In the programs, the panes are implemented with window objects displaying forms, which are controlled by parallel dialogs.

> **Important:** This feature is only for mobile platforms.

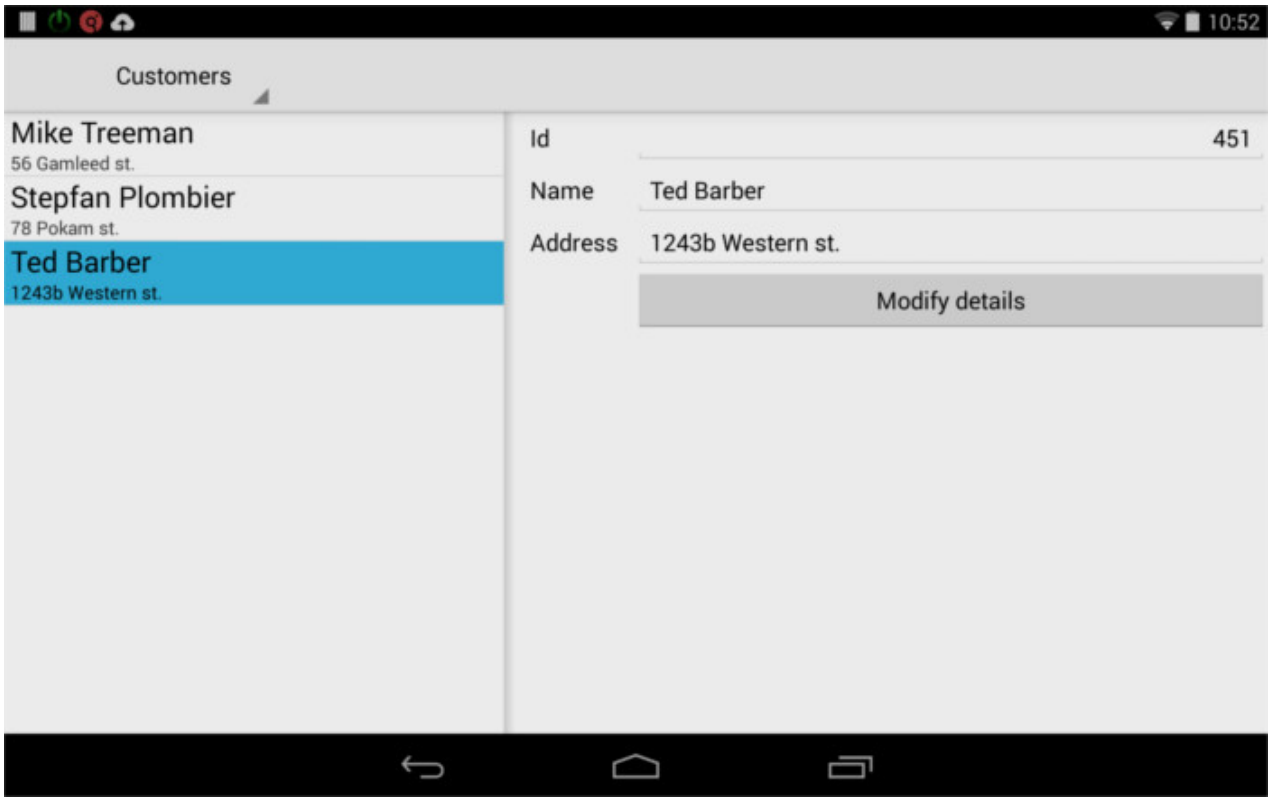


Figure 99: Form with Split View (Android™)

Split views (controlled by parallel dialogs) are typically used to browse in the application data, while modal dialogs are used for data input in a single form. An application based on split views will start with parallel dialogs, and switch to a simple modal dialog when the user chooses to edit application data. Parallel dialog handling is suspended by the runtime system, when a modal dialog executes. For details about parallel dialogs compared to modal dialogs, see Introducing dialogs on page 1250.

If the application displays several split views simultaneously, implement a navigator pane, to let the end user switch between the different split views.

Creating split view windows

The application specifies which window opens in which pane by using the `STYLE` attribute in the `OPEN WINDOW` instruction.

Specify either `LEFT` or `RIGHT` for the `TYPE` attribute, to define a left-hand side pane and a right-hand side pane of the split view respectively.

> **Important:** Both left (`TYPE=LEFT`) and right (`TYPE=RIGHT`) splitview windows need to be created.

This example specifies that the window `w_left` (with the form `customer_list`) display in the left pane, and the window `w_right` (with the form `customer_detail`) display in the right pane:

```
OPEN WINDOW w_left WITH FORM "customer_list" ATTRIBUTES(TYPE=LEFT)
OPEN WINDOW w_right WITH FORM "customer_detail" ATTRIBUTES(TYPE=RIGHT)
```

The window content of both panels will be controlled by parallel dialogs.

> **Important:**
>
> Split-view windows must be the root window (after closing the default `SCREEN` window), or direct children of the `NAVIGATOR` window, if it is used. If regular windows are created before split views, these must be closed:

Case 1: Close regular windows created before the split-views:

```
CLOSE WINDOW screen
OPEN WINDOW w1 WITH FORM "form1"
...
CLOSE WINDOW w1
...
OPEN WINDOW w_left WITH FORM "customer_list" ATTTRIBUTES(TYPE=LEFT)
OPEN WINDOW w_right WITH FORM "customer_detail" ATTRIBUTES(TYPE=RIGHT)
...
```

Case 2: Create split-views as direct NAVIGATOR children

```
CLOSE WINDOW screen
OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNNS ATTRIBUTES(TYPE=NAVIGATOR)
...
OPEN WINDOW w_left WITH FORM "customer_list" ATTTRIBUTES(TYPE=LEFT)
OPEN WINDOW w_right WITH FORM "customer_detail" ATTRIBUTES(TYPE=RIGHT)
...
```

When using a navigator window, the names of the split view windows must match the action names created in the parallel dialog controlling the options of the navigator pane. For more details, see Navigator pane on page 1399.

Parallel dialogs for split views
In order to control the left-hand and right-hand split view content, you must implement two parallel dialogs, each dedicated to a pane.

Create each window and start the parallel dialog for that window. Repeat for each window. When all windows have been created and all dialogs started, run the event loop to activate them.

```
OPEN WINDOW w_left WITH FORM "customer_list"
       ATTRIBUTES(TYPE=LEFT)
START DIALOG d_list_view

OPEN WINDOW w_right WITH FORM "customer_detail"
       ATTRIBUTES(TYPE=RIGHT)
START DIALOG d_detail_view

WHILE fgl_eventLoop()
END WHILE
```

The parallel dialogs must be implemented with a declarative dialog block. See Parallel dialogs (START DIALOG) on page 1199 for more details.

For small iOS devices (not tablets), consider using the `ACCESSORYTYPE=DISCLOSUREINDICATOR` in the `DISPLAY ARRAY` dialog, for left-pane controllers.

Refreshing a parallel dialog
To restart a parallel dialog, use `TERMINATE DIALOG` + `START DIALOG`.

Once the split view parallel dialogs are started, the typically programming pattern to refresh the detail view of the right pane is to restart the detail dialog by executing a `TERMINATE DIALOG` followed by a `START DIALOG`.

The next example shows the case of a list view master (`d_list_view` dialog) displayed on the left pane, which is bound to a detail view of the right pane (`d_detail_view` dialog). The detail information must be refreshed when moving to a new row (BEFORE ROW control block):

```
DIALOG d_list_view()
  DISPLAY ARRAY arr TO sr.*
                ATTRIBUTES(ACCESSORYTYPE=DISCLOSUREINDICATOR)
```

```
      BEFORE ROW -- in BEFORE ROW, we restart the details view
        CURRENT WINDOW IS w_right
        TERMINATE DIALOG d_detail_view
        LET curr_pa = arr_curr()
        DISPLAY BY NAME arr[curr_pa].*
        DISPLAY SFMT("tapped row %1",arr_curr()) TO info
        START DIALOG d_detail_view
        CURRENT WINDOW IS w_left
  END DISPLAY
END DIALOG
```

One or two panes
The same application displays as a split view application with two panes on some devices, yet displays as a single pane on other devices. What controls this?

With split views, you open two windows, assigning one to the left pane and one to the right pane of the split view. Not all mobile devices, however, can display multiple panes on the same screen. While the application code is the same, the mobile client displays either one pane (typical for phones) or two panes (typical for tablets).

If the device only allows a single pane to display, the window in the left pane is the first window displayed.

The rules for single-pane or two-pane display differ according to the mobile platform:

- On Android™ devices, the two-pane mode is activated if the width of the screen is more than 900 dp (density-independent pixels). The width of the screen depends on the orientation; you may notice that you have two panes when the tablet is held in landscape mode (width greater than height), yet only one pane when the tablet is held in portrait mode (height greater than width).

  Note: A density-independent pixel (dp) is an abstract unit that is based on the physical density of the screen. The unit is relative to a 160 dpi screen, so one dp is one pixel on a 160 dpi screen. The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion.
- On iOS devices:
  - With the iPad, the two-pane mode is activated, regardless of the orientation of the tablet.
  - With the iPhone or iTouch devices, only a single pane displays.

Switching between panes
How to switch between the left and right panes of a split view depends on the mobile platform and the ergonomic standards of that platform.

Switching between panes by program

After creating the split view windows and starting the parallel dialogs to control them, the application program can switch between the left and right panes of a split view by selecting the corresponding window with the `CURRENT WINDOW IS` instruction.

```
CURRENT WINDOW IS w_customers
```

Switching between panes on phone devices

On a mobile device (such as phones) that only displays one split view pane at the time, switching from the left pane to the right pane is handled automatically by the front-end.

Note: The ergonomics and rendering depend on the device's operating system.

When starting the application, the left-pane is displayed first. This pane typically uses a table view controlled by a `DISPLAY ARRAY` dialog.

On an iOS phone, consider using the ACCESSORYTYPE=DISCLOSUREINDICATOR in the DISPLAY ARRAY dialog of left-pane controllers.

If the end user taps on a row in the list of the left pane, the right pane is automatically shown. To avoid this implicit switch from the left to the right pane, define a DOUBLECLICK = *action-name* attribute in the DISPLAY ARRAY dialog, and bind this action to an ON ACTION handler which does not change the current window.

Once the right pane is displayed, the user can switch to the left pane:

- On an Android™ phone, press the physical back button.
- On an iOS phone, press the back arrow on the top left of the window.

> **Important:** This automatic "back to left panel" option is only possible if the dialog on the right side does not have a `close`, `cancel` or `accept` action defined. If one of these actions are defined, it will be attached to the back button, and that action will be executed when pressed.

Navigator pane
A *navigator pane* enables access to several views in an application from a main panel.

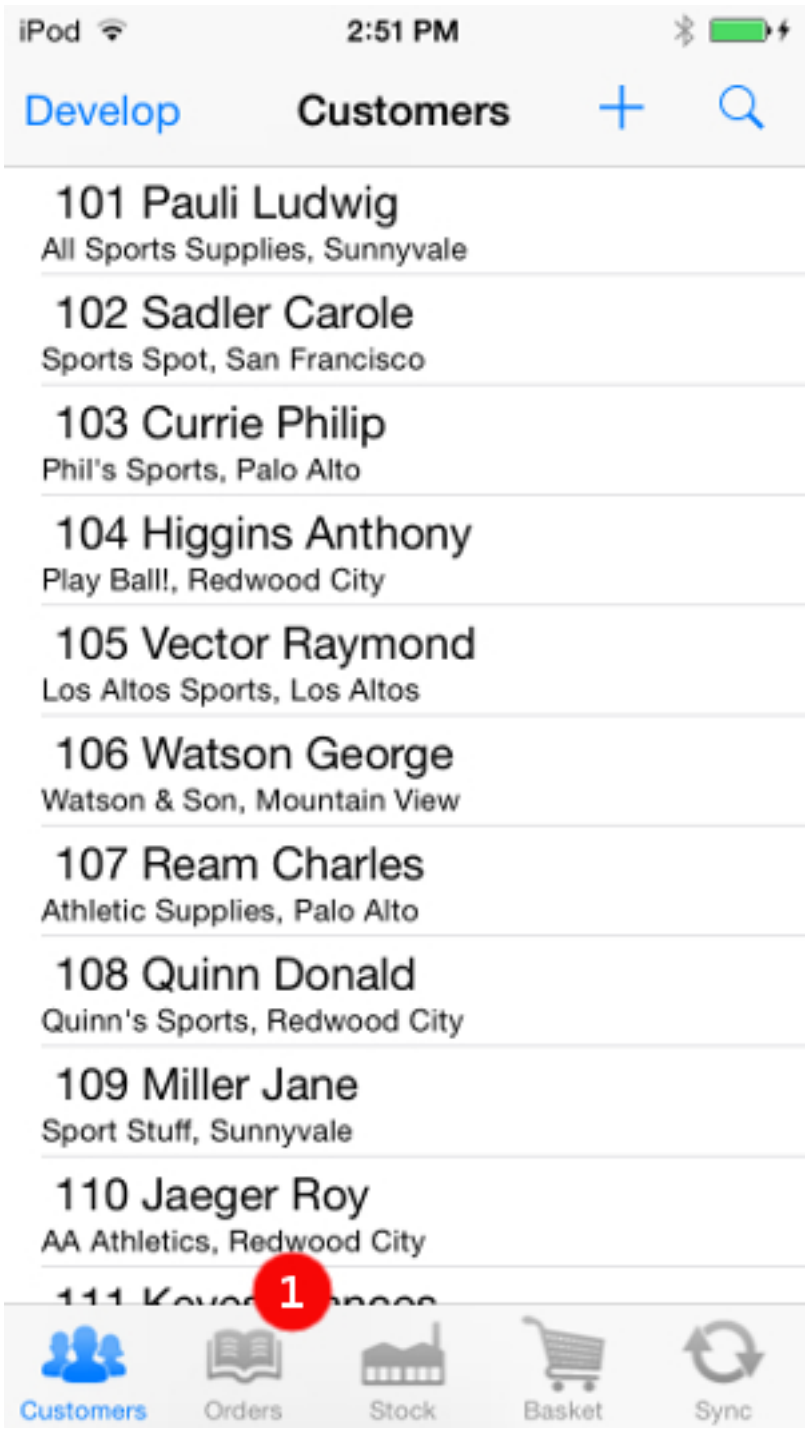
For many mobile applications, you will want to provide a view that allows you to show different forms and views that are active at the same time, to expose different functional areas for your application. This can be achieved by providing a top-level navigator with several views, controlled by parallel dialogs.

In order to implement a *top-level navigator*, create a window with the TYPE=NAVIGATOR attribute and without a form (i.e. using the x ROWS y COLUMNS clause). This window will only be used to display a set of actions views, to let the user switch between views. A view can be implemented as a split view by using a left and right typed window.

> **Important:** The navigator window must be the root window (after closing the default SCREEN window). If regular windows are created before the navigator window, these must be closed:

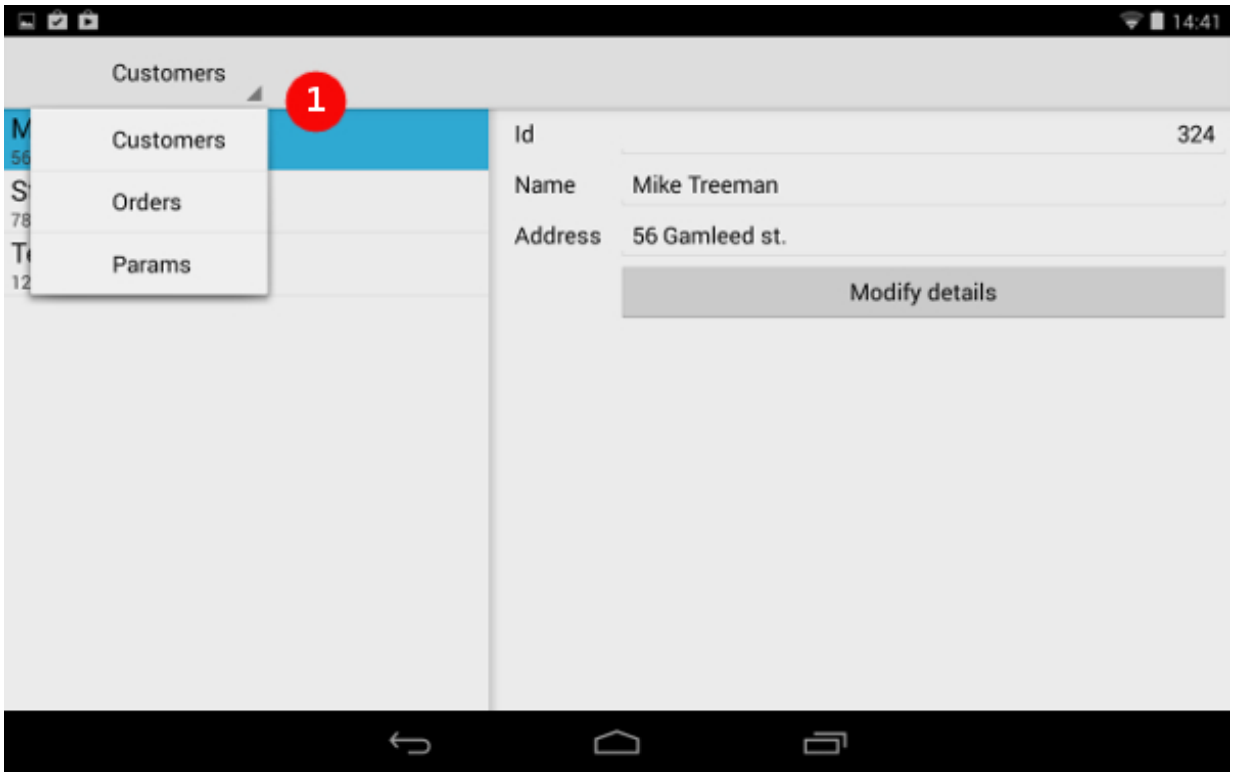
```
-- Case 1: Screen window is closed, navigator is the root window
CLOSE WINDOW screen
...
OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNNS ATTRIBUTES(TYPE=NAVIGATOR)

-- Case 2: Close regular windows created before the navigator window
OPEN WINDOW w1 WITH FORM "form1"
...
CLOSE WINDOW w1
...
OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNNS ATTRIBUTES(TYPE=NAVIGATOR)
```

- On iOS devices, the navigator window displays in a typical iOS tab bar at the bottom of the screen:



To customize the iOS application, define the color of the iOS tab bar the iosTabBarTintColor Window-class style attribute.

- On Android™ devices, the navigator window displays in the top of the screen, in the view control of the action bar (2):



The navigator window will be controlled by a dedicated parallel dialog implementing a `MENU` instruction, with the action handlers to select the related window, when the corresponding action is fired.

> **Important:** The name of the actions in the navigator menu must match the name of the corresponding window, which is typically, the left-typed window when using split views.

The next example implements:

- The `w_main` window, and its corresponding controller, the `d_navigator` parallel dialog.
- The `w_customers` window as a left-pane, with the `d_customers` parallel dialog.
- The `w_orders` window as a right-pane, with the `d_orders` parallel dialog.
- The navigator `MENU` dialog implemens the `w_customer` and `w_orders` action handlers.

```
...
OPEN WINDOW w_main WITH 10 ROWS, 80 COLUMNNS ATTRIBUTES(TYPE=NAVIGATOR)
START DIALOG d_navigator
OPEN WINDOW w_customers WITH FORM "customers" ATTRIBUTES(TYPE=LEFT)
START DIALOG d_customers
OPEN WINDOW w_orders WITH FORM "orders" ATTRIBUTES(TYPE=RIGHT)
START DIALOG d_orders
...
DIALOG d_navigator()
   MENU
       ON ACTION w_customers ATTRIBUTES(TEXT="Customers", IMAGE="smiley")
          CURRENT WINDOW IS w_customers
       ON ACTION w_orders ATTRIBUTES(TEXT="Orders")
          CURRENT WINDOW IS w_orders
   END MENU
END DIALOG
```

The functionality is the same on either mobile platforms: providing the user with the ability to navigate between multiple views efficiently. The rendering depends on the platform:

- On an iOS device, navigator window renders as a tab bar, displaying at the bottom of the screen.

- On an Android device, navigator window renders as a spinner, which is a drop-down menu in the action bar.

Rendering an HBox as a split view
Achieve a split view display with HBOX container and style attribute.

By defining a TABLE and a GRID container in a parent HBOX container, it is possible to get a splitview display by setting the splitViewRendering style attribute of the HBox class. When using this style attribute, the TABLE displays as a listview on the left of the screen, while the GRID displays as a form on the right.

First define a form with the HBOX container, TABLE and GRID. In the code example below, the HBOX container gets a STYLE attribute referencing a style named "splitview" in the .4st file:

```
LAYOUT
HBOX (STYLE="splitview")
TABLE
{
[c1  |c2          ]
[c1  |c2          ]
[c1  |c2          ]
[c1  |c2          ]
}
END
GRID
{
First name:   [f1                ]
Last name:    [f2                ]
...
}
END
END
...
```

The .4st file should look like follows:

```
<StyleList>
  <Style name="HBox.mystyle">
     <StyleAttribute name="splitViewRendering" value="yes" />
  </Style>
  ...
</StyleList>
```

The program must implement a dialog that handles both parts of the splitview. You typically implement a DISPLAY ARRAY to handle the TABLE, and refresh the right part of the screen contained in the GRID, with code in the BEFORE ROW control block:

```
DISPLAY ARRAY arr TO sr.*
    BEFORE ROW
      DISPLAY arr[arr_curr()].first_name TO f_first_name
      DISPLAY arr[arr_curr()].last_name TO f_last_name
END DISPLAY
```

Examples

Example applications using split views.

Example 1: Single split view application

This application uses a minimum amount of code to describe a typical implementation of parallel dialogs that result in a split view application, with a list in the left pane and the detail for the selected row in the right pane. It uses only one split view.

main.4gl

The code in the `MAIN` block creates the left pane/window and the right pane/window by specifiying the `TYPE` attribute in `OPEN WINDOW`.

The left window will display a form comprised of a table view of all records (`a1_list_view`), the other window contains a form with the detail view of a single record from the array (`a1_detail_view`)

The `START DIALOG` statements, along with the `WHILE fgl_eventLoop()` loop, creates the parallel dialog on which a split view depends.

```
DEFINE arr DYNAMIC ARRAY OF RECORD
            id INTEGER,
            name VARCHAR(15),
            date DATE,
            comment VARCHAR(30)
  END RECORD
DEFINE curr_pa SMALLINT

MAIN
  CLOSE WINDOW SCREEN
  CALL populate_array()

  OPEN WINDOW w_left WITH FORM "list_view"
       ATTRIBUTES(TYPE=LEFT)
  START DIALOG d_list_view

  OPEN WINDOW w_right WITH FORM "detail_view"
       ATTRIBUTES(TYPE=RIGHT)
  START DIALOG d_detail_view

  WHILE fgl_eventLoop()
  END WHILE

END MAIN

DIALOG d_list_view()
  DISPLAY ARRAY arr TO sr.*
                ATTRIBUTES(ACCESSORYTYPE=DISCLOSUREINDICATOR)
      BEFORE ROW -- in BEFORE ROW, we restart the details view
        CURRENT WINDOW IS w_right
        TERMINATE DIALOG d_detail_view
        LET curr_pa = arr_curr()
        DISPLAY BY NAME arr[curr_pa].*
        DISPLAY SFMT("tapped row %1",arr_curr()) TO info
        START DIALOG d_detail_view
        CURRENT WINDOW IS w_left
  END DISPLAY
END DIALOG

DIALOG d_detail_view()
  MENU
    ON ACTION an_action
      MESSAGE "The action an_action was selected!"
    ON ACTION details
      IF edit_details() THEN
```

```
         DISPLAY BY NAME arr[curr_pa].*
      END IF
  END MENU
END DIALOG

FUNCTION edit_details()
  -- A modal dialog disables all parallel dialogs
  OPEN WINDOW w_details WITH FORM "details"
       ATTRIBUTES(TYPE=POPUP, STYLE="popup")
  LET int_flag=FALSE
  INPUT BY NAME
        arr[curr_pa].name,
        arr[curr_pa].comment
      WITHOUT DEFAULTS
  CLOSE WINDOW w_details
  RETURN (int_flag==FALSE)
END FUNCTION

FUNCTION populate_array()
  DEFINE i INT
  FOR i=1 TO 40
    LET arr[i].id=i
    LET arr[i].name="item "||i
    LET arr[i].date=TODAY
    LET arr[i].comment="item-detail "||i
  END FOR
END FUNCTION
```

Left form definition file (list_view.per)

This form definition file provides the table, or list, of records in the array. Even though four table columns are defined, only two display.

```
LAYOUT (TEXT="Items")
TABLE
{
[c1    |c2             ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.id;
EDIT c1=FORMONLY.name;
PHANTOM FORMONLY.date;
EDIT c2=FORMONLY.comment;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

Right form definition file (detail_view.per)

This form definition file displays the details for a single record in the array.

```
LAYOUT (TEXT="Details")
GRID
{
Id      [f01              ]
Name    [f02              ]
Date    [f03              ]
Comment [f04              ]
Info    [f05              ]
```

```
        [b1_details        ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.id;
EDIT f02=FORMONLY.name, SCROLL;
EDIT f03=FORMONLY.date;
EDIT f04=FORMONLY.comment, SCROLL;
EDIT f05=FORMONLY.info;
BUTTON b1_details:details,TEXT="Modify details";
END
```

Detail form definition file (details.per)

This is a simple form containing a two fields that will be used in the program by the `edit_details()` function to modify item details.

```
LAYOUT (TEXT="Edit details")
GRID
{
Name:    [f01                  ]
Comment: [f02                  ]
         [                     ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.name, SCROLL;
TEXTEDIT f02=FORMONLY.comment, STRETCH=BOTH;
END
```

Example 2: Multiple split views with navigation bar
This example shows how to write an application that handles two split views, each having a left and right pane, with a top level navigation pane that allows the end user to easily switch between the two split views.

main.4gl

This module implements the window creation and the parallel dialogs to control their content.

The code in the `MAIN` block creates four windows:

- The main window is the navigation window/pane, defined by the `TYPE=NAVIGATION` attribute. Only the `d_navigator()` main dialog is started.
- Two other windows are created for the customer list and details, in the `customers()` function. This function is called when the main dialog starts. The function checks if the `w_customers` window exists and if needed, opens the splitview windows and starts the dialogs handling customer records. If windows already exists, it performs a `CURRENT WINDOW IS w_customers`, to select the customer pane.
- The second window showing orders and its corresponding dialog are created in the `orders()` function, using the same programming pattern as in the `customers()` function.
- When the user selects one of the main dialog actions, it calls either the `customers()`, the `orders()`, or the `params()` function, to show the corresponding pane.
- The configuration pane is handled in the `params()` function, with the corresponding `d_params_menu` dialog: When selected, the form is in read-only mode by default. The menu implements the "modify" action to edit the parameters. This action will create a modal dialog, that stops temporarly the parallel dialogs.

```
DEFINE c_arr DYNAMIC ARRAY OF RECORD
            id INTEGER,
```

```
            name VARCHAR(30),
            address VARCHAR(100)
  END RECORD,
  c_curr INTEGER

DEFINE o_arr DYNAMIC ARRAY OF RECORD
            id INTEGER,
            info VARCHAR(100),
            deliv DATE
  END RECORD

DEFINE params RECORD
            user_name VARCHAR(30),
            auto_sync CHAR(1)
  END RECORD

MAIN
  CLOSE WINDOW SCREEN
  OPEN WINDOW w_navigator WITH 10 ROWS, 80 COLUMNS
       ATTRIBUTES(TYPE=NAVIGATOR)
  START DIALOG d_navigator
  WHILE fgl_eventLoop()
  END WHILE
END MAIN

DIALOG d_navigator()
  MENU
      BEFORE MENU
         CALL customers()
      -- Note that action names must match the window names
      ON ACTION w_customers ATTRIBUTES(TEXT="Customers",IMAGE="customers")
         CALL customers()
      ON ACTION w_orders ATTRIBUTES(TEXT="Orders",IMAGE="orders")
         CALL orders()
      ON ACTION w_params ATTRIBUTES(TEXT="Params",IMAGE="sync")
         CALL params()
  END MENU
END DIALOG

FUNCTION params()
  IF ui.Window.forName("w_params") IS NULL THEN
     OPEN WINDOW w_params WITH FORM "parameters"
     LET params.user_name="Tom"
     LET params.auto_sync="Y"
     DISPLAY BY NAME params.*
     START DIALOG d_params_menu
  END IF
  CURRENT WINDOW IS w_params
END FUNCTION

DIALOG d_params_menu()
  MENU
    ON ACTION modify ATTRIBUTES(TEXT="Modify")
       CALL edit_params()
    ON ACTION options ATTRIBUTES(TEXT="Options")
       CALL options()
  END MENU
END DIALOG

FUNCTION edit_params() -- This is a modal dialog
  LET int_flag=FALSE
  INPUT BY NAME params.* ATTRIBUTES(WITHOUT DEFAULTS)
  IF NOT int_flag THEN
     -- CALL save_params()
```

```
  END IF
END FUNCTION

FUNCTION options()
  MENU "Options" ATTRIBUTES(STYLE="dialog")
    ON ACTION sync ATTRIBUTES(TEXT="Synchronize")
       --
    ON ACTION exit ATTRIBUTES(TEXT="Exit")
       EXIT PROGRAM
    ON ACTION cancel
       EXIT MENU
  END MENU
END FUNCTION

FUNCTION customers()
    IF ui.Window.forName("w_customers") IS NULL THEN
       CALL populate_customers()
       OPEN WINDOW w_customers WITH FORM "customer_list"
            ATTRIBUTES(TYPE=LEFT)
       START DIALOG d_customer_list
       OPEN WINDOW w_customer_detail WITH FORM "customer_detail"
            ATTRIBUTES(TYPE=RIGHT)
       START DIALOG d_customer_detail
    END IF
    CURRENT WINDOW IS w_customers
END FUNCTION

DIALOG d_customer_list()
  DISPLAY ARRAY c_arr TO c_sr.*
          ATTRIBUTES(ACCESSORYTYPE=DISCLOSUREINDICATOR)
      BEFORE ROW
        CURRENT WINDOW IS w_customer_detail
        TERMINATE DIALOG d_customer_detail
        LET c_curr = arr_curr()
        DISPLAY BY NAME c_arr[c_curr].*
        START DIALOG d_customer_detail
        CURRENT WINDOW IS w_customers
  END DISPLAY
END DIALOG

DIALOG d_customer_detail()
  MENU
    ON ACTION details
       LET int_flag=FALSE
       INPUT BY NAME c_arr[c_curr].name,
                     c_arr[c_curr].address
             WITHOUT DEFAULTS
       IF NOT int_flag THEN
          DISPLAY BY NAME c_arr[c_curr].*
       END IF
  END MENU
END DIALOG

FUNCTION populate_customers()
  LET c_arr[1].id = 324
  LET c_arr[1].name = "Mike Treeman"
  LET c_arr[1].address = "56 Gamleed st."
  LET c_arr[2].id = 8934
  LET c_arr[2].name = "Stepfan Plombier"
  LET c_arr[2].address = "78 Pokam st."
  LET c_arr[3].id = 451
  LET c_arr[3].name = "Ted Barber"
  LET c_arr[3].address = "1243b Western st."
END FUNCTION
```

```
FUNCTION orders()
    IF ui.Window.forName("w_orders") IS NULL THEN
       CALL populate_orders()
       OPEN WINDOW w_orders WITH FORM "order_list"
       START DIALOG d_order_list
    END IF
    CURRENT WINDOW IS w_orders
END FUNCTION

DIALOG d_order_list()
  DISPLAY ARRAY o_arr TO o_sr.*
  END DISPLAY
END DIALOG

FUNCTION populate_orders()
  LET o_arr[1].id = 43249
  LET o_arr[1].info = "Xmass gifts"
  LET o_arr[1].deliv = MDY(12,23,2011)
  LET o_arr[2].id = 33424
  LET o_arr[2].info = "Dressing items"
  LET o_arr[2].deliv = MDY(2,13,2012)
END FUNCTION
```

customer_list.per

This is the form defining the customer list, it is used for the left-pane of the customers split view.

```
LAYOUT (TEXT="Customers")
TABLE
{
[c1    |c2             ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.id;
EDIT c1=FORMONLY.name;
EDIT c2=FORMONLY.address;
END
INSTRUCTIONS
SCREEN RECORD c_sr(FORMONLY.*);
END
```

customer_detail.per

This is the form defining fields to show customer details, it is used for the right-pane of the customers split view.

```
LAYOUT (TEXT="Customer details")
GRID
{
Id      [f01              ]
Name    [f02              ]
Address [f03              ]
        [b1_details       ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.id;
EDIT f02=FORMONLY.name, SCROLL;
```

```
EDIT f03=FORMONLY.address, SCROLL;
BUTTON b1_details:details,TEXT="Modify details";
END
```

order_list.per

This is the form defining the order list, it is a single form (not a split view)

```
LAYOUT (TEXT="Orders")
TABLE
{
[c1    |c2             ]
}
END
END
ATTRIBUTES
PHANTOM FORMONLY.id;
EDIT c1=FORMONLY.info;
EDIT c2=FORMONLY.date;
END
INSTRUCTIONS
SCREEN RECORD o_sr(FORMONLY.*);
END
```

parameters.per

```
LAYOUT (TEXT="Settings")
GRID
{
User       [f01              ]
Auto sync  [f02              ]
}
END
END
ATTRIBUTES
EDIT f01=FORMONLY.user_name, SCROLL;
CHECKBOX f02=FORMONLY.auto_sync, NOT NULL,
     VALUECHECKED="Y", VALUEUNCHECKED="N";
END
```

Example 3: Split view using an HBox
This app uses a minimum amount of code to show a split view implementation using an hbox container.

Styles definition file (mystyles.4st)

For this example, we start with the style file. The style file specifies the `splitViewRendering` attribute for the `HBox` container when the style is set to `mysplitview`.

```
<?xml version="1.0" encoding="ANSI_X3.4-1968"?>
<StyleList>
  <Style name="HBox.mysplitview">
     <StyleAttribute name="splitViewRendering" value="yes" />
  </Style>
</StyleList>
```

Form definition file (splitview.per)

The form definition file defines a `HBOX` container using the `mysplitview` style. It contains a `TABLE` followed by a `GRID`. The table will become the left pane of the split view app, and the grid will become the right pane of the split view app.

```
LAYOUT
  HBOX (STYLE="mysplitview")
    TABLE
    {
    [c1            |c2                            ]
    [c1            |c2                            ]
    [c1            |c2                            ]
    [c1            |c2                            ]
    }
    END
    GRID
    {
    <GROUP g1                                       >
      Name:          [lb_name             :lb_id  ]
      E-mail:        [lb_email                    ]
      Address:       [lb_address                  ]
      City:          [lb_city                     ]
    <                                               >
    <GROUP g2                                       >
      Phone:         [lb_phone                    ]
      Mobile:        [lb_mobile                   ]
    <                                               >
    }
    END
  END
END

ATTRIBUTES

PHANTOM FORMONLY.id;
EDIT c1 = FORMONLY.name;
EDIT c2 = FORMONLY.address;
PHANTOM FORMONLY.city;
PHANTOM FORMONLY.phone;
PHANTOM FORMONLY.mobile;
PHANTOM FORMONLY.email;
GROUP g1: group1, TEXT="Contact";
EDIT lb_id = FORMONLY.cont_id;
EDIT lb_name = FORMONLY.cont_name;
EDIT lb_address = FORMONLY.cont_address;
EDIT lb_city = FORMONLY.cont_city;
GROUP g2: group2, TEXT="Numbers";
EDIT lb_phone = FORMONLY.cont_phone;
EDIT lb_mobile = FORMONLY.cont_mobile;
EDIT lb_email = FORMONLY.cont_email;
END

INSTRUCTIONS
SCREEN RECORD sr(id, name, address, city,
                 phone, mobile, email);
END
```

Application (main.4gl)

The application starts by loading the `splitview.4st` style file.

After populating the array with our sample data, the `splitview.per` form is loaded and displayed in the default `SCREEN` window.

Then, a `DISPLAY ARRAY` statement takes control, and fills the fields in the grid in the `BEFORE ROW` trigger, when a new row is selected by the user.

```
DEFINE carr DYNAMIC ARRAY OF RECORD
    cont_id INTEGER,
    cont_name VARCHAR(40),
    cont_address VARCHAR(100),
    cont_city VARCHAR(50),
    cont_phone VARCHAR(20),
    cont_mobile VARCHAR(20),
    cont_email VARCHAR(30)
END RECORD

MAIN
  CALL ui.Interface.loadStyles("splitview")
  CALL load_samples()
  OPEN FORM f FROM "splitview"
  DISPLAY FORM f
  DISPLAY ARRAY carr TO sr.* ATTRIBUTES(UNBUFFERED)
    BEFORE ROW
      DISPLAY BY NAME carr[arr_curr()].*
  END DISPLAY
END MAIN

FUNCTION load_samples()
  DEFINE i INTEGER
  LET i=0
  LET carr[i:=i+1].cont_id = 982
  LET carr[i].cont_name = "Mike Stanford"
  LET carr[i].cont_address = "5 Marbel St."
  LET carr[i].cont_city = "Balmberg"
  LET carr[i].cont_phone = "8723847234"
  LET carr[i].cont_mobile= "8732487833"
  LET carr[i].cont_email = "mikest@xyz.com"
  LET carr[i:=i+1].cont_id = 8234
  LET carr[i].cont_name = "Phil Karlmon"
  LET carr[i].cont_address = "341 Merlo Bld"
  LET carr[i].cont_city = "Clerckmont"
  LET carr[i].cont_phone = "9823498234"
  LET carr[i].cont_mobile= "9999991213"
  LET carr[i].cont_email = "pkarl@yoioyoio.com"
  LET carr[i:=i+1].cont_id = 1045
  LET carr[i].cont_name = "Clark Gambello"
  LET carr[i].cont_address = "35 Straw St."
  LET carr[i].cont_city = "Bringstone"
  LET carr[i].cont_phone = "9823498234"
  LET carr[i].cont_mobile= "8234981111"
  LET carr[i].cont_email = "cgamb@youhoowha.com"
END FUNCTION
```

Drag & drop

Explains programming techniques for the drag & drop feature.

- Understanding drag & drop on page 1412
- Syntax of drag & drop interaction blocks on page 1412
- Default drag & drop operation on page 1413
- Control block execution order on page 1413
- Handle drag & drop data with MIME types on page 1413

- Examples on page 1415
  - Example 1: Two lists side-by-side with drag & drop on page 1415

Understanding drag & drop

Drag & drop is a well know feature of graphical applications, allowing the end user to use the mouse to drag an element of a window to another window in the same program or into an external application. The front-end platform/device must support this feature.

> **Important:** This feature is not supported on mobile platforms.

Drag & drop can be implemented in regular tables and tree-views controlled by a singular `DISPLAY ARRAY` or a `DISPLAY ARRAY` sub-dialog within a `DIALOG` instruction. Drag & drop is not supported in other dialog contexts, such as a singular `INPUT`, `INPUT ARRAY` or `CONSTRUCT`.

With drag & drop, end users can:

- Move drag-able objects between lists and tree-views in the same Genero form or program.
- Move drag-able objects between lists and tree-views in different Genero forms and programs.
- Move drag-able objects between other desktop applications and tables / tree-views in Genero programs.

Drag & drop control is implemented in a `DISPLAY ARRAY` with specific interaction blocks, to handle the events related to the drag and drop operation. These specific blocks will be triggered when drag and drop events arrive from the front-end.

- ON DRAG_START
- ON DRAG_FINISHED
- ON DRAG_ENTER
- ON DRAG_OVER
- ON DROP

Each of these interaction blocks takes a `ui.DragDrop` object as a parameter. A reference variable to that object must be declared before the dialog. In the interaction block, the `ui.DragDrop` object can be used to configure the drag & drop action to take. For example, a "drag enter" event can be refused.

The `ON DRAG_START` and `ON DRAG_FINISHED` triggers apply to the source of the drag & drop operation; the dialog where the object was dragged. The other triggers provide notification to the drop target dialog, used to inform the program when the different drop events occur and to let the target accept or reject the drop action.

This example illustrates the use of a drag & drop interaction block with the `ui.DragDrop` control object:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
    ON DRAG_ENTER(dnd)
      IF ok_to_drop THEN
        CALL dnd.setOperation("move")
      ELSE
        CALL dnd.setOperation(NULL)
      END IF
...
END DISPLAY
```

Syntax of drag & drop interaction blocks

The `ON DRAG*` / `ON DROP` interaction blocks implement drag & drop operations.

```
{ ON DRAG_START ( dnd-object )
| ON DRAG_FINISHED ( dnd-object )
```

```
| ON DRAG_ENTER( dnd-object )
| ON DRAG_OVER ( dnd-object )
| ON DROP ( dnd-object ) }
    dialog-statement
    [...]
```

1. *dnd-object* is a variable referencing an object of the class `ui.DragDrop`.

Default drag & drop operation

By default, all `DISPLAY ARRAY` dialogs implement a default drag operation that copies all selected rows to the drag & drop buffer as a tab-separated list of values.

The user code equivalent to the default drag & drop operation would look like this:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
  ON DRAG_START(dnd)
    CALL dnd.setOperation("copy")
    CALL dnd.setMimeType("text/plain")
    CALL dnd.setBuffer(DIALOG.selectionToString("sr"))
...
END DISPLAY
```

Control block execution order

The table below shows the order in which the runtime system executes the control blocks related to drag & drop events:

Table 293: Control block execution order for drag & drop events

Context / User action	Control Block execution order
The user starts to drag an object from the source dialog.	**1.** `ON DRAG_START` (in source dialog)
The mouse cursor enters the drop target dialog.	**1.** `ON DRAG_ENTER` (in target dialog)
After entering the target dialog, the mouse cursor moves from row to row, or user chooses to change the drop operation (move or copy).	**1.** `ON DRAG_OVER` (in target dialog)
The user releases the mouse button over the target dialog.	**1.** `ON DROP` (in target dialog) **2.** `ON DRAG_FINISHED` (in source dialog)

Handle drag & drop data with MIME types

If a drag & drop is intended to work only in the same application, data can be passed with variables in the context of the current program. For example, in a program using two tables where the user can drag & drop elements between the two lists, identify the selected rows and update the program arrays accordingly. When drag & drop is limited to the current application, avoid the drop outside the current application.

When a drag & drop operation comes from (or goes to) external applications, data can be of various types/formats: plain text, formatted text, documents, images, sounds, videos, and so on. In order to handle the drag & drop data, you must identify the type of data held in the drag & drop buffer. The type of data in the buffer is identified by the MIME type (Multiple Internet Mail Extensions). MIME types are a widely used internet standard specification, first introduced to identify the content of e-mail attachments.

Only text data can be passed with drag & drop; binary data is not supported. However, you can pass files by using the `fgl_getfile()` file transfer function, and identify the file with a URI (text-uri-list MIME type). For a working example, see the demos in `FGLDIR/demo/DragAndDrop`.

Example of MIME types:

- text/plain
- text/uri-list
- text/x-vcard

You can also define your own MIME type, as long as it does not conflict with existing standard MIME types. For example:

- text/my-remote-file
- text/my-customer-record

If you do not specify a MIME type when the drag starts, the type defaults to text/plain, and the dialog will by default copy the data from selected rows into the drag & drop buffer. To prevent drag & drop to external applications, you must pass an application-specific MIME type to the `ui.DragDrop.setMimeType()` method, to be sure that other applications do not recognize the MIME type and will deny the drop.

Preparing the dragged object for external targets

If the program implements drag & drop of objects that can be dropped to external programs, you must specify the MIME type of the object and copy the data to the drag & drop buffer, so that the external application can identify the data format and receive it.

In the `ON DRAG_START` block, you must call the `ui.DragDrop.setMimeType()` method to define the MIME type of the object, and copy the text data into the buffer with the `ui.DragDrop.setBuffer()` method.

This example shows a `DISPLAY ARRAY` dialog preparing the drag & drop buffer to export VCard data from a dragged row:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
  ON DRAG_START(dnd)
    -- Define the MIME type and copy text data to DnD buffer
    CALL dnd.setMimeType("text/x-vcard")
    CALL dnd.setBuffer( buildVCardData( arr[arr_curr()].cid ) )
    CALL dnd.setOperation("copy")
...
END DISPLAY
```

Receiving the dragged object from external sources

This describes how to handle the drop action when the target dialog receives an object dragged from an external source, by identifying the MIME type of the object.

In the `ON DRAG_ENTER` block, you must call the `ui.DragDrop.selectMimeType()` method to check that data is available in a format identified by the MIME type, passed as a parameter. If the type of data is available in the buffer, the method returns `TRUE`. Later, when the dragged object is dropped (`ON DROP`), you can get the previously selected MIME type with `ui.DragDrop.getSelectedMimeType()` before calling `ui.DragDrop.getBuffer()` to retrieve the actual data.

The next example shows the usage of those methods: In `ON DRAG_ENTER`, the program checks available MIME types, and denies the drop operation if the buffer does not hold any of the MIME types that can be treated by the program. In `ON DROP`, the program calls `getSelectMimeType()` to check what MIME

type was selected, retrieves the data with `getBuffer()` , then inserts a new row and puts the data in dedicated fields according to the MIME type:

```
DEFINE dnd ui.DragDrop
...
DISPLAY ARRAY arr TO sr.* ...
...
  ON DRAG_ENTER(dnd)
    -- Set operation to NULL if unexpected MIME type found
    CASE
    WHEN dnd.selectMimeType("text/plain")
    WHEN dnd.selectMimeType("text/uri-list")
    OTHERWISE
      CALL dnd.setOperation(NULL)
    END CASE
...
  ON DROP(dnd)
    -- Select MIME type and get data from buffer
    LET row = dnd.getLocationRow()
    CALL DIALOG.insertRow("sr", row)
    IF dnd.getSelectMimeType() == "text/plain" THEN
      LET arr[row].text_data = dnd.getBuffer()
    END IF
...
END DISPLAY
```

Examples

See demo directory for more Drag & Drop examples.

Example 1: Two lists side-by-side with drag & drop

```
MAIN
    DEFINE drag_index, drop_index, i INT
    DEFINE drag_source, drag_value SRING
    DEFINE arr_left, arr_right DYNAMIC ARRAY OF STRING
    DEFINE dnd ui.DragDrop
    CONSTANT S_LEFT="sr_left"
    CONSTANT S_RIGHT="sr_right"

    OPEN FORM f FROM "dnd"
    DISPLAY FORM f

    FOR i = 1 TO 10
       LET arr_left[i] = "left " || i
       LET arr_right[i] = "right" || i
    END FOR

    INITIALIZE drag_index TO NULL

    DIALOG ATTRIBUTES(UNBUFFERED)

        DISPLAY ARRAY arr_left TO sr_left.*
        ON DRAG_START(dnd)
            LET drag_source = S_LEFT
            LET drag_index = arr_curr()
            LET drag_value = arr_left[drag_index]
        ON DRAG_FINISHED(dnd)
            INITIALIZE drag_source TO NULL
        ON DRAG_ENTER(dnd)
            IF drag_source IS NULL THEN
                CALL dnd.setOperation(NULL)
            END IF
```

```
        ON DROP(dnd)
            IF drag_source == S_LEFT THEN
                CALL dnd.dropInternal()
            ELSE
                LET drop_index = dnd.getLocationRow()
                CALL DIALOG.insertRow(S_LEFT, drop_index)
                CALL DIALOG.setCurrentRow(S_LEFT, drop_index)
                LET arr_left[drop_index] = drag_value
                CALL DIALOG.deleteRow(S_RIGHT, drag_index)
            END IF
        END DISPLAY

        DISPLAY ARRAY arr_right TO sr_right.*
        ON DRAG_START(dnd)
            LET drag_source = S_RIGHT
            LET drag_index = arr_curr()
            LET drag_value = arr_right[drag_index]
        ON DRAG_FINISHED(dnd)
            INITIALIZE drag_source TO NULL
        ON DRAG_ENTER(dnd)
            IF drag_source IS NULL THEN
                CALL dnd.setOperation(NULL)
            END IF
        ON DROP(dnd)
            IF drag_source == S_RIGHT THEN
                CALL dnd.dropInternal()
            ELSE
                LET drop_index = dnd.getLocationRow()
                CALL DIALOG.insertRow(S_RIGHT, drop_index)
                CALL DIALOG.setCurrentRow(S_RIGHT, drop_index)
                LET arr_right[drop_index] = drag_value
                CALL DIALOG.deleteRow(S_LEFT, drag_index)
            END IF
        END DISPLAY

    ON ACTION cancel
        EXIT DIALOG

    END DIALOG
END MAIN
```

Web components

This section describes how to use web components in your application.

- Understanding web components on page 1417
- WEBCOMPONENT item type on page 900
- Controlling the web component layout on page 1418
- Using a URL-based web component on page 1419
  - Defining a URL-based web component in forms on page 1420
  - Specifying the URL source of a web component on page 1420
  - Controlling the URL web component in programs on page 1421
- Using a gICAPI web component on page 1422
  - HTML document and JavaScript for the gICAPI object on page 1422
  - The gICAPI web component interface script on page 1423
  - Deploying the gICAPI web component files on page 1426
  - Defining a gICAPI web component in forms on page 1428
  - Controlling the gICAPI web component in programs on page 1429
  - Using image resources with the gICAPI web component on page 1430

- Examples on page 1432
  - Example 1: URL-based web component using Google maps on page 1432
  - Example 2: Calling a JavaScript function of a gICAPI web component on page 1432
  - Example 3: Implementing Google+ authentication with a URL-based web component on page 1434
  - Example 4: Color picker gICAPI web component on page 1437

Understanding web components

External graphical components can be integrated into forms by using the `WEBCOMPONENT` form item type.

A `WEBCOMPONENT` form field is a form element that defines an area in the form layout to hold an external component, typically not available as a native widget on the front-end platform.

Web components are designed for a specific need, and usually have advanced and powerful features which can bring added value to your applications. For example, you can find chart and graph widgets, calendar widgets, drawing widgets, and more. Such specialized widgets are not part of the standard GUI toolkits used by Genero front-ends. They need to be integrated as external components.

> **Important:** Depending on the type of front-end, the web components can have limitations: When using native front-ends (GDC, GMA, GMI), the web components are implemented with a "webview" widget, which is not a full-featured web browser.
>
> The main web component limitations on native front-ends are:
>
> - lack of plugin support,
> - less accurate javascript engine,
> - lack of advanced html+css features.

Some web components are free, and some are licensed, so you should take the cost into account before integrating a new web component in your application.

Web components can be implemented with two different techniques:

1. Using an URL specification, by setting the URL as value of the `WEBCOMPONENT` field at runtime. This is the easiest way to implement a web component. The widget is controlled with URL values by the program, but requires some additional coding to handle URLs, instead of flat field values.
2. Using an gICAPI object (based on JavaScript™), by defining the `COMPONENTTYPE` attribute in the form file. This kind of web component requires some JavaScript coding, to write a form field "plugin", which is usable in a normal dialog instruction, that behaves as all the other widgets in terms of value setting/getting.

The content and/or behavior of a web component can be controlled in the program code by using the field value. To detect events inside the web component, the program dialogs must implement an `ON CHANGE` control block, that will be fired immediately after a user action on the web component.

WEBCOMPONENT item type

Defines a specialized form item that holds an external component.

WEBCOMPONENT item basics

The `WEBCOMPONENT` form item defines a form field that will hold an external component, implemented with a front-end plug-in mechanism.

This topic describes the `WEBCOMPONENT` item type in form definition files, a complete section is dedicated to web component programming.

Defining a WEBCOMPONENT

The `COMPONENTTYPE` attribute identifies gICAPI external objects to be used for the field. The `PROPERTIES` attribute is typically used to define attributes that are specific to a given gICAPI-based web component. For example, a chart component might have properties to define x-axis and y-axis labels. For more details, see Using a gICAPI web component on page 1422.

If the COMPONENTTYPE attribute is not used, the web component will be a URL-based web component. For more details, see Using a URL-based web component on page 1419.

Some front-ends support different presentation and behavior options, which can be controlled by a STYLE attribute. For more details, see Common style attributes on page 818.

Where to use a WEBCOMPONENT

A WEBCOMPONENT form item can be defined in two different ways:

1. With an item tag and a WEBCOMPONENT item definition on page 949 in a grid-layout container (GRID, SCROLLGRID and TABLE).
2. As a WEBCOMPONENT stack item on page 927 in a STACK container.

Defining the widget size

The size of a WEBCOMPONENT widget can be controlled in grid-based or stack-based layout, according to several attributes such as SIZEPOLICY and STRETCH.

For more details about image sizing, see Controlling the web component layout on page 1418.

Controlling the web component layout

Web component sizing basics

Web components are usually complex widgets displaying detailed information, such as charts, graphs, or calendars. Such widgets are generally resizeable. Therefore, the WEBCOMPONENT form item must be large and stretchable.

Viewport zooming on mobile devices

In order to avoid automatic viewport zooming with mobile applications, consider adding a meta tag with name='viewport' in the HTML file of your gICAPI-based web components, with initial and maximal scale attributes set to 1:

```
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
```

> **Note:** Don't use such responsive meta tag, if your web component isn't specifically designed to be responsive.

Web component size in grid-based layout

In a grid-bases layout, the item tag of the WEBCOMPONENT defines the default dimensions of the web component area:

```
LAYOUT
GRID
{
<GROUP g1                   >
[f1            ][f2         ]
[f3                         ]
 ...
<                           >
[f5                         ]
[                           ]
[                           ]
[                           ]
}
END
```

In the ATTRIBUTES section, use the SIZEPOLICY, SCROLLBARS and STRETCH attributes, to define the sizing policy of a web component field:

```
WEBCOMPONENT f5 = FORMONLY.mymap,
   SIZEPOLICY = FIXED,
   STRETCH = BOTH;
```

By default, the WEBCOMPONENT widget gets the size of the form item (like SIZEPOLICY=FIXED). When SIZEPOLICY=INITIAL, the web component is scaled to the right size after the first webpage is loaded and stays at that size. When SIZEPOLICY=DYNAMIC, the web component is resized after each load of a new webpage so that no scrollbars should appear.

Web component size layout in stack-based layout

In a stack-based layout, a WEBCOMPONENT item is defined with other items in a logical presentation order, without any size information:

```
LAYOUT
 STACK
  GROUP (TEXT="Chart example")
    COMBOBOX FORMONLY.chart_type, NOT NULL,
             INITIALIZER=chart_type_init;
    WEBCOMPONENT FORMONLY.chart,
                 COMPONENTTYPE = "chartjs",
                 STYLE="regular";
  END
 END
END
```

By default, the WEBCOMPONENT widget size will adapt to the content of the web component: It will stretch vertically to the appropriate size, in order to show the complete web component content.

To limit the size of the WEBCOMPONENT widget, you can use the HEIGHT attribute in the form definition:

```
WEBCOMPONENT FORMONLY.chart,
             HEIGHT = 5,        -- 5 lines
             ...
```

> **Note:** If the HEIGHT attribute of the web component is defined in the form file, it fixes the widget height, which may result in vertical scrollbars inside the widget. This is like using SIZEPOLICY=FIXED for a web component in a grid-based layout.

If the HEIGHT attribute is not specified in the .per file, the front-end will take the height attribute of the HTML elements of the web component HTML file into account, for example when using a <canvas /> element:

```
<body>
  <canvas id="myChart" height="100px" />
```

Using a URL-based web component

This section describes how to add a *URL-based web component* to your application.

To implement an URL-based web component:

URL-based web components are hosted on a third-party server and provide a specific service, such as a geographical location on a map. Make sure that the service is available.

1. Identify the URL of the hosted web component you want to use.
2. In the form file, define a WEBCOMPONENT field, without a COMPONENTTYPE attribute.

See Defining a URL-based web component in forms on page 1420 for more details.

3. In the program, set the URL of the hosted web component in the form field value.

   See Specifying the URL source of a web component on page 1420 for more details.

4. In the program, detect user interactions with an `ON CHANGE` control block, and control the URL-based web component with dedicated front calls.

   See Controlling the URL web component in programs on page 1421 for more details.

Defining a URL-based web component in forms

Adding a WEBCOMPONENT to the form file

To define a URL-based web component field, add a form field with the `WEBCOMPONENT` item type, without the `COMPONENTTYPE` attribute:

```
WEBCOMPONENT f001 = FORMONLY.mymap;
```

A web component field is typically defined with the `FORMONLY` prefix, as the data for the field is rarely stored in a database column.

The field type (and its corresponding program variable) must be a character string type. Consider using the `STRING` type to avoid any size limitation for the URL specification.

Sizing policy for web component fields

Web components are usually complex widgets displaying detailed information, such as charts, graphs, or calendars, which are generally resizeable. Use the appropriate form item attributes to get the expected layout and behavior. For more details, see Controlling the web component layout on page 1418.

Example

```
LAYOUT
GRID
{
[wc                        ]
[                          ]
[                          ]
[                          ]
[                          ]
}
END
END
ATTRIBUTES
WEBCOMPONENT wc = FORMONLY.mychart,
         STRETCH = BOTH;
END
```

Specifying the URL source of a web component

The content of URL-based web components is defined by the form field value. It can only be set by program.

Setting the initial URL

When the current form defines a `WEBCOMPONENT` form item without the `COMPONENTTYPE` attribute, it is a URL-based web component. The program can set the URL dynamically in field value:

```
DISPLAY "wc-URL" TO wc-field
```

or with:

```
DEFINE wc_field STRING
LET wc_field = "wc-URL"
DISPLAY BY NAME wc_field
```

or by using the variable in an `INPUT` dialog with the `UNBUFFERED` option:

```
DEFINE rec RECORD
   name STRING,
   mymap STRING
 END RECORD
...
LET rec.mymap = "http://www.openstreetmap.org"
INPUT BY NAME rec.* WITHOUT DEFAULTS
      ATTRIBUTES(UNBUFFERED)
   ...
```

Once the URL of the web component is defined, the initial URL content is shown by the front-end, and the end user can interact with it.

Changing the URL

During program execution, you can assigning another URL to the web component field value. The content will be updated to show the new URL.

This example implements a `MENU` dialog with actions that set different URLs to the web component field, changing the content based on the selected action:

```
MENU "test"
   ON ACTION map_1
      DISPLAY "http://www.openstreetmap.org" TO wc_field
   ON ACTION map_2
      DISPLAY "http://www.wikimapia.org" TO wc_field
   ON ACTION map_3
      DISPLAY "http://maps.google.com" TO wc_field
END MENU
```

Controlling the URL web component in programs

URL-based web components can be controlled with the field value and with front calls

Detecting user interaction in a web component with ON CHANGE

The content of an URL-based web component is defined by the field value.

When the end user interacts with the content, and if the remote service points to a different URL, the field value changes.

The URL change can be detected by implementing an `ON CHANGE` control block for the web component field. The trigger will be fired immediately when the URL changes:

```
DEFINE rec RECORD
           num INTEGER,
           name STRING,
           map STRING
       END RECORD
...
INPUT BY NAME rec.* WITHOUT DEFAULTS
    ATTRIBUTES(UNBUFFERED)
    ...
    ON CHANGE map
```

```
        CALL map_changed(rec.map)
    ...
```

Controlling URL-based web components with front calls

The web component can be manipulated with specific front calls. The web component-specific front calls are provided in the "`webcomponent`" front call module.

The `call` front call that can be used for general purposes. It takes as parameters the name of the form field, a JavaScript function to call, and optional parameters as required. The JavaScript function must be implemented in the HTML content pointed by the URL of the web component field. The front call returns the result of the JavaScript function.

```
DEFINE title STRING
CALL ui.Interface.frontCall("webcomponent", "call",
     ["formonly.url_field", "eval", "document.title"],
     [title] )
```

The `getTitle` function is another useful `webcomponent` front call that can get the title of the HTML document of the web component:

```
DEFINE info STRING
CALL ui.Interface.frontCall("webcomponent", "getTitle",
     ["formonly.url_field"], [info] )
```

Some providers return key information in the title of the HTML document.

Using a gICAPI web component

This section describes how to add a *gICAPI-based web component* to your application.

To implement a gICAPI-based web compoent:

1. Identify the web component you want to use and get the source code (HTML, JavaScript™, CSS).
2. Implement the gICAPI interface script for the web component.

   See The gICAPI web component interface script on page 1423 for more details about the gICAPI interface script implementation.
3. Define the location where the front end can find the gICAPI interface files. This depends on the front end technology used by your application.

   See Deploying the gICAPI web component files on page 1426 for more details.
4. Define a `WEBCOMPONENT` field in the form file. Use the `COMPONENTTYPE` attribute to define the root HTML file name describing the gICAPI web component.

   See Defining a gICAPI web component in forms on page 1428 for more details.
5. Use the web component in the dialog of the program.

   See Controlling the gICAPI web component in programs on page 1429 for more details.
6. If image resources are required by your web component, you must provide them as part of the gICAPI web component assets, or provide them from the program with a specific API.

   See Using image resources with the gICAPI web component on page 1430 for more details.

HTML document and JavaScript for the gICAPI object

A gICAPI web component is identified by an HTML document containing the JavaScript interface (or a reference to the .js file).

The HTML document is defined by the `COMPONENTTYPE` attribute of the `WEBCOMPONENT` form field. The name specified in this attribute will be used to identify the HTML file:

```
WEBCOMPONENT wc = FORMOMLY.chart,
```

```
    COMPONENTTYPE = "mychart"; -- Identifies "mychart.html"
```

The HTML document must reference (or contain) the JavaScript implementing the gICAPI interface:

```
<!DOCTYPE html>
<html>
<head>
  <title>The title</title>
  <script language="JavaScript" type="text/javascript" src="wc_echo.js"></
script>
</head>
<body>
<div style="background-color:green;width:3000px;height:3000px;" >
here
</div>
</body>
</html>
```

The gICAPI web component interface script

The gICAPI web components are controlled on the front end through a `gICAPI` interface object, defined in an JavaScript™ script.

gICAPI interface basics

The goal of the gICAPI interface is to manage communication between the program and the web component with a basic API, to handle the interaction events, the focus and the value of the web component field.

The interface script is written in JavaScript™ and bound to the `WEBCOMPOMENT` form field by using an HTML document as container.

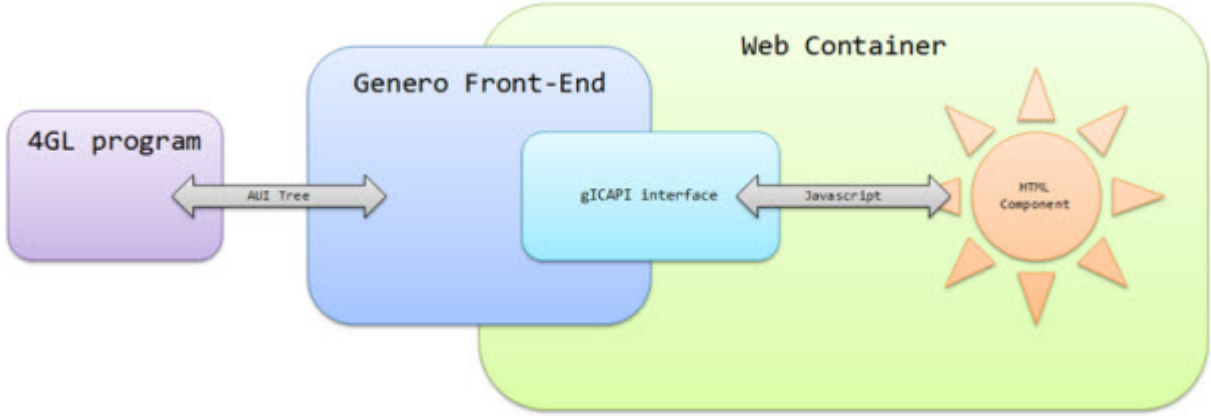


Figure 100: Web Component communication management

The gICAPI web component API relies on a published global JavaScript™ object named `gICAPI`.

Predefined function for gICAPI interface initialization

The `onICHostReady()` global function must be implemented, to execute code after the `gICAPI` interface has been initialized.

> **Note:** The `gICAPI` object is ready in the context of `onICHostReady()`.

The programming interface of the gICAPI class is identified by a version number, to make sure that the user code corresponds to the current gICAPI implementation. Verify that t runtime version number matches the gICAPI version used during development, by checking the value passed as parameter to `onICHostReady()`.

> **Note:** The `onICHostReady()` function is mandatory to check the gICAPI interface version, and to implement the assignment of `gICAPI.on*` callback functions as described later in this topic.

Table 294: Function to handle the gICAPI interface

Method	Description
`onICHostReady( version String )`	Called when the gICAPI web component interface is ready. The version passed in the parameter allows you to check that your component is compatible with the API, and initialization code can be execute in this function.

In order to check the interface version, implement the `onICHostReady()` function as follows:

```
onICHostReady = function(version) {
    if ( version != 1.0 )
        alert('Invalid API version');
    // More initialization code...
}
```

gICAPI callbacks to handle events coming from the program

The `on*` functions must be implemented as callbacks (functions assigned to `gICAPI` object members), in order to detect changes comming from the program (such as a web component field value modification with `onData()`).

> **Important:** The `gICAPI` object must be instanciated, before defining and assigning these methods. The `gICAPI` object is created and initialized by the web component framework before calling the `onICHostReady()` global function. Therefore, `on*` callback methods are typically defined and assigned to the `gICAPI` object, inside the body of the `onICHostReady()` function.

```
onICHostReady = function(version) {

    if ( version != 1.0 )
        alert('Invalid API version');

    current_color = "#000000";

    gICAPI.onProperty = function(properties) {
        var ps = eval('(' + properties + ')');
        document.getElementById("title").innerHTML = ps.title;
    }

    ...
```

Table 295: Object methods of gICAPI (program to component)

Method	Description
`onData( data String )`	Called when the program value of the form field is updated by the runtime system (for example, when doing a `DISPLAY` *`variable`* `TO` *`wc_field`*) or when the VM returns the validated value, after a `gICAPI.SetData()` was performed by the web component. The VM value is provided in the `data` parameter. When the `onData()` function is fired, you will typicaly assign the data value to the web component, or check that the VM has validated your `SetData()` request, when the value sent and the value returned by the VM do match.

Method	Description
	The data must be a string, it is not a scalar value, it is typically serialized as a JSON string. **Note:** Use util.JSON classes to serialize / de-serialize structured data (i.e. RECORDs or ARRAYs)
`onFocus( polarity Boolean )`	Called when the runtime system / program changes the focus (for example, with a `NEXT FIELD` instruction, or when the user tabs through the form fields). If the web component gains the focus, `polarity` is set to `true`. If the web component loses the focus, `polarity` is set to `false`. **Note:** This function is also called after a `gICAPI.setFocus()`, if the runtime system has accepted to set the focus to the web component field.
`onProperty( properties String )`	Called when one of the properties of the component is set at form creation time, or when a property is changed dynamically once the form and web component are loaded. The format used to pass the property set is JSON. **Note:** Each time this function is called, all properties are provided in the parameter.

gICAPI functions to send events to the program

Table 296: Object methods of gICAPI (component to program)

Method	Description
`Action( action String )`	Triggers an action event, which will execute the corresponding `ON ACTION` code. If the named action is not available (not active or does not exist), this function has no effect.
`SetData( data String )`	Registers data to be sent to the program, to set the form field value. The data must be a string, it is not a scalar value, it is typically serialized as a JSON string. **Note:** The gICAPI-based web component field must be the current field (i.e. the web component field must have the focus), otherwise `SetData()` will be ignored. The value change is transmitted to the runtime system when the web component field loses the focus. If you want to transmit the value change the runtime system immediately, `gICAPI.SetData()` must be used in conjunction with `gICAPI.Action()`, in order to fire an action. After a `SetData()` the value is sent to the VM which can accept or reject the field value change. In order to detect that the VM has accepted the value, the `onData()` function will be called with the same value as the value sent by `SetData()`. The web component then receives an indication that the VM has accepted the value change. **Note:** Data is transmitted as plain text; Sending a large amount of data is not recommended.
`SetFocus()`	Generates a focus change request. The tabbing order and focus management is controlled by the runtime system: If the focus change request succeeds, `gICAPI.onFocus()` will be called with the `true` parameter.

Method	Description
	The focus request may fail when: • the current field does not satisfy field constraints (`VERIFY`, data type conversion, and so on.) • due to program logic (`AFTER FIELD ...`, `NEXT FIELD`).

Deploying the gICAPI web component files
Deploy web component files to the front-end platform before using gICAPI web components.

Deploying the HTML document and the JavaScript gICAPI interface

The gICAPI web component files (main HTML file, additional JavaScript files and other potential assets) must be available on the platform where the front-end executes. According to your configuration, Genero supports several solutions to provide the gICAPI web component files from a single location. In a distributed configuration with many individual front-end nodes, consider centralizing the gICAPI files on a server, instead of copying the gICAPI web component files manually to each front-end device.

> **Important:** If the main gICAPI HTML document references external JavaScript files, put these files in the same directory as the HTML file referencing them.

Deploying gICAPI web component files with the GAS (using any front-end)

When using the Genero Application Server, the gICAPI web component files must be deployed as part of the application program files.

The `.xcf` configuration file of your application must define the base path to search for HTML web component files. This base path is defined by the `WEB_COMPONENT_DIRECTORY` entry of the `EXECUTION` element:

```
<APPLICATION ...
  <EXECUTION>
     ...
     <WEB_COMPONENT_DIRECTORY>$(application.path)/webcomponents</
WEB_COMPONENT_DIRECTORY>
     ...
```

The HTML document must be located in a sub-directory below the base path, using the same name as defined by the `COMPONENTTYPE` attribute. As result, the complete path to the HTML document will be:

`base-path/component-type/component-type``.html`

> **Note:** The above example uses the default value of the `WEB_COMPONENT_DIRECTORY` parameter. If you locate your gICAPI web component files under *`appdir`*`/webcomponents/`*`component-type`*, you do not need to set this element in the `.xcf` file.

For example, if the form file defines the `COMPONENTTYPE` attribute as follows:

```
WEBCOMPONENT wc = FORMOMLY.mychart,
   COMPONENTTYPE = "3DChart";
```

If `WEB_COMPOMENT_DIRECTORY` is defined as "`$(application.path)/webcomponents`", and `application.path` is "`/opt/var/gas/appdata/app/myapp`", the HTML document will be found in:

- `/opt/var/gas/appdata/app/myapp/webcomponents/3DCshart/3DChart.html`

To simplify deployment of gICAPI web components with the GAS, consider using the `fglgar` utility. For more details, see GAS documentation.

> **Note:** Unlike other front-ends, the GAS will not ask the VM for gICAPI web component files through the FGLIMAGEPATH mechanism, as described in the next section. The FGLIMAGEPATH mechanism applies only to front-ends using direct connection.

Centralizing gICAPI web component files for GMA, GMI and GDC front-ends (direct connection)

When using a front-end with a direct connection (i.e. not through the GAS), you can automatically transfer web component files to the front-end. Locate the gICAPI web component files on the computer where programs execute and set the FGLIMAGEPATH on page 182 environment variable. The web component files are automatically transferred if the program executes on a server and the gICAPI web component files are not found locally by the front-end.

When using FGLIMAGEPATH, gICAPI web component files are searched in the following order:

1. *FGLIMAGEPATH-location*`/webcomponents/`*component-type*`/`*component-type*`.html`
2. *FGLIMAGEPATH-location*`/`*component-type*`.html`

If assets such as `.js`, `.css`, `.png` files are referenced by a relative path name in the HTML content, the resources are also transferred via the FGLIMAGEPATH mechanism. If the assets use an absolute path with a concrete URL scheme (`http://something`), the HTML viewer will try to get the resource from the URL location.

> **Note:** Providing gICAPI web component files through FGLIMAGEPATH simplifies the development process for mobile applications, as you do not have to copy the files to the device.

For example, if you define the gICAPI web component field as follows:

```
WEBCOMPONENT wc = FORMOMLY.mychart,
   COMPONENTTYPE = "3DChart";
```

If the FGLIMAGEPATH search path contains "`/opt/myapp`", and the gICAPI files are located under /opt/myapp/webcomponents/3DChart", the gICAPI web component HTML document will be found on the server at:

- `/opt/myapp/webcomponents/3DChart/3DChart.html`

Deploying gICAPI web component files for an embedded mobile application

When running the application on mobile (i.e. in embedded mode), the gICAPI web component files (along with other assets) can be deployed on the device: The files will be found locally on the device.

> **Note:** The GDC front-end supports also local gICAPI file lookup in the GDC installation directory. However, this solution is supported for backward compatibility: Consider centralizing the gICAPI web component files on the application server, by using the GAS or the FGLIMAGEPATH mechanism as described above.

Mobile front-ends make a local search for gICAPI web component files in the following order:

1. *appdir*`/webcomponents/`*component-type*`/`*component-type*`.html`
2. *appdir*`/`*component-type*`.html`

Here *component-type* is the name defined by the `COMPONENTTYPE` attribute in the form definition file.

For more details about *appdir* on mobile devices, see Deploying mobile apps on Android devices on page 2572 and Deploying mobile apps on iOS devices on page 2584.

Defining the gICAPI files search path by program

To define the base URL to the web component files for a given application, you can also use the setWebComponentPath on page 1901 front call. The URL must be a well formatted absolute URL (e.g. `"http://myserver/components"` or `"file:///c:/components"`).

Important: This front-call is provided for backward compatibility, consider using one of the other mechanisms described in this topic.

Recommended web component directory layout

When using the default settings in any configuration (i.e. no FGLIMAGEPATH defined, default GAS settings), put the gICAPI web component files under a `webcomponents` directory, along with the other program files, for example:

```
appdir
appdir/main.42m
appdir/form1.42f
appdir/form2.42f
appdir/webcomponents/3DChart
appdir/webcomponents/3DChart/3DChart.html
appdir/webcomponents/3DChart/3DChart.js
appdir/webcomponents/3DChart/3DChart.css
appdir/webcomponents/3DChart/icon_close.png
...
```

Defining a gICAPI web component in forms

When defining a gICAPI web component in a form specification file, you can also provide a sizing policy and define additional properties.

Adding a WEBCOMPONENT to the form file

To define an gICAPI web component field, add a form field with the `WEBCOMPONENT` item type and the `COMPONENTTYPE` attribute. The `COMPONENTTYPE` attribute is mandatory when defining a gICAPI web component; it defines the root HTML file name describing the gICAPI web component.

A web component field is typically defined with the `FORMONLY` prefix, as the data for the field is rarely stored in a database column.

Sizing policy for web component fields

Web components are usually complex widgets displaying detailed information, such as charts, graphs, or calendars, which are generally resizeable. Use the appropriate form item attributes to get the expected layout and behavior. For more details, see Controlling the web component layout on page 1418.

Defining gICAPI web component properties

Since web component field definitions are generic, you must use the `PROPERTIES` attribute to set specific parameters for the component.

The `PROPERTIES` attribute can define a list of:

- simple properties (*`name = value`*),
- array properties (*`name = ( value1, value2, ... )`*)
- map/dictionary properties (*`name = ( sub-name1 = value1,sub-name2 = value2, ... )`*)

where *name* is a simple identifier, and where *values* can be numeric or string literals.

Component properties defined in the `PROPERTIES` attribute are transmitted to the web component through the `onProperty()` method of the `gICAPI` object.

The name of a property defined in the `PROPERTIES` attribute is converted to lowercase by the form compiler. To avoid mistakes, a good programming pattern is to define properties in lowercase, in both the interface script and in the form definition file. Property names are not checked at compile time, so nonexistent or mistyped properties will be ignored at runtime.

Example

```
LAYOUT
GRID
{
[wc                        ]
[                          ]
[                          ]
[                          ]
[                          ]
}
END
END
ATTRIBUTES
WEBCOMPONENT wc = FORMONLY.mychart,
         COMPONENTTYPE = "3DCharts",
         STRETCH = BOTH,
         PROPERTIES = ( type = "bars",
                        x_label = "Months",
                        y_label = "Sales" );
END
```

Controlling the gICAPI web component in programs

Controlling the gICAPI-based web components with ON ACTION

Once a `WEBCOMPONENT` field is defined in the form file with the `COMPONENTTYPE` attribute pointing to an HTML content file, it can be used as a regular edit field in program dialogs. The data of the gICAPI web component is transmitted with the field value, and usually needs to be serialized and deserialized (typically in JSON), when the data is not a simple scalar value.

When the web component field value is changed in the program, the `onData()` method of the `gICAPI` object is fired, and you can parse the serialized string in your JavaScript.

In order to detect web component value changes in the program, you need to combine the `gICAPI.setData()` and `gICAPI.Action()` methods, to transmit the value and to fire an action, that will be handled by an `ON ACTION` block.

> **Note:** The `ON CHANGE` trigger is not executed automatically for gICAPI-based web components, just by using `gICAPI.SetData()`.

The next example serializes and de-serializes a dymamic array using the JSON format:

```
IMPORT util
...
DEFINE mywc STRING
DEFINE data_array DYNAMIC ARRAY OF RECORD ...
...
...
INPUT BY NAME mywc, ...
      ATTRIBUTES(WITHOUT DEFAULTS, UNBUFFERED)
   ...
   ON ACTION set_wc_values -- Bound to form button
      LET mywc = util.JSON.stringify( data_array )

   ON ACTION wc_data_changed -- Triggered by gICAPI.Action()
      CALL util.JSON.parse( mywc, data_array )
...
```

> **Important:** All data will be transmitted through the abstract user interface protocol: Transmitting a lot of data will not be efficient and is likely to slow down your application.

Controlling the gICAPI-based web components with properties

Use the `PROPERTIES` attribute in the form specification, to define the configuration of the field. When a property of the web component is modified, the `onProperty()` method of the `gICAPI` object in the JavaScript will be invoked. Note that the complete property set will be passed, even if a single property is modified. Use JSON utilities to handle property set:

```
gICAPI.onProperty = function(propertySet) {
    var ps = eval('(' + propertySet + ')');
    document.getElementById("title").innerHTML = ps.title;
}
```

Controlling gICAPI-based web components with front calls

The web component can be manipulated with specific front calls. The web component-specific front calls are provided in the "`webcomponent`" front call module.

The `call` front call that can be used for general purposes. It takes as parameters the name of the form field, a JavaScript function to call, and optional parameters as required. The JavaScript function must be implemented in the HTML content of the gICAPI web component field. The front call returns the result of the JavaScript function.

```
DEFINE title STRING
CALL ui.Interface.frontCall("webcomponent", "call",
     ["formonly.mychart", "eval", "document.title"],
     [title] )
```

The `getTitle` function is another useful `webcomponent` front call that can get the title of the HTML document of the web component:

```
DEFINE info STRING
CALL ui.Interface.frontCall("webcomponent", "getTitle",
     ["formonly.url_field"], [info] )
```

Some providers return key information in the title of the HTML document.

Using image resources with the gICAPI web component

This section explains how to use image resources in a gICAPI web component.

Image resources in gICAPI web components

In some cases, web components require image resources, which can be classified as follows:

1. Common (static) image resources, that are part of the gICAPI web component implementation. This category of image resource can be referenced with absolute URLs (retrieved automatically by the HTML viewer), or can be deployed as part of the gICAPI web component assets, when referenced with relative URLs.
2. Private (variable) image resources, that are displayed by the program at runtime. This category of image resource can be referenced with absolute URLs (retrieved automatically by the HTML viewer), or can be provided by using the `ui.Interface.filenameToURI()` / FGLIMAGEPATH mechanism (as described below).

Referencing image resources in HTML

Image resources are typically referenced in HTML within the `<img/>` element, by setting the `src` attribute to a relative or absolute URL:

The following example uses an absolute URL:

```
<img src="http://www.4js.com/images/smiley.gif" alt="Smiley face" height=
 "42" width= "42" >
```

This example uses a relative URL:

```
<img src="smiley.gif" alt="Smiley face" height= "42" width= "42" >
```

The gICAPI web component framework can automatically retrieve image resources. If the value is not an absolute or relative URL that can be resolved by the HTML viewer, the image resources are retrieved from the Genero application using the the `ui.Interface.filenameToURI()` / FGLIMAGEPATH mechanism.

Providing static images in gICAPI web component files

To provide common static images as assets of your gICAPI web component, provide the image files along with the main HTML file, typically in a dedicated directory. For example, if you define the following directory structure:

```
3DChart/3DChart.html
3DChart/images/redraw.gif
3DChart/images/fetchdata.gif
```

The HTML content of the web component can reference common static images as follows:

```
<img src="images/redraw.gif" alt="Smiley face" height= "42" width= "42" >
```

Providing application images from Genero programs

Some gICAPI web components display variable image resources provided at runtime. For example, a photo gallery web component displaying pictures. Such image resources are usually private to the application.

To use image resources that are not static images part of the gICAPI web component assets:

1. Reference absolute URLs directly in the HTML content (in `"src"` attributes of image elements) with `http:`, `https:` or `file:` shemes, to be retrieved automatically by the HTML viewer, or:
2. Reference image resources in the HTML content with the URI returned from the `ui.Interface.filenameToURI()` method, to provide image files from the platform where the application executes (can be a server or mobile device):
   - When running the application on a server behind the GAS, the `filenameToURI()` method will convert the local file path to a URL that will make the image file available through the GAS.
   - When using a direct connection to the front-end (typical GDC desktop configuration with application running on a server), the file name will be returned as is and the images will then be transmitted through the FGLIMAGEPATH mechanism, as described in Providing the image resource on page 784.
   - When running apps on mobile devices, the `filenameToURI()` method will build the complete path to the local file, according to the list of directories defined in the FGLIMAGEPATH environment variable. The image resource is then directly read from the device file system.

   Try Example 5: Application images in gICAPI web component on page 1445, to see this method in practice.

Examples

Several examples show you how to include Web components in your program.

Example 1: URL-based web component using Google maps

This example shows how to implement a simple mobile application using a `WEBCOMPONENT` field interacting with Google maps

The form file: webcomp.per

```
LAYOUT
GRID
{
[f1                    ]
[                      ]
[                      ]
[                      ]
[                      ]
[                      ]
[                      ]
[                      ]
[f2                    ]
[f3                    ]
[                      ]
}
END
END
ATTRIBUTES
WEBCOMPONENT f1 = FORMONLY.mymap, STRETCH=BOTH;
BUTTONEDIT f2 = FORMONLY.location, ACTION=set_loc;
TEXTEDIT f3 = FORMONLY.value, STRETCH=X;
END
```

The program file: webcomp.4gl

```
MAIN
    CONSTANT c_gmaps = "http://maps.google.com/"
    DEFINE rec RECORD
                mymap STRING,
                location STRING,
                value STRING
           END RECORD
    OPEN FORM f1 FROM "webcomp"
    DISPLAY FORM f1
    LET rec.location = "Paris"
    LET rec.mymap = c_gmaps||"?q="||rec.location
    INPUT BY NAME rec.* WITHOUT DEFAULTS
          ATTRIBUTES(UNBUFFERED)
        ON ACTION set_loc
           LET rec.mymap = c_gmaps||"?q="||rec.location
           LET rec.value = rec.mymap
        ON CHANGE mymap
           LET rec.value = rec.mymap
           MESSAGE "URL has changed! "||CURRENT HOUR TO FRACTION(3)
    END INPUT
END MAIN
```

Example 2: Calling a JavaScript function of a gICAPI web component

This example shows how to call a JavaScript function with the "call" front call

The form file: wc_echo.per

```
ACTION DEFAULTS
  ACTION data_available(DEFAULTVIEW=NO)
```

```
END
LAYOUT
GRID
{
[data                                  ]
[                                      ]
[                                      ]
}
END
END
ATTRIBUTES
WEBCOMPONENT data = formonly.data,
   COMPONENTTYPE="wc_echo",
   STRETCH=BOTH;
END
```

The HTML file: wc_echo.html

```
<!DOCTYPE html>
<html>
<head>
  <title>The title</title>
  <script language="JavaScript" type="text/javascript" src="wc_echo.js"></
script>
</head>
<body>
<div style="background-color:green;width:3000px;height:3000px;" >
here
</div>
</body>
</html>
```

The JavaScript file: wc_echo.js

```
function echoString(str) {
  return str;
}

function echoObject(ostr) {
  var o = JSON.parse(ostr);
  // do something and return back
  return JSON.stringify(o);
}

onICHostReady = function(version) {

   if ( version != 1.0 )
      alert('Invalid API version');

   gICAPI.onProperty = function(p) {
      var myObject = eval('(' + p + ')');
      if (myObject.url!="") {
        setTimeout( function () {
          downloadURL(myObject.url);
        }, 0);
      }
   }

}
```

The program file: wc_echo.4gl

```
MAIN
  OPEN FORM f FROM "wc_echo"
  DISPLAY FORM f
  MENU "test"
    COMMAND "echo"
      CALL echo()
    COMMAND "exit"
      EXIT MENU
  END MENU
END MAIN

FUNCTION echo()
  DEFINE a,title,ut STRING
  TRY
    CALL ui.Interface.frontCall("webcomponent","call",
            ["formonly.data","eval","Math.floor(5/2)"],[ut])
    CALL ui.Interface.frontCall("webcomponent","getTitle",
            ["formonly.data"],[title])
    CALL ui.Interface.frontCall("webcomponent","call",
            ["formonly.data","echoString","hello"],[a])
    MESSAGE "ut:",ut,",a:",a,",title:",title
  CATCH
     ERROR err_get(status)
  END TRY
END FUNCTION
```

Example 3: Implementing Google+ authentication with a URL-based web component
This example shows how to authenticate the user with a google+ account on a mobile platform, using the OAuth technology.

The form file: wc_oauth.per

```
LAYOUT(text="Proceed to the authorization")
GRID
{
[f1                ]
[                  ]
[                  ]
}
END
END
ATTRIBUTES
WEBCOMPONENT f1 = FORMONLY.wc_oauth, STRETCH=BOTH;
END
```

The Google+ API utility file: wc_oauth.4gl

```
# Google+ Authorization API:
# See https://developers.google.com/accounts/docs/OAuth2InstalledApp

IMPORT com
IMPORT util

# The persistant datastore
PRIVATE DEFINE datastore RECORD
      client_id           STRING,
      client_secret       STRING,
      authorization_code  STRING,
      expiration_date     DATETIME YEAR TO SECOND,
      auth_data RECORD
         access_token    STRING,
```

```
         token_type      STRING,
         expires_in      INTEGER,
         id_token        STRING,
         refresh_token   STRING
      END RECORD,
      user_info RECORD
         id              STRING,
         name            STRING,
         link            STRING,    # google plus profile URL
         picture         STRING,    # face URL
         email           STRING
      END RECORD
   END RECORD

#+ This function checks if the google account is authorized and manages to
 get authorization
#+ @return boolean
FUNCTION googleplus_isAuthorized()
   DEFINE httpReq         com.HttpRequest
   DEFINE httpPostData    STRING
   DEFINE httpResp        com.HttpResponse
   DEFINE httpRespData    STRING
   DEFINE authUrl         STRING

   LET datastore.client_id = "****999.apps.googleusercontent.com"
   LET datastore.client_secret = "rlg*******-HUB"

   # Check token expiration
   IF datastore.expiration_date > CURRENT YEAR TO SECOND THEN
      RETURN TRUE
   END IF
   # The authorization token expired
   # If we already have an authorization, we need to refresh our token
   # See https://developers.google.com/accounts/docs/OAuth2InstalledApp?
hl=fr#refresh
   IF datastore.auth_data.refresh_token.getLength() > 2 THEN
      # Refresh the token
      LET httpReq = com.HttpRequest.Create("https://accounts.google.com/o/
oauth2/token")
      CALL httpReq.setMethod("POST")
      LET httpPostData =
         SFMT(
            "client_id=%1&client_secret=%2&refresh_token=
%3&grant_type=refresh_token",
            datastore.client_id,
            datastore.client_secret,
            datastore.auth_data.refresh_token
         )
   ELSE
      # Get an authorization code
      # See https://developers.google.com/accounts/docs/OAuth2InstalledApp?
hl=fr#formingtheurl
      LET authUrl =
         SFMT( "https://accounts.google.com/o/oauth2/auth?"
             ||"response_type=code"
             ||"&client_id=%1"
             ||"&redirect_uri=urn:ietf:wg:oauth:2.0:oob"
             ||"&scope=https://www.googleapis.com/auth/userinfo.email"
                ||"%%20https://www.googleapis.com/auth/userinfo.profile"
                ||"%%20https://www.googleapis.com/auth/plus.login",
            datastore.client_id
         )
      LET datastore.authorization_code =
 googleplus_getAuthorization(authUrl)
```

```
      IF datastore.authorization_code IS NULL THEN
         # User did not authorize the accesss to the data
         RETURN FALSE
      END IF
      # Ask for the first token
      # See https://developers.google.com/accounts/docs/OAuth2InstalledApp?
hl=fr#handlingtheresponse
      LET httpReq = com.HttpRequest.Create("https://accounts.google.com/o/
oauth2/token")
      CALL httpReq.setMethod("POST")
      LET httpPostData =
         SFMT("code=%1&client_id=%2&client_secret=%3"
            ||"&redirect_uri=urn:ietf:wg:oauth:2.0:oob"
            ||"&grant_type=authorization_code",
            datastore.authorization_code,
            datastore.client_id,
            datastore.client_secret
         )
   END IF
   TRY
      CALL httpReq.doFormEncodedRequest(httpPostData, FALSE)
      LET httpResp = httpReq.getResponse()
      IF httpResp.getStatusCode() <> 200 THEN
         RETURN FALSE
      END IF
      LET httpRespData = httpResp.getTextResponse()
      CALL util.JSON.parse(httpRespData, datastore.auth_data)
      LET datastore.expiration_date =
          CURRENT YEAR TO SECOND + ((datastore.auth_data.expires_in - 60)
 UNITS SECOND)
      RETURN (datastore.expiration_date > CURRENT YEAR TO SECOND)
   CATCH
      # Network error...
      RETURN FALSE
   END TRY
END FUNCTION

#+ This function manages the authentication and authorization UI for Google+
#+ @param authorizationUrl the built URL to display the authorization dialog
 on google website
#+ @return The authorization code
FUNCTION googleplus_getAuthorization(authorizationUrl)
   DEFINE authorizationUrl  STRING
   DEFINE authorizationCode STRING
   DEFINE wc_oauth          STRING
   DEFINE doc_title         STRING
   DEFINE flag              BOOLEAN

   DEFINE authorizationCodeBegin INTEGER
   DEFINE authorizationCodeEnd   INTEGER

   OPEN WINDOW w_oauth WITH FORM "wc_oauth"
   LET authorizationCode = NULL
   LET wc_oauth = authorizationUrl
   INPUT BY NAME wc_oauth ATTRIBUTES(WITHOUT DEFAULTS, ACCEPT=FALSE)
      ON CHANGE wc_oauth -- a new page is loaded in the webview
         CALL ui.Interface.frontCall("webcomponent","getTitle",
["formonly.wc_oauth"],[doc_title])
         IF doc_title.getIndexOf("Success", 1) == 1 THEN
            LET authorizationCodeBegin = doc_title.getIndexOf("code=", 1)
            LET authorizationCodeEnd =  doc_title.getIndexOf("&",
 authorizationCodeBegin)
            IF authorizationCodeEnd = 0 THEN
               LET authorizationCodeEnd = doc_title.getLength() + 1
```

```
            END IF
            LET authorizationCode =
 doc_title.subString(authorizationCodeBegin+5, authorizationCodeEnd - 1)
            EXIT INPUT
         END IF
      ON ACTION cancel
         MENU "Confirmation"
              ATTRIBUTES(STYLE="dialog", COMMENT="Cancel the authorization
 process?")
            ON ACTION accept
               LET flag = TRUE
            ON ACTION cancel
               LET flag = FALSE
         END MENU
         IF flag THEN
            LET authorizationCode = NULL
            EXIT INPUT
         END IF
   END INPUT
   CLOSE WINDOW w_oauth
   RETURN authorizationCode
END FUNCTION
```

The program file: main.4gl

```
IMPORT FGL wc_oauth
MAIN
   MENU
       ON ACTION get_auth
          IF googleplus_isAuthorized() THEN
             MESSAGE "Google+ authorization acquired."
          ELSE
             ERROR "Unable to get Google+ authorization."
          END IF
       ON ACTION close
          EXIT MENU
   END MENU
END MAIN
```

Example 4: Color picker gICAPI web component
This topic describes the different steps to implement a gICAPI-based web component.

Introduction

In this example, we will implement a simple color picker, that will allow the user the select a color from a predefined set. Colors are drawn as square boxes using SVG graphics, user can change the current selected color with a separate `COMBOBOX` field, modify the title of the HTML body, and query for the color list with a `webcomponent.call` front call.

The HTML file is described in detail, and complete code example with program and form file is available at the end of this topic.

HTML code description

As any HTML source code, the file starts with the typical HTML tags:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html" charset="utf-8" />
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
```

Note: The "viewport" meta is provided to adjust the viewport size for mobile devices.

The JavaScript™ code needs to the enclosed in a `<script>` element:

```
<script language="JavaScript">
```

Global variables are defined to hold information that must be persistent during the web component life:

```
var current_color;
var wanted_color;
var has_focus;
```

The global function `onICHostReady()` will be called by the front end, when the web component interface is ready. The version passed as parameter allows you to check that your component code is compatible with the current gICAPI framework, and to define and assign the `gICAPI.on*` callback methods (these will be defined in the body of the `onICHostReady()` function:

```
onICHostReady = function(version) {

    if ( version != 1.0 )
        alert('Invalid API version');

    ... some initialization code ...

    gICAPI.onProperty = function(propertySet) {
        ... see below for function body ...
    }

    gICAPI.onData = function(data) {
        ... see below for function body ...
    }

    gICAPI.onFocus = function(polarity) {
        ... see below for function body ...
    }

}
```

At this point, the gICAPI interface is ready and the `gICAPI` object can be used.

The `onProperty()` method is called when a web component property changes (properties will be initialized at form creation, or changed during form usage). In this code example, when the property "title" is changed by the program, the element with id="title" is updated with the new title. The `eval` built-in JavaScript function is used to convert the JSON string property set to a JSON object, to find the "title" property:

```
gICAPI.onProperty = function(propertySet) {
    var ps = eval('(' + propertySet + ')');
    document.getElementById("title").innerHTML = ps.title;
}
```

Note: The `ON ACTION change_title` in the dialog code will change the title property after the form initialization, to show that the `onProperty()` function can also be invoked after the web component field creation.

The `showFocusRectangle()` function shows a border around the specified color item (SVG element), according to the color identifier passed as parameter and the focus status (focus can be true, false or -1, to keep the current border color and just modify the position of the border):

```
showFocusRectangle = function(color, focus) {
```

```
    // See complete code example for details
}
```

The `changeColor()` function implements a color change, by registering a field value change with `gICAPI.SetData()`, and by triggering a specific action with `gICAPI.Action()`, to inform the program that a color was selected:

```
changeColor = function(color) {
    current_color = color;
    showFocusRectangle(current_color, true);
    gICAPI.SetData(current_color);
    gICAPI.Action("color_selected");
}
```

Next lines implement the `onFocus()` function, executed when the web component gets or loses the focus. The code distinguishes the case when the focus is gained (by a mouse click on a color item), selecting a new color with a call to `changeColor()`, and the case when the focus is set to the web component by the runtime system. A blue border will be added to the current color item, when the component gets the focus, and the border color is reset to gray when the focus is lost.:

```
gICAPI.onFocus = function(polarity) {
    if ( polarity == true ) {
        has_focus = true;
        if (wanted_color != undefined) {
            changeColor(wanted_color);
            wanted_color = undefined;
        } else {
            showFocusRectangle(current_color, true);
        }
    } else {
        has_focus = false;
        showFocusRectangle(current_color, false);
    }
}
```

The `onData()` function must be implemented to detect web component value changes done in the program, and to acknowledge . This will be triggered by assigning the `rec.webcomp` variable in the dialog code, typically in the `ON CHANGE color` block, when modifying the combobox value. The `showFocusRectangle()` function moves the focus border to the color item corresponding to the color identifier passed as parameter.:

```
gICAPI.onData = function(data) {
    current_color = data;
    showFocusRectangle(current_color, -1);
}
```

The `selectColor()` function will be called through the `onclick` event of the `<rect>` SVG elements representing colors. If the web component does not have the focus yet, the function will call `gICAPI.SetFocus()`, in order to ask the runtime system, if the focus can go to the web component field. If the runtime system accepts to set the focus to the web component field, the `onFocus()` method will be called with `true` as parameter, and will handle the requested color change (using `wanted_color`). if the focus cannot be set to the web component, the `onFocus()` method will not be called:

```
selectColor = function(color) {
    if (has_focus) {
        changeColor(color);
    } else {
        wanted_color = color;
        gICAPI.SetFocus();
        // Color item change is done in onFocus(), because
```

```
        // VM may refuse to set the focus to the wc field.
    }
}
```

> **Note:** The only way to detect that the focus was gained by the web compoment field, is when `onFocus(true)` is called.

End the JavaScript element with the `</script>` ending tag:

```
</script>
```

Close the HTML head element with the `</head>` ending tag:

```
</head>
```

The rest of the HTML page defines the graphical elements for the color picker, with a <h3> title and and an `<svg>` element containing `<rect>` element to show clickable color items. Note the `<rect>` element with `id="focus_rectangle"`, used to show a border for the current color item:

```
<body height="100%" width="100%">
<h3 id="title">no-title</h3>

<svg id="svg_container" width="230" height="130">

  <rect x="5" y= "5" rx="5" ry="5" width="30" height="30"
        id="#FFFFCC"
        style="fill:#FFFFCC;stroke:black;stroke-width:1"
        onclick="selectColor('#FFFFCC')" />
  ...

  <rect x="178" y= "73" rx="7" ry="7" width="34" height="34"
        id="focus_rectangle"
        style="fill:none;stroke:gray;stroke-width:3" />

</svg>

</body>
```

Complete source code

File `color_picker.per`:

```
ACTION DEFAULTS
  ACTION color_selected ( DEFAULTVIEW = NO )
END
LAYOUT
GRID
{
 Id:    [f1            ]
[f2                     ]
[                       ]
[                       ]
[                       ]
[                       ]
[                       ]
[                       ]
[                       ]
 Color: [f3             ]
[f4                     ]
[                       ]
[                       ]
```

```
}
END
END
ATTRIBUTES
EDIT f1 = FORMONLY.id;
WEBCOMPONENT f2 = FORMONLY.webcomp,
   COMPONENTTYPE="color_picker",
   PROPERTIES = (title="My color picker"),
   STRETCH=BOTH;
COMBOBOX f3 = FORMONLY.pgcolor, NOT NULL,
   ITEMS=( "#FFFFCC", "#FFFFAA", "#FFFF00",
           "#FFAD99", "#FF0000", "#990000",
           "#99CCFF", "#0066FF", "#000099",
           "#FF99FF", "#FF00FF", "#990099",
           "#99FF99", "#009933", "#006600",
           "#FFFFFF", "#AAAAAA", "#000000" );
TEXTEDIT f4 = FORMONLY.info, STRETCH=X;
END
```

File `color_picker.4gl`:

```
IMPORT util

MAIN
    DEFINE rec RECORD
               id INTEGER,
               webcomp STRING,
               pgcolor STRING,
               info STRING
           END RECORD,
           f ui.Form,
           n om.DomNode,
           tmp STRING,
           colors DYNAMIC ARRAY OF STRING

    OPTIONS INPUT WRAP

    OPEN FORM f1 FROM "color_picker"
    DISPLAY FORM f1

    LET rec.id = 98344
    LET rec.webcomp = "#FF0000"
    LET rec.pgcolor = rec.webcomp

    INPUT BY NAME rec.* WITHOUT DEFAULTS
          ATTRIBUTES(UNBUFFERED)

        ON CHANGE pgcolor
           LET rec.webcomp = rec.pgcolor

        ON ACTION color_selected
           IF rec.webcomp == "#000000" THEN
              LET rec.webcomp = rec.pgcolor
              LET rec.info = NULL
              ERROR "Black color is denied!"
           ELSE
              LET rec.pgcolor = rec.webcomp
              LET rec.info = "Color selected:", rec.pgcolor
           END IF

        ON ACTION change_title ATTRIBUTES(TEXT="Change title")
           LET f = DIALOG.getForm()
           LET n = f.findNode("Property", "title")
```

```
            CALL n.setAttribute("value", "New title")
            LET rec.info = "Title changed."

        ON ACTION get_colors ATTRIBUTES(TEXT="Get colors")
            TRY
                CALL ui.Interface.frontCall("webcomponent", "call",
                        ["formonly.webcomp", "getColorList"], [tmp] )
                CALL util.JSON.parse(tmp, colors)
                LET rec.info = "Color list: ", tmp
            CATCH
                ERROR "Front call failed."
            END TRY

    END INPUT

END MAIN
```

File `color_picker.html`:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html" charset="utf-8" />
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />

<script language="JavaScript">

var current_color;
var wanted_color;
var has_focus;

onICHostReady = function(version) {

    if ( version != 1.0 )
        alert('Invalid API version');

    current_color = "#000000";

    gICAPI.onProperty = function(properties) {
        var ps = eval('(' + properties + ')');
        document.getElementById("title").innerHTML = ps.title;
    }

    gICAPI.onFocus = function(polarity) {
        if ( polarity == true ) {
            has_focus = true;
            if (wanted_color != undefined) {
                changeColor(wanted_color);
                wanted_color = undefined;
            } else {
                showFocusRectangle(current_color, true);
            }
        } else {
            has_focus = false;
            showFocusRectangle(current_color, false);
        }
    }

    gICAPI.onData = function(data) {
        current_color = data;
        showFocusRectangle(current_color, -1);
    }
```

```
}

showFocusRectangle = function(color, focus) {
    var f = document.getElementById("focus_rectangle");
    var e = document.getElementById(color);
    if (e == null) {
        e = document.getElementById("#000000");
    }
    var e_x = e.getAttribute("x") - 2;
    var e_y = e.getAttribute("y") - 2;
    f.setAttribute("x", e_x );
    f.setAttribute("y", e_y );
    if (focus == true) {
        f.style.stroke = "blue";
    } else if (focus == false) {
        f.style.stroke = "gray";
    }
}

changeColor = function(color) {
    current_color = color;
    showFocusRectangle(current_color, true);
    gICAPI.SetData(current_color);
    gICAPI.Action("color_selected");
}


selectColor = function(color) {
    if (has_focus) {
        changeColor(color);
    } else {
        wanted_color = color;
        gICAPI.SetFocus();
        // Color item change is done in onFocus(), because
        // VM may refuse to set the focus to the wc field.
    }
}

</script>

</head>

<body height="100%" width="100%">
<h3 id="title">no-title</h3>

<svg id="svg_container" width="230" height="130">

  <rect x="5" y= "5" rx="5" ry="5" width="30" height="30"
        id="#FFFFCC"
        style="fill:#FFFFCC;stroke:black;stroke-width:1"
        onclick="selectColor('#FFFFCC')" />
  <rect x="5" y="40" rx="5" ry="5" width="30" height="30"
        id="#FFFFAA"
        style="fill:#FFFFAA;stroke:black;stroke-width:1"
        onclick="selectColor('#FFFFAA')" />
  <rect x="5" y="75" rx="5" ry="5" width="30" height="30"
        id="#FFFF00"
        style="fill:#FFFF00;stroke:black;stroke-width:1"
        onclick="selectColor('#FFFF00')" />

  <rect x="40" y= "5" rx="5" ry="5" width="30" height="30"
        id="#FFAD99"
        style="fill:#FFAD99;stroke:black;stroke-width:1"
        onclick="selectColor('#FFAD99')" />
```

```
  <rect x="40" y="40" rx="5" ry="5" width="30" height="30"
        id="#FF0000"
        style="fill:#FF0000;stroke:black;stroke-width:1"
        onclick="selectColor('#FF0000')" />
  <rect x="40" y="75" rx="5" ry="5" width="30" height="30"
        id="#990000"
        style="fill:#990000;stroke:black;stroke-width:1"
        onclick="selectColor('#990000')" />

  <rect x="75" y= "5" rx="5" ry="5" width="30" height="30"
        id="#99CCFF"
        style="fill:#99CCFF;stroke:black;stroke-width:1"
        onclick="selectColor('#99CCFF')" />
  <rect x="75" y="40" rx="5" ry="5" width="30" height="30"
        id="#0066FF"
        style="fill:#0066FF;stroke:black;stroke-width:1"
        onclick="selectColor('#0066FF')" />
  <rect x="75" y="75" rx="5" ry="5" width="30" height="30"
        id="#000099"
        style="fill:#000099;stroke:black;stroke-width:1"
        onclick="selectColor('#000099')" />

  <rect x="110" y= "5" rx="5" ry="5" width="30" height="30"
        id="#FF99FF"
        style="fill:#FF99FF;stroke:black;stroke-width:1"
        onclick="selectColor('#FF99FF')" />
  <rect x="110" y="40" rx="5" ry="5" width="30" height="30"
        id="#FF00FF"
        style="fill:#FF00FF;stroke:black;stroke-width:1"
        onclick="selectColor('#FF00FF')" />
  <rect x="110" y="75" rx="5" ry="5" width="30" height="30"
        id="#990099"
        style="fill:#990099;stroke:black;stroke-width:1"
        onclick="selectColor('#990099')" />

  <rect x="145" y= "5" rx="5" ry="5" width="30" height="30"
        id="#99FF99"
        style="fill:#99FF99;stroke:black;stroke-width:1"
        onclick="selectColor('#99FF99')" />
  <rect x="145" y="40" rx="5" ry="5" width="30" height="30"
        id="#009933"
        style="fill:#009933;stroke:black;stroke-width:1"
        onclick="selectColor('#009933')" />
  <rect x="145" y="75" rx="5" ry="5" width="30" height="30"
        id="#006600"
        style="fill:#006600;stroke:black;stroke-width:1"
        onclick="selectColor('#006600')" />

  <rect x="180" y= "5" rx="5" ry="5" width="30" height="30"
        id="#FFFFFF"
        style="fill:#FFFFFF;stroke:black;stroke-width:1"
        onclick="selectColor('#FFFFFF')" />
  <rect x="180" y="40" rx="5" ry="5" width="30" height="30"
        id="#AAAAAA"
        style="fill:#AAAAAA;stroke:black;stroke-width:1"
        onclick="selectColor('#AAAAAA')" />
  <rect x="180" y="75" rx="5" ry="5" width="30" height="30"
        id="#000000"
        style="fill:#000000;stroke:gray;stroke-width:1"
        onclick="selectColor('#000000')" />

  <rect x="178" y= "73" rx="7" ry="7" width="34" height="34"
        id="focus_rectangle"
        style="fill:none;stroke:gray;stroke-width:3" />
```

```
</svg>

</body>
</html>
```

Example 5: Application images in gICAPI web component
This topic shows how to display application images in a gICAPI-based web component.

Introduction

In this example, we will focus on the technique to display application images dynamically in gICAPI web component HTML content, by using the `ui.Interface.filenameToURI()` method.

This sample application can be used with any Genero front-end configuration (as a web application with the GAS, in direct (development) mode with GDC/GMA/GMI, or as a mobile app running on a device)

For gICAPI programming basics, see Example 4: Color picker gICAPI web component on page 1437.

The complete code example with program and form file is available at the end of this topic.

HTML code description

The HTML source file starts with the typical HTML tags:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html" charset="utf-8" />
<meta name='viewport' content='initial-scale=1.0, maximum-scale=1.0' />
```

The JavaScript™ code defines the `onICHostReady()`. This function checks for the API version and defines the `set_image()` JavaScript function that will set the `src` attribute in the image element:

```
<script language="JavaScript">
  onICHostReady = function(version) {

    if ( version != 1.0 )
        alert('Invalid API version');

    set_image = function(ressource) {
        var ie=document.getElementsByName("myimage")[0];
        ie.src=ressource;
    }

  }
</script>
```

Close the HTML head element with the `</head>` ending tag:

```
</head>
```

The body of the HTML page contains two elements:

- an `h2` title,
- the `image` element, identified by a name:

```
<body height="100%" width="100%">
    <h2>Testing application images in gICAPI Web Component</h2>
    <img name="myimage" />
  </body>
```

```
</html>
```

Application directory structure

In order to easily build and install on mobile devices, create the following directory structure:

```
top-dir
|
|-- fglprofile
|-- main.4gl
|-- main.42m
|-- myform.per
|-- myform.42f
|-- images                  (application image files)
|   |-- image01.jpg
|   |-- image02.jpg
|   |-- image03.jpg
|   ...
|-- webcomponents
|   |-- mywebcomp
|       |-- mywebcomp.html
|
|-- gmi
|   |-- iOS resources       (icons, etc)
|   ...
|-- gma
|   |-- Android resources   (icons, etc)
|   ...
|
```

For more details about building mobile apps from the command line, see Deploying mobile apps on page 2572.

Providing image files

Copy some of your favorite images in the "`images`" directory.

The sample program will scan this directory to fill a combobox and let you choose the image to be displayed:

```
FUNCTION init_image_list(cb)
    DEFINE cb ui.ComboBox
    DEFINE h INTEGER,
           fn STRING
    LET
 h=os.Path.dirOpen(os.Path.join(base.Application.getProgramDir(),"images"))
    WHILE h > 0
        LET fn = os.Path.dirNext(h)
        IF fn IS NULL THEN EXIT WHILE END IF
        IF fn=="." OR fn==".." THEN CONTINUE WHILE END IF
        CALL cb.addItem(fn, fn)
    END WHILE
END FUNCTION
```

> **Note:** When deployed on a mobile device, the images directory will be part of the application program files. Thus to access the directory you need to add the `base.Application.getProgramDir` on `page 1705` path. For more details, see Directory structure for GMA apps on page 2572 and Directory structure for GMI apps on page 2584.

In the program code, the `ON CHANGE image` interaction block will perform a front call to set the image resource in the gICAPI web component:

```
  ON CHANGE image
     LET rec.uri = ui.Interface.filenameToURI(rec.image)
     CALL ui.Interface.frontCall("webcomponent","call",
                      ["formonly.wc","set_image",rec.uri],[])
```

FGLIMAGEPATH environment settings

In order to find image resources when not executing behing a GAS, you need to define the FGLIMAGEPATH environment variable as follows:

```
$ FGLIMAGEPATH=$PWD/images:.
```

For deployed mobile applications, the FGLIMAGEPATH environment variable must be set in the default fglprofile file, by using the $FGLAPPDIR place holder:

```
mobile.environment.FGLIMAGEPATH = "$FGLAPPDIR/images:."
```

For more details about FGLIMAGEPATH settings, see Providing the image resource on page 784.

Complete source code

File `myform.per`:

```
LAYOUT
GRID
{
Current image: [f1                  ]
Image URI:     [f2                  ]
[wc                                 ]
[                                   ]
[                                   ]
[                                   ]
[                                   ]
}
END
END

ATTRIBUTES
COMBOBOX f1 = FORMONLY.image,
     INITIALIZER = init_image_list;
EDIT f2 = FORMONLY.uri, SCROLL;
WEBCOMPONENT wc = FORMONLY.wc,
   COMPONENTTYPE="mywebcomp",
   STRETCH=BOTH;
END
```

File `main.4gl`:

```
IMPORT os

MAIN
    DEFINE rec RECORD
                image STRING,
                uri STRING,
                wc STRING
            END RECORD
    OPEN FORM f1 FROM "myform"
```

```
    DISPLAY FORM f1
    INPUT BY NAME rec.* WITHOUT DEFAULTS ATTRIBUTES(UNBUFFERED)
        ON CHANGE image
           LET rec.uri = ui.Interface.filenameToURI(rec.image)
           CALL ui.Interface.frontCall("webcomponent","call",
                               ["formonly.wc","set_image",rec.uri],[])
    END INPUT
END MAIN

FUNCTION init_image_list(cb)
    DEFINE cb ui.ComboBox
    DEFINE h INTEGER,
           fn STRING
    LET
 h=os.Path.dirOpen(os.Path.join(base.Application.getProgramDir(),"images"))
    WHILE h > 0
        LET fn = os.Path.dirNext(h)
        IF fn IS NULL THEN EXIT WHILE END IF
        IF fn=="." OR fn==".." THEN CONTINUE WHILE END IF
        CALL cb.addItem(fn, fn)
    END WHILE
END FUNCTION
```

File `mywebcomp.html`:

```
<!DOCTYPE html>
<html>
<head>
<title>Test</title>
<meta Http-Equiv="Cache-Control" Content="no-cache">
<meta Http-Equiv="Pragma" Content="no-cache">
<meta Http-Equiv="Expires" Content="0">
<script LANGUAGE="JavaScript">
    onICHostReady = function(version) {

        if ( version != 1.0 )
            alert('Invalid API version');

        set_image = function(ressource) {
            var ie=document.getElementsByName("myimage")[0];
            ie.src=ressource;
        }

    }
</script>
</head>
  <body>
    <h2>Testing application images in gICAPI Web Component</h2>
    <img name="myimage" />
  </body>
</html>
```

Canvases

Canvases are form drawing areas.

- Understanding canvases on page 1449
- CANVAS item definition on page 936
- Syntax of canvas nodes on page 1450
- Using canvases on page 1451
  - Canvas drawing area on page 1451

- Step by step canvas example on page 1452
- Canvas drawing functions on page 1453

Understanding canvases

A canvas element defines a drawing area in a form, to show basic colored shapes.

> **Important:** This feature is not supported on mobile platforms.

Canvas can draw lines, rectangles, ovals, circles, texts, arcs, and polygons. Keys can be bound to graphical elements for selection with a right or left mouse click.

In programs, you select a given canvas area by name and you create the shapes in the abstract user interface tree by using the built-in DOM API, or helper functions.

The painted canvas is automatically displayed on the front end when an interactive instruction is executed, such as `MENU` or `INPUT`.

Each canvas element is identified by a unique number (id). You can use this identifier to bind mouse clicks to canvas elements.

> **Note:** Consider using Web Components for specific drawing needs (charts, graphics). For example using SVG graphics in a Web Component is more powerfull as the Canvas framework.

CANVAS item definition

Defines an area in which you can draw shapes, in a grid-based layout.

Syntax

```
CANVAS item-tag: item-name [  , attribute-list ] ;
```

1. *item-tag* is an identifier that defines the name of the item tag in the layout section.
2. *item-name* identifies the form item.
3. *attribute-list* defines the aspect and behavior of the form item.

Attributes

`COMMENT`, `HIDDEN`, `TAG`.

Usage

Define the rendering and behavior of a canvas drawing area item tag, with a `CANVAS` element in the `ATTRIBUTES` section.

> **Note:** The `CANVAS` feature is deprecated, consider using a `WEBCOMPONENT` with SVG graphics.

Example

```
LAYOUT
GRID
{
[cvs1             ]
[                 ]
[                 ]
 ...

}
END
END

ATTRIBUTES
```

```
CANVAS cvs1: canvas1;
...
```

Syntax of canvas nodes

Canvas areas are defined in forms with the following XML syntax:

```
<Canvas colName="name" >
{ <CanvasArc canvasitem-attribute="value" [...] />
| <CanvasCircle canvasitem-attribute="value"[...] />
| <CanvasLine canvasitem-attribute="value"[...] />
| <CanvasOval canvasitem-attribute="value"[...] />
| <CanvasPolygon canvasitem-attribute="value"[...] />
| <CanvasRectangle canvasitem-attribute="value"[...] />
| <CanvasText canvasitem-attribute="value"[...] />
} [...]
</Canvas>
[...]
```

Table 297: Types of canvas element

Name	Description
CanvasArc	Arc defined by the bounding square top left point, a diameter, a start angle, a end angle, and a fill color.
CanvasCircle	Circle defined by the bounding square top left point, a diameter, and a fill color.
CanvasLine	Line defined by a start point, an end point, width, and a fill color.
CanvasOval	Oval defined by rectangle (with start point and endpoint), and a fill color.
CanvasPolygon	Polygon defined by a list of points, and a fillcolor.
CanvasRectangle	Rectangle defined by a start point, an end point,and a fill color.
CanvasText	Text defined by a start point, an anchor hint, the text, and a fill color.

Table 298: Attributes of canvas elements

Name	Values	Description
startX	INTEGER (0->1000)	X position of starting point.
startY	INTEGER (0->1000)	Y position of starting point.
endX	INTEGER (0->1000)	X position of ending point.
endY	INTEGER (0->1000)	Y position of ending point.
xyList	STRING	Space-separated list of Y X coordinates. For example: "23 45 56 78" defines (x=23,y=45) (x=56,y=78).
width	INTEGER	Width of the shape.

Name	Values	Description
`height`	`INTEGER`	Height of the shape.
`diameter`	`INTEGER`	Diameter for circles and arcs.
`startDegrees`	`INTEGER`	Beginning of the angular range occupied by an arc.
`extentDegrees`	`INTEGER`	Size of the angular range occupied by an arc.
`text`	`STRING`	The text to draw.
`anchor`	`"n","e","w","s"`	Anchor hint to give the draw direction for texts.
`fillColor`	`STRING`	Name of the color to be used for the element.
`acceleratorKey1`	`STRING`	Name of the key associated to a left button click.
`acceleratorKey3`	`STRING`	Name of the key associated to a right button click.

Using canvases

Canvas drawing area

The canvas area represents an abstract drawing page where you define size and location of shapes with coordinates from (0,0) to (1000,1000).

The origin point (0,0), is on the left-bottom of the drawing area.

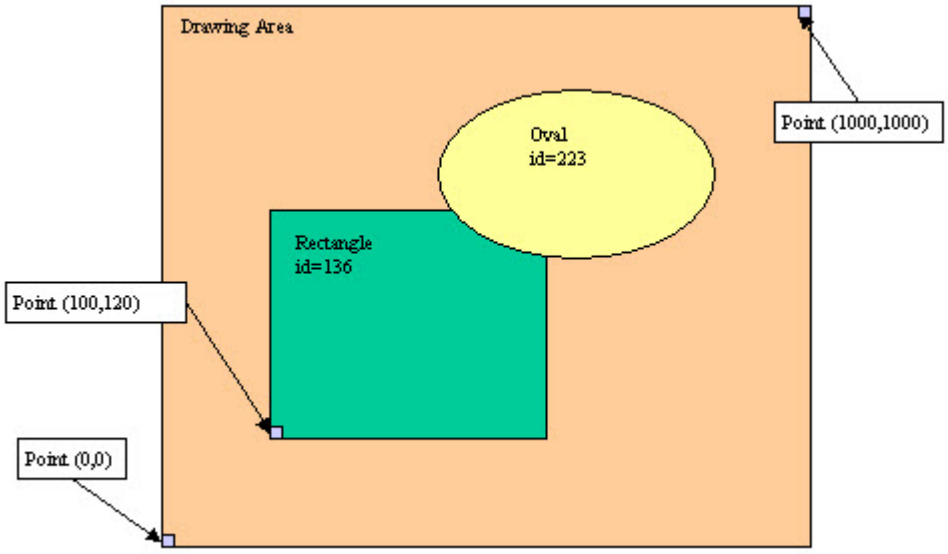


Figure 101: Canvas area diagram

The drawing area is defined in the form file with a `CANVAS` form item. At runtime, you draw the content of canvas areas in the Abstract User Interface tree: In a form defining canvas areas, the Abstract User Interface tree contains empty `<Canvas>` nodes that you can fill with canvas items.

A canvas node is identified in the program by the `name` attribute. You can get the canvas node by name with the `Window.getElement(`*name*`)` method.

You cannot drop canvas area nodes, as they are read-only in a form definition.

Step by step canvas example

First define a drawing area in the form file with the CANVAS form item type. In this example, the name of the canvas field is '`canvas01`'. This field name identifies the drawing area:

```
DATABASE FORMONLY
LAYOUT
GRID
{
  Canvas example:
  [ca01                         ]
  [                             ]
  [                             ]
  [                             ]
  [                             ]
  [                             ]
}
END
END
ATTRIBUTES
CANVAS ca01: canvas01;
END
```

In programs, you draw canvas shapes by creating canvas nodes in the abstract user interface tree with the DOM API utilities.

Define a variable to hold the DOM node of the canvas and a second to handle children created for shapes:

```
DEFINE c, s om.DomNode
```

Define a window object variable; open a window with the form containing the canvas area; get the current window object, and then get the canvas DOM node:

```
DEFINE w ui.Window
OPEN WINDOW w1 WITH FORM "form1"
LET w = ui.Window.getCurrent()
LET c = w.findNode("Canvas","canvas01")
```

Create a child node with a specific type defining the shape:

```
LET s = c.createChild("CanvasRectangle")
```

Set attributes to complete the shape definition:

```
CALL s.setAttribute( "fillColor", "red" )
CALL s.setAttribute( "startX", 10 )
CALL s.setAttribute( "startY", 20 )
CALL s.setAttribute( "endX", 100 )
CALL s.setAttribute( "endY", 150 )
```

It is possible to bind keys / actions to Canvas items in order to let the end user select elements with a mouse click. You can assign a function key for left-button mouse clicks with the `acceleratorKey1` attribute, while `acceleratorKey2` is used to detect right-button mouse clicks. The function keys you can bind are F1 to F255. If the user clicks on a Canvas item bound to key actions, the corresponding action handler will be executed in the current dialog. Several canvas items can be bound to the same action keys; In order to identify what items have been selected by a mouse click, you can use the

`drawGetClickedItemId()` function of `fgldraw.4gl`. This method will return the AUI tree node id of the Canvas items that was selected (i.e. s.getId()).

```
... Create the Canvas item with s node variable ...
CALL s.setAttribute( "acceleratorKey1", "F50" )
MENU "test"
   COMMAND KEY (F50)
      IF drawGetClickedItemId() = s.getId() THEN
        ...
      END IF
...
END MENU
```

To clear a given shape in the canvas, remove the element in the canvas node:

```
CALL c.removeChild(s)
```

To clear the drawing area completely, remove all children of the canvas node:

```
LET s=c.getFirstChild()
WHILE s IS NOT NULL
  CALL c.removeChild(s)
  LET s=c.getFirstChild()
END WHILE
```

Canvas drawing functions

This table describes the helper functions provided to ease canvas usage. Use these functions or use the DOM API to directly create canvas elements in the form. The helper functions are implemented in `FGLDIR/src/fgldraw.4gl`. See the source file for more details.

Table 299: CANVAS Built-in functions provided for backward compatibility with version 3

Name	Description
`drawInit()`	Initializes the drawing API. It is mandatory to call this function at the beginning of your program, before the first display instruction.
`drawSelect()`	Selects a canvas area for drawing.
`drawDisableColorLines()`	By default, simple lines drawn with `drawLine()` are colored by `drawFillColor()`. Pass `TRUE` to the function to get black lines.
`drawFillColor()`	Defines the fill color for shapes and lines. Color value are named colors like "red", "green", "blue"...
`drawLineWidth()`	Defines the width of lines.
`drawAnchor()`	Defines the anchor hint for texts.
`drawLine()`	Draws a line in the selected canvas.
`drawCircle()`	Draws a circle in the selected canvas.
`drawArc()`	Draws an arc in the selected canvas.
`drawRectangle()`	Draws a rectangle in the selected canvas.
`drawOval()`	Draws an oval in the selected canvas.
`drawText()`	Draws a text in the selected canvas.

Name	Description
`drawPolygon()`	Draws a polygon in the selected canvas.
`drawClear()`	Clears the selected canvas.
`drawButtonLeft()`	Enables left mouse click on a canvas element.
`drawButtonRight()`	Enables right mouse click on a canvas element.
`drawClearButton()`	Disables all mouse clicks on a canvas element.
`drawGetClickedItemId()`	Returns the id of the last clicked canvas element

Start menus

Start menus define a tree of application programs that can be started.

- Understanding start menus on page 1454
- Syntax of start menu files (.4sm) on page 1454
- Using start menus on page 1456
  - Loading a start menu from an XML file on page 1456
  - Creating the start menu dynamically on page 1456
- Examples on page 1456
  - Example 1: Start menu in XML format on page 1456
  - Example 2: Start menu created dynamically on page 1457

Understanding start menus

The start menu defines a tree of commands that start programs on the application server where the runtime system executes.

> **Important:** This feature is not supported on mobile platforms.

It is recommended that you create a specific program dedicated to running the start menu. This program must create (or load) a start menu, and then perform an interactive instruction to enter the interaction loop.

The start menu must be defined in the abstract user interface tree under the "`UserInterface`" root node.

The start menu is unique for a program and cannot be redefined.

When a start menu command is selected by the user, the runtime system automatically starts a child process with the command specified in the command attribute.

Syntax of start menu files (.4sm)

Start menus are defined in a .4sm file with the following XML syntax:

```
<StartMenu [ startmenu-attribute="value"[...] ] >
  group[...]
</StartMenu>
```

where *group* is:

```
<StartMenuGroup group-attribute="value"
    [...]>{ <StartMenuSeparator/>| <StartMenuCommand
     command-attribute="value"
    [...] />|
    group}
    [...]
</StartMenuGroup>
```

1. *startmenu-attribute*defines a property of the `StartMenu`.
2. *command-attribute* defines a property of a `StartMenuCommand`.
3. *group-attribute* defines a property of a `StartMenuGroup`.

Table 300: Attributes of the StartMenu node

Attribute	Type	Description
`name`	`STRING`	Identifies the StartMenu, can be omitted.
`text`	`STRING`	Defines the text to be displayed as title.

Table 301: Attributes of the StartMenuGroup node

Attribute	Type	Description
`disabled`	`INTEGER`	Indicates if the group must be disabled (grayed, cannot be selected).
`hidden`	`INTEGER`	Indicates if the group is hidden or visible.
`image`	`STRING`	Defines the icon to be used for this group.
`name`	`STRING`	Identifies the start menu group, can be omitted.
`text`	`STRING`	Defines the text to be displayed for this group.

Table 302: Attributes of the StartMenuCommand node

Attribute	Type	Description
`disabled`	`INTEGER`	Indicates if the item must be disabled (grayed, cannot be selected).
`comment`	`STRING`	Specifies the comment to be shown for this command.
`exec`	`STRING`	Defines the command to be executed when the user selects this command.
`hidden`	`INTEGER`	Indicates if the command is hidden or visible.
`image`	`STRING`	Defines the icon to be used for this command.
`name`	`STRING`	Identifies the StartMenu item, can be omitted.
`text`	`STRING`	Defines the text to be displayed for this command.

Attribute	Type	Description
waiting	INTEGER	Defines if the command must be started without waiting (0, default) or waiting (1).

Table 303: Attributes of the StartMenuSeparator node

Attribute	Type	Description
name	STRING	Identifies the StartMenu separator, can be omitted.

Using start menus
To use start menus, you must understand how they work and how to structure the code.
Loading a start menu from an XML file

To load a start menu definition file, use the utility method provided by the `ui.Interface` built-in class:

```
CALL ui.Interface.loadStartMenu("standard")
```

Creating the start menu dynamically

You can create a startmenu dynamically with the `om.DomNode` class:

First, get the abstract user interface root node:

```
DEFINE aui om.DomNode
LET aui = ui.Interface.getRootNode()
```

Next, create a node with the "`StartMenu`" tag name:

```
DEFINE sm om.DomNode
LET sm = aui.createChild("StartMenu")
```

Next, create a "`StartMenuGroup`" node to group a couple of command nodes:

```
DEFINE smg om.DomNode
LET smg = sm.createChild("StartMenuGroup")
CALL smg.setAttribute("text","Programs")
```

Then, create "`StartMenuCommand`" nodes for each program and, if needed, add "`StartMenuSeparator`" nodes to separate entries:

```
DEFINE smc, sms om.DomNode
LET smc = smg.createChild("StartMenuCommand")
CALL smc.setAttribute("text","Orders")
CALL smc.setAttribute("exec","fglrun orders.42r")
LET smc = smg.createChild("StartMenuCommand")
CALL smc.setAttribute("text","Customers")
CALL smc.setAttribute("exec","fglrun customers.42r")
LET sms = smg.createChild("StartMenuSeparator")
LET smc = smg.createChild("StartMenuCommand")
CALL smc.setAttribute("text","Items")
CALL smc.setAttribute("exec","fglrun items.42r")
```

Examples
Example 1: Start menu in XML format

```
<StartMenu>
```

```
   <StartMenuGroup text="Ordering" >
      <StartMenuCommand text="Orders" exec="fglrun orders.42r"
                    disabled="1" />
      <StartMenuCommand text="Customers" exec="fglrun custs.42r"
                    image="smiley" />
      <StartMenuCommand text="Items" exec="fglrun items.42r"
                    waiting="1" />
   <StartMenuCommand text="Reports" exec="fglrun reports.42r"
                comment="Run reports" />
   </StartMenuGroup>
   <StartMenuGroup text="Configuration" >
      <StartMenuCommand text="Database" exec="fglrun dbseconf.42r" />
      <StartMenuCommand text="Users" exec="fglrun userconf.42r" />
      <StartMenuCommand text="Printers" exec="fglrun prntconf.42r" />
   </StartMenuGroup>
</StartMenu>
```

Example 2: Start menu created dynamically

```
MAIN
  DEFINE aui om.DomNode
  DEFINE sm  om.DomNode
  DEFINE smg om.DomNode
  DEFINE smc om.DomNode

  LET aui = ui.Interface.getRootNode()

  LET sm = aui.createChild("StartMenu")

  LET smg = createStartMenuGroup(sm,"Ordering")
  LET smc = createStartMenuCommand(smg,"Orders","fglrun orders.42r",NULL)
  LET smc = createStartMenuCommand(smg,"Customers","fglrun custs.42r",NULL)
  LET smc = createStartMenuCommand(smg,"Items","fglrun items.42r",NULL)
  LET smc = createStartMenuCommand(smg,"Reports","fglrun reports.42r",NULL)
  LET smg = createStartMenuGroup(sm,"Configuration")
  LET smc = createStartMenuCommand(smg,"Database","fglrun
 dbseconf.42r",NULL)
  LET smc = createStartMenuCommand(smg,"Users","fglrun userconf.42r",NULL)
  LET smc = createStartMenuCommand(smg,"Printers","fglrun
 prntconf.42r",NULL)

  MENU "Example"
    COMMAND "Quit"
      EXIT PROGRAM
  END MENU

END MAIN

FUNCTION createStartMenuGroup(p,t)
  DEFINE p om.DomNode
  DEFINE t STRING
  DEFINE s om.DomNode
  LET s = p.createChild("StartMenuGroup")
  CALL s.setAttribute("text",t)
  RETURN s
END FUNCTION

FUNCTION createStartMenuCommand(p,t,c,i)
  DEFINE p om.DomNode
  DEFINE t,c,i STRING
  DEFINE s om.DomNode
  LET s = p.createChild("StartMenuCommand")
  CALL s.setAttribute("text",t)
```

```
  CALL s.setAttribute("exec",c)
  CALL s.setAttribute("image",i)
  RETURN s
END FUNCTION
```

Window containers (WCI)

WCI containers define window containers to group several programs in a parent multiple document interface presentation.

- Understanding the Window Container Interface on page 1458
- Configuration of WCI parent programs on page 1458
- Configuration of WCI child programs on page 1459
- Implement tabbed WCI containers on page 1459

Understanding the Window Container Interface

By default, application windows are displayed independently in separate windows on the front-end window manager. This mode is well known as SDI, "Single Document Interface".

The user interface can be configured to group program windows in a parent container. This is known as MDI, "Multiple Document Interface". In Genero, Multiple Document Interface is called WCI: *Window Container Interface*.

> **Important:** WCI is typically a desktop application feature and is not supported with other front-ends (web and mobile).

The WCI can be used to group several programs together in a parent window. The parent program is the container for the other programs, defined as children of the container. The container program can have its own windows, but this makes sense only for temporary modal windows (with style="dialog").

WCI configuration is done dynamically at the beginning of programs, with methods of the `ui.Interface` built-in class.

Configuration of WCI parent programs

The WCI container program is a separate program of a special type, dedicated to contain other program windows. On the front-end, container programs automatically display a parent window that will hold all child program windows that will attach to the container.

The WCI container program must indicate that its type is special (`ui.Interface.setType()` method), and must identify itself (`ui.Interface.setName()` method):

```
MAIN
  CALL ui.Interface.setName("parent1")
  CALL ui.Interface.setType("container")
  CALL ui.Interface.setText("SoftStore Manager")
  CALL ui.Interface.setSize("600px","1000px")
  CALL ui.Interface.loadStartMenu("mystartmenu")
  MENU "Main"
    COMMAND "Help" CALL help()
    COMMAND "About" CALL aboutbox()
    COMMAND "Exit" EXIT MENU
  END MENU
END MAIN
```

You can define the initial size of the parent container window with the `ui,interface.setSize(height,width)` method.

When the program is identified as a container, a global window is automatically displayed as an container window. The default toolbar and the default topmenu are displayed and a startmenu can be used. Other windows created by this kind of program can be displayed, inside the container (`windowType="normal"`

) or as dialog windows (`windowType="modal"`). Window styles can be applied to the parent window by using the default style specification (`name="Window.main"`).

Configuration of WCI child programs

WCI children programs must attach to a parent container by giving the name of the container program:

```
MAIN
   CALL ui.Interface.setName("custapp")
   CALL ui.Interface.setType("child")
   CALL ui.Interface.setText("Customers")
   CALL ui.Interface.setContainer("parent1")
   ...
END MAIN
```

Multiple container programs can be used to group programs by application modules.

The client displays a system error and the programs stops when:

- A child program is started, but the parent container is not
- A container program is started twice

When the parent container program is stopped, other applications are automatically stopped by the front-end. This will result in a runtime error **-6313** on the application server side. To avoid this, you should control that there are no more running child programs before terminating the parent container program. The WCI container program can query for the existence of children with the `ui.Interface.getChildCount()` and `ui.Interface.getChildInstances()` methods:

```
MAIN
  CALL ui.Interface.setName("parent1")
  CALL ui.Interface.setType("container")
  CALL ui.Interface.setText("SoftStore Manager")
  CALL ui.Interface.setSize("600px","1000px")
  CALL ui.Interface.loadStartMenu("mystartmenu")
  MENU "Main"
    COMMAND "Help" CALL help()
    COMMAND "About" CALL aboutbox()
    COMMAND "Exit"
      IF ui.Interface.getChildCount()>0 THEN
         ERROR "You must first exit the child programs."
      ELSE
         EXIT MENU
      END IF
  END MENU
END MAIN
```

Implement tabbed WCI containers

WCI container can also display the child programs in a folder tab, when the presentation style attribute `tabbedContainer` is set to `yes`.

With a tabbed window container, the style attribute `tabbedContainerCloseMethod` defines how to close the current page.

Values can be:

- `"container"` (default), the container has a close button on the top right corner, which closes the current tab.
- `"page"`, each page has its own close button.
- `"both"`, each page and the container have a close button.
- `"none"`, no close button is shown.

The close button is enabled depending on the window style attribute.

Reports

- Understanding reports on page 1460
- XML output for reports on page 1461
- The report driver on page 1464
- The report routine on page 1469
- Two-pass reports on page 1480
- Report instructions on page 1480
- Report operators on page 1486
- Report aggregate functions on page 1489
- Report engine configuration on page 1492

Understanding reports

A *report* can arrange and format the data according to your instructions and display the output on the screen, send it to a printer, or store it as a file for future use.

To implement a report, a program must include two distinct components:

- The *report driver* specifies what data the report includes.
- The *report routine* formats the data for output.

The report driver retrieves the specified rows from a database, stores their values in program variables, and sends these - one input record at a time - to the report routine. After the last input record is received and formatted, the runtime system calculates any aggregate values based on all the data and sends the entire report to some output device.

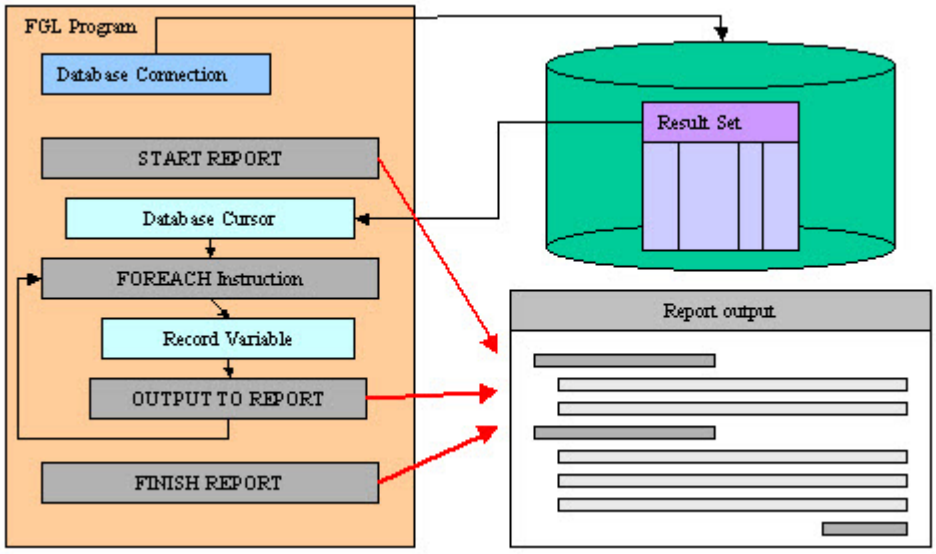


Figure 102: Report driver and database cursor

By separating the two tasks of data retrieval and data formatting, the runtime system simplifies the production of recurrent reports and makes it easy to apply the same report format to different data sets.

The report engine supports the following features:

- The option to display report output to the screen, to the printer, to a file or to a SAX handler to transform the output following XML standards.

- Full control over page layout, including first page header and generic page headers , page trailers, columnar presentation, and row grouping.
- Facilities for creating the report either from the rows returned by a cursor or from input records assembled from any other source, such as output from several different `SELECT` statements through the report driver.
- Control blocks to manipulate data from a database cursor on a row-by-row basis, either before or after the row is formatted by the report.
- Aggregate functions that can calculate frequencies, percentages, sums , averages , minimum, and maximum values.
- The `USING` operator and other built-in functions and operators for formatting and displaying information in output from the report.
- The `WORDWRAP` operator to format long character strings that occupy multiple lines of output from the report.
- The option to execute other language statements while generating a report.
- Stopping a report in the report definition code, with `EXIT REPORT` or `TERMINATE REPORT`.

The report engine supports one-pass reports and two-pass reports. The one-pass requires sorted data to be produced by the report driver in order to handle row grouping with the `BEFORE GROUP` / `AFTER GROUP` blocks. The two-pass record handles sort automatically and does not need sorted data from the report driver. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

XML output for reports

For better integration with external tools based on XML standards, reports can produce XML output.

The purpose of XML-based reports is to sort and group data, not to decorate. Data decoration and formatting can be done by external tools, or you can redirect the XML report output to a SAX document handler object to process the output and generate for example HTML pages.

- Writing an XML report driver and routine on page 1461
- Structure of XML report output on page 1462
- Conditional statement output in XML reports on page 1462

Writing an XML report driver and routine

To produce an XML report, you must start the report with the `TO XML HANDLER` clause in the `START REPORT` instruction, and then use the `PRINTX` statement inside the report routine:

```
MAIN
  ...
  START REPORT orders_report
       TO XML HANDLER om.XmlWriter.createFileWriter("orders.xml")
  ...
END MAIN

REPORT order_report(rec)
  ...
  FORMAT
    ON EVERY ROW
      PRINTX NAME = order rec.*
  ...
END REPORT
```

If all the reports of the program must generate XML output, you can also use the global function `fgl_report_set_document_handler()`.

Structure of XML report output

The generated XML output contains the structure of the formatted pages, with page header, page trailer and group sections. Every `PRINTX` instruction will generate a `<Print>` node with a list of `<Item>` nodes containing the data. The XML processor can use this structure to format and render the output as needed.

If a new report is started with `START REPORT` instruction inside a `REPORT` routine producing XML, and if there is no destination specified in the `START REPORT` instruction, the sub-report inherits the XML output target of the parent, and sub-report nodes will be merged into the parent XML output.

The output of an XML report will have the following node structure:

```
<Report ...>
  <PageHeader pageNo="...">
     ...
  </PageHeader>
  <Group>
    <BeforeGroup>
      <Print name="...">
        <Item name="..." type="..." value="..." isoValue="..." />
        <Item name="..." type="..." value="..." isoValue="..." />
        ...
      </Print>
      ...
    </BeforeGroup>
    <OnEveryRow>
      <Print name="...">
        <Item name="..." type="..." value="..." isoValue="..." />
        <Item name="..." type="..." value="..." isoValue="..." />
        ...
      </Print>
      ...
    </OnEveryRow>
    ...
    <AfterGroup>
      <Print name="...">
        <Item name="..." type="..." value="..." isoValue="..." />
        <Item name="..." type="..." value="..." isoValue="..." />
        ...
      </Print>
    </AfterGroup>
    ...
  </Group>
  ...
  <OnLastRow ...>
     ...
  </OnLastRow>

  <PageTrailer ...>
     ...
  </PageTrailer>

</Report>
```

Conditional statement output in XML reports

If `PRINTX` commands are used inside program flow control instructions like `IF`, `CASE`, `FOR`, `FOREACH` and `WHILE`, the XML output will contain additional nodes to identify such conditional print instructions:

```
<For>
  <ForItem>
    <Print name="...">
```

```
      <Item name="..." type="..." value="..." isValue="..." />
    </Print>
    ...
  </ForItem>
  ...
</For>
```

```
<While>
  <WhileItem>
    <Print name="...">
      <Item name="..." type="..." value="..." isValue="..." />
    </Print>
    ...
  </WhileItem>
  ...
</While>
```

```
<Foreach>
  <ForeachItem>
    <Print name="...">
      <Item name="..." type="..." value="..." isValue="..." />
    </Print>
    ...
  </ForeachItem>
  ...
</Foreach>
```

```
<Case>
  <When id="position">
    <Print name="...">
      <Item name="..." type="..." value="..." isValue="..." />
    </Print>
    ...
  </When>
  ...
</Case>
```

```
<If>
  <IfThen>
    <Print name="...">
      <Item name="..." type="..." value="..." isValue="..." />
    </Print>
    ...
  </IfThen>
  <IfElse>
    <Print name="...">
      <Item name="..." type="..." value="..." isValue="..." />
    </Print>
    ...
  </IfElse>
</If>
```

That information can be useful to process an XML report output.

The report driver

The *report driver* retrieves data, starts the report engine and sends the data (as input records) to be formatted by the REPORT routine.

Usage

A report driver can be part of the MAIN program block, or it can be in one or more functions.

The report driver typically consists of a loop (such as WHILE , FOR, or FOREACH) with the following statements to process the report:

Table 304: Report driver statements

Instruction	Description
START REPORT	This statement is required to instantiate the report driver.
OUTPUT TO REPORT	Provide data for one row to the report driver.
FINISH REPORT	Normal termination of the report.
TERMINATE REPORT	Cancels the processing of the report.

A report driver is started by the START REPORT instruction. Once started, data can be provided to the report driver through the OUTPUT TO REPORT statement. To instruct the report engine to terminate output processing, use the FINISH REPORT instruction. To cancel a report from outside the report routine, use TERMINATE REPORT (from inside the report routine, you cancel the report with EXIT REPORT).

In order to handler report interruption, the report driver can check if the INT_FLAG variable is TRUE to stop the loop when the user asked to interrupt the report execution.

It is possible to execute several report drivers at the same time. It is even possible to invoke a report driver inside a REPORT routine, which is different from the current driver.

The programmer must make sure that the runtime system will always execute these instructions in the following order:

1. START REPORT
2. OUTPUT TO REPORT
3. FINISH REPORT

Example

```
SCHEMA stores7
MAIN
  DEFINE rcust RECORD LIKE customer.*
  DATABASE stores7
  DECLARE cu1 CURSOR FOR SELECT * FROM customer
  LET int_flag = FALSE
  START REPORT myrep
  FOREACH cu1 INTO rcust.*
     IF int_flag THEN EXIT FOREACH END IF
     OUTPUT TO REPORT myrep(rcust.*)
  END FOREACH
  IF int_flag THEN
    TERMINATE REPORT myrep
  ELSE
    FINISH REPORT myrep
  END IF
```

```
END MAIN
```

START REPORT

The START REPORT instruction initializes a report execution.

Syntax

```
START REPORT report-routine
  [ TO to-clause ]
  [ WITH dimension-option [,...]]
```

where *to-clause* is one of:

```
{ SCREEN
| PRINTER
| [FILE] filename
| PIPE program [ IN FORM MODE |  IN LINE MODE ]
| XML HANDLER sax-handler-object
| OUTPUT destination-expr [ DESTINATION { program  | filename} ]
}
```

where *dimension-option* is one of:

```
{ LEFT MARGIN = m-left
| RIGHT MARGIN = m-right
| TOP MARGIN = m-top
| BOTTOM MARGIN = m-bottom
| PAGE LENGTH = m-length
| TOP OF PAGE = c-top
}
```

1. *report-routine* is the name of the REPORT routine.
2. *filename* is a string expression specifying the file that receives report output.
3. *program* is a string expression specifying a program, a shell script, or a command line to receive report output.
4. *destination-expr* is a string expression that specifies one of: SCREEN, PRINTER, FILE, PIPE, PIPE IN LINE MODE, PIPE IN FORM MODE.
5. *sax-handler-object* is a variable referencing an om.SaxDocumentHandler instance.
6. *m-left* is the left margin in number of characters. The default is 5.
7. *m-right* is the right margin in number of characters. The default is 132.
8. *m-top* is the top margin in number of lines. The default is 3.
9. *m-bottom* is the bottom margin in number of lines. The default is 3.
10. *m-length* is the total number of lines on a report page. The default page length is 66 lines.
11. *c-top* is a string that defines the page-eject character sequence.

Usage

The START REPORT statement initializes a report. The instruction allows you to specify the report output destination and the page dimensions and margins.

START REPORT typically precedes a loop instruction such as FOR, FOREACH, or WHILE in which OUTPUT TO REPORT feeds the report routine with data. After the loop terminates, FINISH REPORT completes the processing of the output.

```
DEFINE file_name VARCHAR(200), page_size INTEGER
...
```

```
START REPORT myrep
  TO FILE file_name
  WITH PAGE LENGTH = page_size
```

If a START REPORT statement references a report that is already running, the report is re-initialized; any output might be unpredictable.

Output specification

The TO clause can be used to specify a destination for output. If you omit the TO clause, the Genero runtime system sends report output to the destination specified in the report routine definition. If the report routine does not define an OUTPUT clause, the report output is sent by default to the report viewer when in GUI mode, or to the screen when in TUI mode.

Report output can be specified dynamically as follows:

- The TO FILE option can specify the *filename* as a character variable that is assigned at runtime.
- The TO PIPE option can specify the *program* as a character variable that is assigned at runtime.
- The TO OUTPUT option can specify the report output with a string expression, described later in detail.

The SCREEN option specifies that output is to the report window. The way the report is displayed to the end user depends on whether you are in TUI mode or GUI mode. In TUI mode, the report output displays to the terminal screen. In GUI mode, the report output displays in a dedicated popup window called the Report Viewer.

The PRINTER option instructs the runtime system to output the report to the device or program defined by the DBPRINT environment variable.

When using the FILE option, you can specify a file name as the report destination. Output will be sent to the specified file. If the file exists, its content will be overwritten by the new report output. The FILE keyword is optional, but it's best to include it to make your code more readable.

The PIPE option defines a program, shell script, or command line to which the report output must be sent, using the standard input channel. When using the TUI mode, you can use the IN [LINE|FORM] MODE option to specify whether the program is in line mode or in formatted mode when report output is sent to a pipe.

The TO OUTPUT option allows you to specify one of the output options dynamically at runtime. The character string expression must be one of: "SCREEN", "PRINTER", "FILE", "PIPE", "PIPE IN LINE MODE", " PIPE IN FORM MODE". If the expression specifies "FILE" or "PIPE", you can also specify a *filename* or *program* in a character variable following the DESTINATION keyword.

The XML HANDLER option indicates that the report output will be generated as XML and redirected to a SAX-document handler. When using XML output, the report result can be shown in the Genero Report Engine installed on the front-end workstation. See XML output for more details.

Page dimensions specification

The WITH clause defines the dimensions of each report page and the left, top, right and bottom margins. The values corresponding to a margin and page length must be valid integer expressions. The margins can be defined in any order, but a comma "," is required to separate two page dimensions options.

- The LEFT MARGIN clause defines the number of blank spaces to include at the start of each new line of output. The default is 5.
- The RIGHT MARGIN clause defines the total number of characters in each line of output, including the left margin. If you omit this but specify FORMAT EVERY ROW, the default is 132.
- The TOP MARGIN clause specifies how many blank lines appear above the first line of text on each page of output. The default is 3.
- The BOTTOM MARGIN clause specifies how many blank lines follow the last line of output on each page. The default is 3.

- The PAGE LENGTH clause specifies the total number of lines on each page, including data, the margins, and any page headers or page trailers from the FORMAT section. The default page length is 66 lines.

In addition to the page dimension options, the TOP OF PAGE clause can specify a page-eject sequence for a printer. On some systems, specifying this value can reduce the time required for a large report to produce output, because SKIP TO TOP OF PAGE can substitute this value for multiple line feeds.

OUTPUT TO REPORT

The OUTPUT TO REPORT instruction provides a data row to the report execution.

Syntax

```
OUTPUT TO REPORT report-name ( parameters )
```

1. *report-name* is the name of the report to which the *parameters* should be sent.
2. *parameters* is the data that needs to be sent to the report.

Usage

The OUTPUT TO REPORT instruction feeds the report routine with a single set of data values (called an *input record*), which corresponds usually to one printed line in the report output.

An input record is the ordered set of values returned by the expressions that you list between the parentheses following the report name in the OUTPUT TO REPORT statement. The specified values are passed to the report routine, as part of the input record. The input record typically corresponds to a retrieved row from the database.

The set of values is usually grouped in a RECORD variable and best practice is to define a user defined type (TYPE) in order to ease the variable definitions required in the code implementing the report driver and the report routine definition, for example:

```
SCHEMA stores
TYPE t_cust RECORD LIKE customer.*
...
DEFINE r_cust t_cust
...
  OUTPUT TO REPORT cust_report(r_cust.*)
...
REPORT cust_report(r)
  DEFINE r t_cust
  ...
```

The OUTPUT TO REPORT statement is included within a WHILE, FOR, or FOREACH loop, so that the program passes data to the report one input record at a time. The next example uses a FOREACH loop to fetch data from the database and pass it as input record to a report:

```
SCHEMA stores
DEFINE o LIKE orders.*
...
DECLARE order_c CURSOR FOR
  SELECT orders.*
    FROM orders ORDER BY ord_cust
  START REPORT order_list
  FOREACH order_c INTO o.*
    OUTPUT TO REPORT order_list(o.*)
  END FOREACH
  FINISH REPORT order_list
...
```

Special consideration should be taken regarding row ordering with reports: For example if the report groups rows with `BEFORE GROUP OF` or `AFTER GROUP OF` sections, the rows must be ordered by the column specified in these sections, and rows should preferably be ordered by the report driver to avoid two-pass reports.

If `OUTPUT TO REPORT` is not executed, none of the control blocks of the report routine are executed, even if the program also includes the `START REPORT` and `FINISH REPORT` statements.

The members of the input record that you specify in the expression list of the `OUTPUT TO REPORT` statement must correspond to elements of the formal argument list in the REPORT definition in their number and their position, and must be of compatible data types. At compile time, the number of parameters passed with the `OUTPUT TO REPORT` instruction is not checked against the `DEFINE` section of the report routine. This is a known behavior of the language.

Arguments of the `TEXT` and `BYTE` data types are passed by reference rather than by value; arguments of other data types are passed by value. A report can use the `WORDWRAP` operator with the `PRINT` statement to display `TEXT` values. A report cannot display `BYTE` values; the character string <byte value> in output from the report indicates a `BYTE` value.

FINISH REPORT

The `FINISH REPORT` instruction finalizes a report execution.

Syntax

```
FINISH REPORT report-name
```

1. *report-name* is the name of the report to be ended.

Usage

`FINISH REPORT` closes the report driver. Therefore, it must be the last statement in the report driver and must follow a `START REPORT` statement that specifies the name of the same report.

`FINISH REPORT` must be the last statement in the report driver.

`FINISH REPORT` does the following:

1. Completes the second pass, if report is a two-pass report. These 'second pass' activities handle the calculation and output of any aggregate values that are based on all the input records in the report, such as `COUNT(*)` or `PERCENT(*)` with no `GROUP` qualifier.
2. Executes any `AFTER GROUP OF` control blocks.
3. Executes any `PAGE HEADER`, `ON LAST ROW`, and `PAGE TRAILER` control blocks to complete the report.
4. Copies data from the output buffers of the report to the destination.
5. Closes the Select cursor on any temporary table that was created to order the input records or to perform aggregate calculations.

TERMINATE REPORT

The `TERMINATE REPORT` instruction cancels a report execution.

Syntax

```
TERMINATE REPORT report-name
```

1. *report-name* is the name of the report to be canceled.

Usage

`TERMINATE REPORT` cancels the report processing. It is typically used when the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output, or when the user interrupted the report processing.

`TERMINATE REPORT` has the following effects:

- Terminates the processing of the current report.
- Deletes any intermediate files or temporary tables that were created in processing the report.

The `EXIT REPORT` instruction has the same effect, except that it can be used inside the report definition.

The report routine

The *report routine* implements the body of a report, with formatting instructions.

Syntax

```
[PUBLIC|PRIVATE] REPORT report-name (argument-list)
 [ define-section ]
 [ output-section ]
 [  sort-section ]
 [ format-section ]
END REPORT
```

where *define-section* is:

```
DEFINE variable-definition [,...]
```

where *output-section* is:

```
OUTPUT
[
  REPORT TO
  {
   SCREEN
 | PRINTER
 | [ FILE ] filename
 | PIPE program [ IN FORM MODE |  IN LINE MODE ]
  }
]
[
  [ WITH ]
  [ LEFT MARGIN m-left ]
  [ RIGHT MARGIN m-right]
  [ TOP MARGIN m-top ]
  [ BOTTOM MARGIN m-bottom ]
  [ PAGE LENGTH m-length ]
  [ TOP OF PAGE c-top]
 ]
```

where *sort-section* is:

```
 ORDER [ EXTERNAL ] BY report-variable [,...]
```

where *format-section* is:

```
FORMAT EVERY ROW
```

or:

```
FORMAT
  control-block
  [ report-only-fgl-statement | sql-statement | report-statement ]
  [...]
[...]
```

where *control-block* can be one of:

```
{
[ FIRST ]  PAGE HEADER
|  ON EVERY ROW
|  BEFORE GROUP OF report-variable
|  AFTER GROUP OF report-variable
|  PAGE TRAILER
|  ON LAST ROW
}
```

Note:

1. *variable-definition* follows the `DEFINE` instruction syntax and declares *report-variables* .
2. *report-variable* is the name of a variable declared in the `DEFINE` section.
3. *report-only-fgl-statement* is a subset of all the regular language statements.
4. *sql-statement* is a valid static SQL statement.

Usage

The report definition formats input records. Like the `FUNCTION` or `MAIN` statement, it is a program block that can be the scope of local variables. It is not, however, a function; it is not reentrant, and `CALL` cannot invoke it. The report definition receives data from its driver in sets called input records. These records can include program records, but other data types are also supported. Each input record is formatted and printed as specified by control blocks and statements within the report definition. Most statements and functions can be included in a report definition, and certain specialized statements and operators for formatting output can appear only in a report definition.

Like `MAIN` or `FUNCTION`, the report definition must appear outside any other program block. It must begin with the `REPORT` statement and must end with the `END REPORT` keywords.

Some statements are prohibited in a `REPORT` routine control block. For example, it is not possible to use `CONSTRUCT`, `INPUT`, `DEFER`, `DEFINE`, `REPORT`, `RETURN` instructions in a control block of a report.

By default, report routines are public; They can be called by any other module of the program. If a report routine is only used by the current module, you may want to hide that routine to other modules, to make sure that it will not be called by mistake. To keep a report routine local to the module, add the `PRIVATE` keyword before the report header. Private report routines are only hidden to external modules, all function of the current module can still call local private report routines.

The *define section* declares the data types of local variables used within the report, and of any variables (the input records) that are passed as arguments to the report by the calling statement. Reports without arguments or local variables do not require a `DEFINE` section.

The *output-section* can set margin and page size values, and can also specify where to send the formatted output. Output from the report consists of successive pages, each containing a fixed number of lines whose margins and maximum number of characters are fixed.

The *sort-section* specifies how the rows have to be sorted. The specified sort order determines the order in which the runtime system processes any `GROUP OF` control blocks in the `FORMAT` section.

The *format-section* is required. It specifies the appearance of the report, including page headers, page trailers, and aggregate functions of the data. It can also contain control blocks that specify actions to take

before or after specific groups of rows are processed. (Alternatively, it can produce a default report by only specifying `FORMAT EVERY ROW`).

The report prototype

When defining a report routine, the report name must immediately follow the `REPORT` keyword. The name must be unique among function and report names within the program. Its scope is the entire program.

The list of formal arguments of the report must be enclosed in parentheses and separated by commas. These are local variables that store values that the calling routine passes to the report. The compiler issues an error unless you declare their data types in the subsequent `DEFINE` section of the report routine. You can include a program record in the formal argument list, but you cannot append the `.*` symbols to the name of the record. Arguments can be of any data type except ARRAY, or a record with an ARRAY member.

When you call a report, the formal arguments are assigned values from the argument list of the `OUTPUT TO REPORT` statement. These actual arguments that you pass must match, in number and position, the formal arguments of the `REPORT` routine. The data types must be compatible, but they need not be identical. The runtime system can perform some conversions between compatible data types.

The names of the actual arguments and the formal arguments do not have to match.

You must include the following items in the list of formal arguments:

- All the values for each row sent to the report in the following cases:
  - If you include an `ORDER BY` section or `GROUP PERCENT(*)` function
  - If you use a global aggregate function (one over all rows of the report) anywhere in the report, except in the `ON LAST ROW` control block
  - If you specify the `FORMAT EVERY ROW` default format
- Any variables referenced in the following group control blocks:
  - `AFTER GROUP OF`
  - `BEFORE GROUP OF`

DEFINE section in REPORT

Defines report parameters and local variables.

Syntax

The syntax of the report `DEFINE` section is the same as for the `DEFINE` statement, except that you cannot define arrays or array members for records.

Usage

This section declares a data type for each formal argument in the `REPORT` prototype and for any additional local variables that can be referenced only within the `REPORT` program block. The `DEFINE` section is required if you pass arguments to the report or if you reference local variables in the report.

For declaring local report variables, the same rules apply to the `DEFINE` section as to the `DEFINE` statement in `MAIN` and `FUNCTION` program blocks. There are some differences and exceptions, however:

- Report arguments cannot be of type `ARRAY`.
- Report arguments cannot be records that include `ARRAY` members.
- Report local variables are not allocated on the stack at every `OUTPUT TO REPORT` call: The scope of the variables in the `DEFINE` section is local to the report routine, but the lifetime is the duration of the program, like module or global variables. In other words, report variables persist across `OUTPUT TO REPORT` calls.

data types of local variables that are not formal arguments are unrestricted. You must include arguments in the report prototype and declare them in the `DEFINE` section, if any of the following conditions is true:

- If you specify `FORMAT EVERY ROW` to create a default report, you must pass all the values for each record of the report.
- If an `ORDER BY` section is included, you must pass all the values that `ORDER BY` references for each input record of the report.
- If you use the `AFTER GROUP OF` control block, you must pass at least the arguments that are named in that control block.
- If an aggregate that depends on all records of the report appears anywhere except in the `ON LAST ROW` control block, you must pass each of the records of the report through the argument list.

Aggregates dependent on all records include:

- `GROUP PERCENT(*)` (anywhere in a report).
- Any aggregate without the `GROUP` keyword (anywhere outside the `ON LAST ROW` control block).

If your report calls an aggregate function, an error might result if any argument of an aggregate function is not also a format argument of the report. You can, however, use global or module variables as arguments of aggregates if the value of the variable does not change while the report is executing.

A report can reference variables of global or module scope that are not declared in the `DEFINE` section. Their values can be printed, but they can cause problems in aggregates and in `BEFORE GROUP OF` and `AFTER GROUP OF` clauses. Any references to non-local variables can produce unexpected results, however, if their values change while a two-pass report is executing.

OUTPUT section in REPORT

Specifies report destination and page format options.

Syntax

```
 OUTPUT
[
  REPORT TO
  {
      SCREEN
    | PRINTER
    | [ FILE ] filename
    | PIPE [ IN FORM MODE | IN LINE MODE ] program
   }
]
[
  [ LEFT MARGIN m-left ]
  [ RIGHT MARGIN m-right ]
  [ TOP MARGIN m-top ]
  [ BOTTOM MARGIN m-bottom ]
  [ PAGE LENGTH m-length ]
  [ TOP OF PAGE c-top]
]
```

1. *program* defines the name of a program, shell script, command receiving the output.
2. *filename* defines the file which receives the output of the report.
3. *m-left* is the left margin in number of characters. The default is 5.
4. *m-right* is the right margin in number of characters. The default is 132.
5. *m-top* is the top margin in number of lines. The default is 3.
6. *m-bottom* is the bottom margin in number of lines. The default is 3.
7. *m-length* is the total number of lines on a report page. The default page length is 66 lines.
8. *c-top* is a string that defines the page-eject character sequence.

Usage

The OUTPUT section can specify the destination and dimensions for output from the report and the page-eject sequence for the printer. If you omit the OUTPUT section, the report uses default values to format each page. This section is superseded by any corresponding START REPORT specifications.

The OUTPUT section can direct the output from the report to a printer, file, or pipe, and can initialize the page dimensions and margins of report output. If PRINTER is specified, the DBPRINT environment variable specifies which printer.

The START REPORT statement of the report driver can override all of these specifications by assigning another destination in its TO clause or by assigning other dimensions, margins, or another page-eject sequence in the WITH clause.

Because the size specifications for the dimensions and margins of a page of report output that the OUTPUT section can specify must be literal integers, consider defining page dimensions in the START REPORT statement, where you can use variables to assign these values dynamically at runtime.

ORDER BY section in REPORT

Forces a sort order of unsorted data rows in reports.

Syntax

```
ORDER [ EXTERNAL ] BY report-variable [ DESC | ASC ] [,...]
```

1. *report-variable* identifies one of the variables passed to the report routine to be used for sorting rows.

Usage

When grouping rows in a report, values that the report definition receives from the report driver are significant in determining how BEFORE GROUP OF or AFTER GROUP OF control blocks will process the data in the formatted report output.

The ORDER BY section defines how the variables of the input records are to be sorted. It is required if the report driver does not send sorted data to the report. The specified sort order determines the order in which the runtime system processes any GROUP OF control blocks in the FORMAT section.

If you omit the ORDER BY section, the runtime system processes input records in the order received from the report driver and processes any GROUP OF control blocks in their order of appearance in the FORMAT section. If records are not sorted in the report driver , the GROUP OF control blocks might be executed at random intervals (that is, after any input record) because unsorted values tend to change from record to record.

If you specify only one variable in the GROUP OF control blocks, and the input records are already sorted in sequence on that variable by the SELECT statement, you do not need to include an ORDER BY section in the report.

Specify ORDER EXTERNAL BY if the input records have already been sorted by the SELECT statement used by the report driver. The list of variables after the keywords ORDER EXTERNAL BY control the execution order of GROUP BY control blocks.

Without the EXTERNAL keyword, the report becomes a two-pass report , meaning that the report engine processes the set of input records twice. During the first pass, the report engine sorts the data and stores the sorted values in a temporary table in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

With the EXTERNAL keyword, the report engine only needs to make a single pass through the data: it does not need to build the temporary table in the database for sorting the data. However, If the report routine contains aggregations functions such as GROUP PERCENT(*) , the report will become a two-pass report because such aggregation function needs all rows to compute the value.

The DESC or ASC clause defines the sort order.

FORMAT section in REPORT

Defines the formatting directives inside a report routine.

Syntax

Default format:

```
FORMAT EVERY ROW
```

Custom format:

```
FORMAT
 control-block
    [ report-statement
    | report-only-fgl-statement
    | sql-statement
    ]
  [...]
 [...]
```

where *control-block* can be one of:

```
{
[ FIRST ] PAGE HEADER
|  ON EVERY ROW
|  BEFORE GROUP OF report-variable
|  AFTER GROUP OF report-variable
|  PAGE TRAILER
|  ON LAST ROW
}
```

1. *report-statement* is any report-specific instruction.
2. *report-only-fgl-statement* is any language instruction supported in the report routine.
3. *sql-statement* is any SQL statement supported by the language.
4. *report-variable* is the name of a variable declared in the DEFINE section.

Usage

A report definition must contain a FORMAT section.

The FORMAT section determines how the output from the report will look. It works with the values that are passed to the REPORT program block through the argument list or with global or module variables in each record of the report. In a source file, the FORMAT section begins with the FORMAT keyword and ends with the END REPORT keywords.

The FORMAT section is made up of the following control blocks:

- FIRST PAGE HEADER
- PAGE HEADER
- PAGE TRAILER
- BEFORE GROUP OF
- AFTER GROUP OF
- ON EVERY ROW
- ON LAST ROW

If you use the FORMAT EVERY ROW, no other statements or control blocks are valid. The EVERY ROW keywords specify a default output format, including every input record that is passed to the report.

Control blocks define the structure of a report by specifying one or more statements to be executed when specific parts of the report are processed.

If a report driver includes START REPORT and FINISH REPORT statements, but no data records are passed to the report, no control blocks are executed. That is, unless the report executes an OUTPUT TO REPORT statement that passes at least one input record to the report; then neither the FIRST PAGE HEADER control block nor any other control block is executed

Apart from BEFORE GROUP OF and AFTER GROUP OF, each control block must appear only one time.

More complex FORMAT sections can contain control blocks like ON EVERY ROW or BEFORE GROUP OF, which contain statements to execute while the report is being processed. Control blocks can contain report execution statements and other executable statements.

A control block may invoke most language statements, except those listed in prohibited statements.

The BEFORE/AFTER GROUP OF control blocks can include aggregate functions to instruct the report engine to automatically compute such values.

A *report-statement* is a statement specially designed for the report format section. It cannot be used in any other part of the program.

The sequence in which the BEFORE GROUP OF and AFTER GROUP OF control blocks are executed depends on the sort list in the ORDER BY section, regardless of the physical sequence in which these control blocks appear within the FORMAT section.

FORMAT EVERY ROW
Default format specification of a report.

A report routine written with FORMAT EVERY ROW formats the report in a simple default format, containing only the values that are passed to the REPORT program block through its arguments, and the names of the arguments. You cannot modify the EVERY ROW statement with any of the statements listed in report execution statements, and neither can you include any control blocks in the FORMAT section.

The report engine uses as column headings the names of the variables that the report driver passes as arguments at runtime. If all fields of each input record can fit horizontally on a single line, the default report prints the names across the top of each page and the values beneath. Otherwise, it formats the report with the names down the left side of the page and the values to the right, as in the previous example. When a variable contains a null value, the default report prints only the name of the variable, with nothing for the value.

The following example is a brief report specification that uses FORMAT EVERY ROW. We assume here that the cursor that retrieved the input records for this report was declared with an ORDER BY clause, so that no ORDER BY section is needed in this report definition:

```
DATABASE stores7

REPORT simple( order_num, customer_num, order_date )

  DEFINE order_num LIKE orders.order_num,
        customer_num LIKE orders.customer_num,
        order_date LIKE orders.order_date

  FORMAT EVERY ROW

END REPORT
```

The example would produce the following output:

```
order_num customer_num order_date
     1001         104  01/20/1993
     1002         101  06/01/1993
     1003         104  10/12/1993
```

```
      1004          106  04/12/1993
      1005          116  12/04/1993
      1006          112  09/19/1993
      1007          117  03/25/1993
      1008          110  11/17/1993
      1009          111  02/14/1993
      1010          115  05/29/1993
      1011          104  03/23/1993
      1012          117  06/05/1993
```

FIRST PAGE HEADER
Defines the printing commands for the first page of a report.

This control block specifies the action that the runtime system takes before it begins processing the first input record. You can use it, for example, to specify what appears near the top of the first page of output from the report.

Because the runtime system executes the `FIRST PAGE HEADER` control block before generating any output, you can use this control block to initialize variables that you use in the `FORMAT` section.

If a report driver includes `START REPORT` and `FINISH REPORT` statements, but no data records are passed to the report, this control block is not executed. That is, unless the report executes an `OUTPUT TO REPORT` statement that passes at least one input record to the report, neither the `FIRST PAGE HEADER` control block nor any other control block is executed.

As its name implies, you can also use a `FIRST PAGE HEADER` control block to produce a title page as well as column headings. On the first page of a report, this control block overrides any `PAGE HEADER` control block. That is, if both a `FIRST PAGE HEADER` and a `PAGE HEADER` control block exist, output from the first appears at the beginning of the first page, and output from the second begins all subsequent pages.

The `TOP MARGIN` (set in the `OUTPUT` section) determines how close the header appears to the top of the page.

Consider the following notes when programming the `FIRST PAGE HEADER` control block:

1. You cannot include a `SKIP` integer `LINES` statement inside a loop within this control block.
2. The `NEED` statement is not valid within this control block.
3. If you use an `IF…THEN…ELSE` statement within this control block, the number of lines displayed by any `PRINT` statements following the `THEN` keyword must be equal to the number of lines displayed by any `PRINT` statements following the `ELSE` keyword.
4. If you use a `CASE`, `FOR`, or `WHILE` statement that contains a `PRINT` statement within this control block, you must terminate the `PRINT` statement with a semicolon (;). The semicolon suppresses any `LINEFEED` characters in the loop, keeping the number of lines in the header constant from page to page.
5. You cannot use a `PRINT` filename statement to read and display text from a file within this control block

Corresponding restrictions also apply to `CASE`, `FOR`, `IF`, `NEED`, `SKIP`, `PRINT`, and `WHILE` statements in `PAGE HEADER` and `PAGE TRAILER` control blocks.

PAGE HEADER
Defines the printing commands for the top of all pages of a report.

This control block is executed whenever a new page is added to the report. The `PAGE HEADER` control block specifies the action that the runtime takes before it begins processing each page of the report. It can specify what information, if any, appears at the top of each new page of output from the report.

The `TOP MARGIN` specification (in the `OUTPUT` section) affects how many blank lines appear above the output produced by statements in the `PAGE HEADER` control block.

You can use the `PAGENO` operator in a `PRINT` statement within a `PAGE HEADER` control block to automatically display the current page number at the top of every page.

The FIRST PAGE HEADER control block overrides this control block on the first page of a report.

New group values can appear in the PAGE HEADER control block when this control block is executed after a simultaneous end-of-group and end-of-page situation.

The runtime system delays the processing of the PAGE HEADER control block until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW, BEFORE GROUP OF, or AFTER GROUP OF control block. This order guarantees that any group columns printed in the PAGE HEADER control block have the same values as the columns printed in the ON EVERY ROW control block.

The details that apply to FIRST PAGE HEADER also apply to PAGE HEADER.

PAGE TRAILER
Defines the printing commands for the tail of all pages of a report.

The PAGE TRAILER control block specifies what information, if any, appears at the bottom of each page of output from the report.

The runtime system executes the statements in the PAGE TRAILER control block before the PAGE HEADER control block when a new page is needed. New pages can be initiated by any of the following conditions:

- PRINT attempts to print on a page that is already full.
- SKIP TO TOP OF PAGE is executed.
- SKIP n LINES specifies more lines than are available on the current page.
- NEED specifies more lines than are available on the current page.

You can use the PAGENO operator in a PRINT statement within a PAGE TRAILER control block to automatically display the page number at the bottom of every page, as in this example:

```
PAGE TRAILER
   PRINT COLUMN 28, PAGENO USING "page <<<<"
```

The BOTTOM MARGIN specification (in the OUTPUT section) affects how close to the bottom of the page the output displays the page trailer.

The details that apply to FIRST PAGE HEADER also apply to PAGE TRAILER.

BEFORE/AFTER GROUP OF
Defines printing commands of row grouping sections withing a report.

The BEFORE/AFTER GROUP OF control blocks specify what action the runtime system takes respectively before or after it processes a group of input records. Group hierarchy is determined by the ORDER BY specification in the SELECT statement or in the report definition.

A group of records is all of the input records that contain the same value for the variable whose name follows the AFTER GROUP OF keywords. This group variable must be passed through the report arguments. A report can include no more than one AFTER GROUP OF control block for any group variable.

When the runtime system executes the statements in a BEFORE/AFTER GROUP OF control block, the report variables have the values from the first / last record of the new group. From this perspective, the BEFORE/AFTER GROUP OF control block could be thought of as the "on first / last record of group" control block.

Each BEFORE GROUP OF block is executed in order, from highest to lowest priority, at the start of a report (after any FIRST PAGE HEADER or PAGE HEADER control blocks, but before processing the first record) and on these occasions:

- Whenever the value of the group variable changes (after any AFTER GROUP OF block for the old value completes execution)
- Whenever the value of a higher-priority variable in the sort list changes (after any AFTER GROUP OF block for the old value completes execution)

The runtime system executes the AFTER GROUP OF control block on these occasions:

- Whenever the value of the group variable changes.
- Whenever the value of a higher-priority variable in the sort list changes.
- At the end of the report (after processing the last input record but before the runtime system executes any ON LAST ROW or PAGE TRAILER control blocks). In this case, each AFTER GROUP OF control block is executed in ascending priority.

How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If records are already sorted, the BEFORE/AFTER GROUP OF block executes before the runtime system processes the first record of the group.
- If records are not sorted, the BEFORE GROUP OF block might be executed after any record because the value of the group variable can change with each record. If no ORDER BY section is specified, all BEFORE/AFTER GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section. The BEFORE/AFTER GROUP OF control blocks are designed to work with sorted data.

You can sort the records by specifying a sort list in either of the following areas:

- An ORDER BY section in the report definition
- The ORDER BY clause of the SELECT statement in the report driver

To sort data in the report definition (with an ORDER BY section), make sure that the name of the group variable appears in both the ORDER BY section and in the BEFORE GROUP OF control block.

To sort data in the ORDER BY clause of a SELECT statement, perform the following tasks:

- Use the column name in the ORDER BY clause of the SELECT statement as the group variable in the BEFORE GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure that you include an ORDER EXTERNAL BY section in the report to specify the precedence of variables in the sort list.

If you specify sort lists in both the report driver and the report definition, the sort list in the ORDER BY section of the REPORT takes precedence. When the runtime system starts to generate a report, it first executes the BEFORE GROUP OF control blocks in descending order of priority before it executes the ON EVERY ROW control block. If the report is not already at the top of the page, the SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block causes the output for each group to start at the top of a page.

If the sort list includes more than one variable, the runtime system sorts the records by values in the first variable (highest priority). Records that have the same value for the first variable are then ordered by the second variable and so on until records that have the same values for all other variables are ordered by the last variable (lowest priority) in the sort list.

The ORDER BY section determines the order in which the runtime system processes BEFORE GROUP OF and AFTER GROUP OF control blocks. If you omit the ORDER BY section, the runtime system processes any GROUP OF control blocks in the lexical order of their appearance within the FORMAT section.

If you include an ORDER BY section, and the FORMAT section contains more than one BEFORE GROUP OF or AFTER GROUP OF control block, the order in which these control blocks are executed is determined by the sort list in the ORDER BY section. In this case, their order within the FORMAT section is not significant because the sort list overrides their lexical order.

The runtime system processes all the statements in a BEFORE GROUP OF or AFTER GROUP OF control block on these occasions:

- Each time the value of the current group variable changes.
- Each time the value of a higher-priority variable changes. How often the value of the group variable changes depends in part on whether the input records have been sorted. If the records are sorted, AFTER GROUP OF executes after the runtime system processes the last record of the group of records; BEFORE GROUP OF executes before the runtime system processes the first records with the same

value for the group variable. If the records are not sorted, the BEFORE GROUP OF and AFTER GROUP OF control blocks might be executed before and after each record because the value of the group variable might change with each record. All the AFTER GROUP OF and BEFORE GROUP OF control blocks are executed in the same lexical order in which they appear in the FORMAT section.

In the AFTER GROUP OF control block, you can include the GROUP keyword to qualify aggregate report functions like AVG(), SUM(), MIN(), or MAX():

```
AFTER GROUP OF r.order_num
  PRINT r.order_date, 7 SPACES,
     r.order_num USING"###&", 8 SPACES,
       r.ship_date, " ",
       GROUP SUM(r.total_price) USING"$$$$,$$$,$$$.&&"
AFTER GROUP OF r.customer_num
  PRINT 42 SPACES, "------------------"
  PRINT 42 SPACES, GROUP SUM(r.total_price) USING"$$$$,$$$,$$$.&&"
```

Using the GROUP keyword to qualify an aggregate function is only valid within the AFTER GROUP OF control block. It is not valid, for example, in the BEFORE GROUP OF control block.

After the last input record is processed, the runtime system executes the AFTER GROUP OF control blocks before it executes the ON LAST ROW control block.

ON EVERY ROW
Defines printing commands for each row in a report.

The ON EVERY ROW control block specifies the action to be taken by the runtime system for every input record that is passed to the report definition.

The runtime system executes the statements within the ON EVERY ROW control block for each new input record that is passed to the report. The following example is from a report that lists all the customers, their addresses, and their telephone numbers across the page:

```
ON EVERY ROW
  PRINT r.fname, " ", r.lname, " ",
       r.address1, " ", r.cust_phone
```

The runtime system delays processing the PAGE HEADER control block (or the FIRST PAGE HEADER control block, if it exists) until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW control block.

If a BEFORE GROUP OF control block is triggered by a change in the value of a variable, the runtime system executes all appropriate BEFORE GROUP OF control blocks (in the order of their priority) before it executes the ON EVERY ROW control block. Similarly, if execution of an AFTER GROUP OF control block is triggered by a change in the value of a variable, the runtime system executes all appropriate AFTER GROUP OF control blocks (in the reverse order of their priority) before it executes the ON EVERY ROW control block.

ON LAST ROW
Defines the printing commands of the last row in a report.

The ON LAST ROW control block specifies the action that the runtime system is to take after it processes the last input record that was passed to the report definition and encounters the FINISH REPORT statement.

The statements in the ON LAST ROW control block are executed after the statements in the ON EVERY ROW and AFTER GROUP OF control blocks if these blocks are present.

When the runtime system processes the statements in an ON LAST ROW control block, the variables that the report is processing still have the values from the final record that the report processed. The ON LAST ROW control block can use aggregate functions to display report totals.

Prohibited report routine statements

Language statements that have no meaning inside a report definition routine are prohibited. These statements are some of the statements that are not valid within any control block of the FORMAT section of a REPORT program block, such as interactive statements (CONSTRUCT, INPUT, DIALOG, MENU), program block definitions (FUNCTION, REPORT), and some flow control instructions like RETURN.

A compile-time error is issued if you attempt to include any of these statements in a control block of a report. You can call a function that includes some of these statements, but this is not recommended.

Two-pass reports

The report engine supports *two-pass reports*, to order rows automatically.

The one-pass report requires sorted data to be produced by the report driver in order to handle before/after groups properly. The two-pass report handles sorts internally and does not need sorted data from the report driver. During the first pass, the report engine sorts the data and stores the sorted values in a temporary file in the database. During the second pass, it calculates any aggregate values and produces output from data in the temporary files.

A report is defined as a two-pass report if it includes any of the following items:

- An ORDER BY section without the EXTERNAL keyword.
- The GROUP PERCENT(*) aggregate function anywhere in the report.
- Any aggregate function that has no GROUP keyword in any control block other than ON LAST ROW.

Two-pass reports create temporary tables. The FINISH REPORT statement uses values from these tables to calculate any global aggregates, and then deletes the tables. Since two-pass reports create temporary tables, the report engine requires a database connection, and the database server must support temporary tables with indexes.

Consider avoiding two-pass reports when a regular report is possible.

Report instructions

The report instruction listed in this section can appear only in control blocks of the FORMAT section of a report routine.

EXIT REPORT

Cancels the report processing.

Syntax

```
EXIT REPORT
```

Usage

EXIT REPORT cancels the report processing. It must appear in the FORMAT section of the report definition. It is useful after the program (or the user) becomes aware that a problem prevents the report from producing part of its intended output.

EXIT REPORT has the following effects:

- Terminates the processing of the current report.
- Deletes any intermediate files or temporary tables that were created in processing the report.

You cannot use the RETURN statement as a substitute for EXIT REPORT. An error is issued if RETURN is encountered within the definition of a report.

PRINT

Formats and prints a row of data in a report routine.

Syntax

```
PRINT
 {
    expression
  | COLUMN left-offset
  | PAGENO
  | LINENO
  | num-spaces SPACES
  | [GROUP] COUNT(*) [ WHERE condition ]
  | [GROUP] PERCENT(*) [ WHERE condition]
  | [GROUP] AVG( variable ) [ WHERE condition]
  | [GROUP] SUM( variable ) [ WHERE condition]
  | [GROUP] MIN( variable ) [ WHERE condition]
  | [GROUP] MAX( variable ) [ WHERE condition]
  | char-expression WORDWRAP [ RIGHT MARGIN rm ]
  | FILE "file-name"
 } [,...]
  [ ; ]
```

1. *expression* is any legal language expression.
2. *left-offset* is described in COLUMN.
3. *num-spaces* is described in SPACES.
4. *char-expression* is a string expression or a TEXT variable.
5. *filename* is a string expression , or a quoted string, that specifies the name of a text file to include in the output from the report.

Usage

The PRINT instruction is used in a report routine to output a line of data.

The PRINT statement can include character data in the form of an ASCII file, a TEXT variable, or a comma-separated expression list of character expressions in the output of the report. (For TEXT variable or filename, you cannot specify additional output in the same PRINT statement.)

If a BYTE value is used in the PRINT statement, the output will show the "<byte value>" text for this element when the report output is regular text. If the report output is XML, the BYTE value is converted to Base64 before it is written to the output stream.

PRINT statement output begins at the current character position, sometimes called simply the current position. On each page of a report, the initial default character position is the first character position in the first line. This position can be offset horizontally and vertically by margin and header specifications and by executing any of the following statements:

- The SKIP statement moves it down to the left margin of a new line.
- The NEED statement can conditionally move it to a new page.
- The PRINT statement moves it horizontally (and sometimes down).

Unless you use the keyword CLIPPED or USING, values are displayed with widths (including any sign) that depend on their declared data types.

Table 305: Default print width for data types

Data type	Default Print Width
BYTE	N/A

Data type	Default Print Width
CHAR	Length of character data type declaration.
DATE	DBDATE dependent, 10 if DBDATE = "MDY4/"
DATETIME	From 2 to 25, as implied in the data type declaration.
DECIMAL	(2 + p + s), where p is the precision and s is the scale from the data type declaration.
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25, as implied in the data type declaration.
MONEY	(2 + c + p + s), where c is the length of the currency defined by DBMONEY and p is the precision and s is the scale from the data type declaration.
NCHAR	Length of character data type declaration.
NVARCHAR	Length current value in the variable.
SMALLFLOAT	14
SMALLINT	6
STRING	Length current value in the variable.
TEXT	Length current value in the variable.
VARCHAR	Length current value in the variable.

Unless you specify the FILE or WORDWRAP option, each PRINT statement displays output on a single line. For example, this fragment displays output on two lines:

```
PRINT fname, lname
PRINT city, ", ", state, " ", zip-code
```

If you terminate a PRINT statement with a semicolon, however, you suppress the implicit LINEFEED character at the end of the line. The next example has the same effect as the PRINT statements in the previous example:

```
PRINT fname;
PRINT lname
PRINT city, ", ", state, " ", zip-code
```

The expression list of a PRINT statement returns one or more values that can be displayed as printable characters. The expression list can contain report variables , built-in functions and operators. Some of these can appear only in a REPORT program block such as PAGENO, LINENO, PERCENT.

If the expression list applies the USING operator to format a DATE or MONEY value, the format string of the USING operator takes precedence over the DBDATE , DBMONEY , and DBFORMAT environment variables.

The PRINT FILE statement reads the contents of the specified filename into the report, beginning at the current character position. This statement permits you to insert a multiple-line character string into the output of a report. If *filename* stores the value of a TEXT variable, the PRINT FILE *file-name*

statement has the same effect as specifying PRINT *text-variable*. (But only PRINT variable can include the WORDWRAP operator)

Aggregate report functions summarize data from several records in a report. The syntax and effects of aggregates in a report resemble those of SQL aggregate functions but are not identical.

The expression (in parentheses) that SUM(), AVG(), MIN(), or MAX() takes as an argument is typically of a number or INTERVAL data type; ARRAY, BYTE, RECORD, and TEXT are not valid. The SUM(), AVG(), MIN() , and MAX() aggregates ignore input records for which their arguments have null values, but each returns NULL if every record has a null value for the argument.

The GROUP keyword is an optional keyword that causes the aggregate function to include data only for a group of records that have the same value for a variable that you specify in an AFTER GROUP OF control block. An aggregate function can only include the GROUP keyword within an AFTER GROUP OF control block.

The optional WHERE clause allows you to select among records passed to the report, so that only records for which the boolean expression is TRUE are included.

Example

The following example is from the FORMAT section of a report definition that displays both quoted strings and values from rows of the customer table:

```
FIRST PAGE HEADER
  PRINT COLUMN 30, "CUSTOMER LIST"
  SKIP 2 LINES
  PRINT "Listings for the State of ", thisstate
SKIP 2 LINES
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
        COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE
PAGE HEADER
  PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
        COLUMN 57, "ZIP", COLUMN 65, "PHONE"
  SKIP 1 LINE
ON EVERY ROW
  PRINT customer_num USING "###&", COLUMN 12, fname CLIPPED,
       1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED, ", ",
       state, COLUMN 57, zip-code, COLUMN 65, phone
```

PRINTX

Prints an XML formatted row of data in a report, with an additional identifier for XML outputs.

Syntax

```
PRINTX [NAME = identifier] expression
```

1. *identifier* is the name to be used in the XML node.
2. *expression* is any legal language expression.

Usage

The PRINTX statement is similar to PRINT, except that when XML is produced by the report, the XML print element will be named as specified. If the NAME clause is omitted or the report is run in non-XML mode, then PRINTX does exactly the same as PRINT.

To generate XML output, you must redirect the report output into a SAX document handler with the `TO XML HANDLER` clause of `START REPORT`:

```
START REPORT orders_report
   TO XML HANDLER om.XmlWriter.createFileWriter("orders.xml")
```

Note that when using XML output, `BYTE` values are converted to Base64 before they are printed with the `PRINTX` instruction.

Example

```
REPORT (fname, lname, ...)
  DEFINE fname VARCHAR(20),
         lname VARCHAR(20)
  ...
  FORMAT
    ...
    ON EVERY ROW
       PRINTX NAME=customer fname, lname
...
```

With the above code, the variable names will appear in the graphical report designer as "`customer.fname`" and "`customer.lname`".

NEED

Specifies the number of rows needed in a report section.

Syntax

```
NEED num-lines LINE[S]
```

1. *num-lines* is the number of lines.

Usage

This statement has the effect of a conditional `SKIP TO TOP OF PAGE` statement, the condition being that the number to which the integer expression evaluates is greater than the number of lines that remain on the current page.

The `NEED` statement can prevent the report from dividing parts of the output that you want to keep together on a single page. In this example, the `NEED` statement causes the `PRINT` statement to send output to the next page unless at least six lines remain on the current page:

```
  AFTER GROUP OF r.order_num
    NEED 6 LINES
    PRINT " ", r.order_date, "  ", GROUP SUM(r.total_price)
```

The `LINES` value specifies how many lines must remain between the line above the current character position and the bottom margin for the next `PRINT` statement to produce output on the current page. If fewer than `LINES` remain on the page, the report engine prints both the `PAGE TRAILER` and the `PAGE HEADER`.

The `NEED` statement does not include the `BOTTOM MARGIN` value when it compares `LINES` to the number of lines remaining on the current page. `NEED` is not valid in `FIRST PAGE HEADER`, `PAGE HEADER`, or `PAGE TRAILER` blocks.

PAUSE

Pauses a report displayed to the screen.

Syntax

```
PAUSE [ "comment" ]
```

1. *comment* is an optional comment to be displayed.

Usage

Output is sent by default to the screen unless the START REPORT statement or the OUTPUT section specifies a destination for report output.

The PAUSE statement can be executed only if the report sends its output to the screen. It has no effect if you include a TO clause in either of these contexts:

- In the OUTPUT section of the report definition.
- In the START REPORT statement of the report driver.

Include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. For example, the following code causes the runtime system to skip a line and pause at the end of each page of report output displayed on the screen:

```
  PAGE TRAILER
    SKIP 1 LINE
    PAUSE "Press return to continue"
```

SKIP

Skips a given number of lines in a report.

Syntax

```
SKIP { num-lines LINE[S] | TO TOP OF PAGE }
```

1. *num-lines* is the number of lines.

Usage

The SKIP statement allows you to insert blank lines into report output or to skip to the top of the next page as if you had included an equivalent number of PRINT statements without specifying any expression list.

The LINE and LINES keywords are synonyms in the SKIP statement.

Output from any PAGE HEADER or PAGE TRAILER control block appears in its usual location.

The SKIP n LINES statement cannot appear within a CASE statement, a FOR loop, or a WHILE loop.

The SKIP TO TOP OF PAGE statement cannot appear in a FIRST PAGE HEADER , PAGE HEADER or PAGE TRAILER control block.

Example

```
  FIRST PAGE HEADER
    PRINT "Customer List"
    SKIP 2 LINES
    PRINT "Number        Name               Location"
    SKIP 1 LINE
  PAGE HEADER
    PRINT "Number        Name               Location"
```

```
    SKIP 1 LINE
  ON EVERY ROW
    PRINT r.customer_num, r.fname, r.city
```

Report operators

Report operators can be used to print dynamic report information.

LINENO

Contains the current line number in a report.

Syntax

```
LINENO
```

Usage

This operator takes no operand but returns the value of the line number of the report line that is currently printing.

The report engine calculates the line number by calculating the number of lines from the top of the current page, including the `TOP MARGIN`.

Example

In this example, a `PRINT` statement instructs the report to calculate and display the current line number, beginning in the tenth character position after the left margin:

```
  ON EVERY ROW
    IF LINENO > 9 THEN
      PRINT COLUMN 10, "Line:", LINENO USING "<<<"
    END IF
```

PAGENO

Contains the current page number in a report.

Syntax

```
PAGENO
```

Usage

This operator takes no operand but returns the number of the page the report engine is currently printing.

You can use `PAGENO` in the `PAGE HEADER` or `PAGE TRAILER` block, or in other control blocks to number the pages of a report sequentially.

Example

If you use the SQL aggregate `COUNT(*)` in the `SELECT` statement to find how many records are returned by the query, and if the number of records that appear on each page

of output is both fixed and known, you can calculate the total number of pages, as in this example:

```
  FIRST PAGE HEADER
    SELECT COUNT(*) INTO cnt FROM customer
    LET y = cnt/50 -- Assumes 50 records per page
  ON EVERY ROW
    PRINT COLUMN 10, r.customer_num, ...
  PAGE TRAILER
    PRINT PAGE PAGENO USING "<<" OF cnt USING "<<"
```

If the calculated number of pages was 20, the first page trailer would be:

```
Page 1 of 20
```

PAGENO increments with each page, so the last page trailer would be:

```
Page 20 of 20
```

SPACES

Generates the given number of blank characters.

Syntax

```
num-spaces SPACES
```

1. *num-spaces* is the number of spaces.

Usage

This operator returns a string of blanks, equivalent to a quoted string containing the specified number of blanks.

In a PRINT statement, these blanks are inserted at the current character position.

Its operand must be an integer expression that returns a positive number, specifying an offset (from the current character position) no greater than the difference (right margin - current position). After PRINT SPACES has executed, the new current character position has moved to the right by the specified number of characters.

Outside PRINT statements, SPACES and its operand must appear within parentheses: (*n* SPACES).

Example

```
ON EVERY ROW
   LET s = (6 SPACES), "=ZIP"
   PRINT r.fname, 2 SPACES, r.lname, s
```

WORDWRAP

Splits a character string to match a given margin limit.

Syntax

```
WORDWRAP [ RIGHT MARGIN position ]
```

1. *position* defines the temporary right margin, as a number of characters, counting from the left.

Usage

The WORDWRAP operator automatically wraps successive segments of long character strings onto successive lines of report output. Any string value that is too long to fit between the current position and the right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary left margin.
- Unless you specify RIGHT MARGIN, the right margin defaults to 132, or to the size value from the RIGHT MARGIN clause of the OUTPUT section or START REPORT instruction.

Specify WORDWRAP RIGHT MARGIN *integer* to set a temporary right margin as a number of characters, counting from the left edge of the page. This value cannot be smaller than the current character position or greater than right margin defined for the report. The current character position becomes the temporary left margin. These temporary values override the specified or default left and right margins of the report.

After the PRINT statement has executed, any explicit or default margins defined in the RIGHT MARGIN clause of the OUTPUT section or START REPORT instruction are restored.

The following PRINT statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the variable called mynovel:

```
PRINT COLUMN 10, mynovel WORDWRAP RIGHT MARGIN 70
```

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), LINEFEED (ASCII 10), and ENTER (ASCII 13) characters to partition the string into words that consist of substrings of other printable characters. Other non-printable characters might cause runtime errors. If the data string cannot fit between the margins of the current line, the report engine breaks the line at a word division, and pads the line with blanks at the right.

From left to right, the report engine expands any TAB character to enough blank spaces to reach the next tab stop. By default, tab stops are in every eighth column, beginning at the left-hand edge of the page. If the next tab stop or a string of blank characters extends beyond the right margin, the report engine takes these actions:

1. Prints blank characters only to the right margin.
2. Discards any remaining blanks from the blank string or tab.
3. Starts a new line at the temporary left margin.
4. Processes the next word.

The report engine starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, this action creates an even left margin. The following rules are applied (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any LINEFEED, or ENTER, or LINEFEED, ENTER pair.
- Break at the last blank (ASCII 32) or TAB character before the right margin.
- Break at the right margin, if no character farther to the left is a blank, ENTER, TAB, or LINEFEED character.

The report engine maintains page discipline under the WORDWRAP option. If the string is too long for the current page, the report engine executes the statements in any page trailer and header control blocks before continuing output onto a new page.

For Japanese locales, a suitable break can also be made between the Japanese characters. However, certain characters must not begin a new line, and some characters must not end a line. This convention creates the need for KINSOKU processing, whose purpose is to format the line properly, without any prohibited word at the beginning or ending of a line.

Reports use the wrap-down method for WORDWRAP and KINSOKU processing. The wrap-down method forces down to the next line characters that are prohibited from ending a line. A character that precedes

another that is prohibited from beginning a line can also wrap down to the next line. Characters that are prohibited from beginning or ending a line must be listed in the locale. The runtime system tests for prohibited characters at the beginning and ending of a line, testing the first and last visible characters. The KINSOKU processing only happens once for each line. That is, no further KINSOKU processing occurs, even if prohibited characters are still on the same line after the first KINSOKU processing.

Report aggregate functions

Report aggregate functions can be used to compute data.

COUNT

Counts a number of rows in a report according to a condition.

Syntax

```
[GROUP] COUNT(*) [ WHERE condition ]
```

1. *condition* is a boolean expression evaluated to compute the aggregate value.

Usage

This aggregate report instruction returns the total number of records qualified by the optional `WHERE` condition.

The `WHERE` condition is evaluated after any `OUTPUT TO REPORT` execution. Even if it is typically used in `AFTER GROUP OF` blocks, the aggregate expression is not evaluated in that block: Changing values of the `WHERE` clause in the `AFTER GROUP` context will not have an immediate effect.

Using the `GROUP` keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the `AFTER GROUP OF` control block.

Example

The following fragment of a report definition uses the `AFTER GROUP OF` control block and `GROUP` keyword to form sets of records according to how many items are in each order. The last `PRINT` statement calculates the total price of each order, adds a shipping charge, and prints the result. Because no `WHERE` clause is specified here, `GROUP SUM()` combines the *total_price* of every item in the group included in the order.

```
  AFTER GROUP OF number
    SKIP 1 LINE
    PRINT 4 SPACES, "Shipping charges for the order: ",
            ship_charge USING "$$$$.&&"
    PRINT 4 SPACES, "Count of small orders: ",
         GROUP COUNT(*) WHERE total_price < 200.00 USING
 "##,###"
    SKIP 1 LINE
    PRINT 5 SPACES, "Total amount for the order: ",
          ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$
$.&&"
```

PERCENT

Calculates the percentage of rows matching a condition.

Syntax

```
[GROUP] PERCENT(*) [ WHERE condition ]
```

1. *condition* is a boolean expression evaluated to compute the aggregate value.

Usage

This aggregate report instruction returns the percentage of the total number of records qualified by the optional `WHERE` condition.

Using the `GROUP` keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the `AFTER GROUP OF` control block.

This aggregate instruction makes a two-pass report when not using the `GROUP` keyword and is used in any control block other than `ON LAST ROW`, or when using the `GROUP PERCENT(*)` anywhere in the report.

SUM

Calculates the total of a report parameter according to a condition.

Syntax

```
[GROUP] SUM(expression) [ WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

Usage

This aggregate report instruction evaluates as the total of expression among all records or among records qualified by the optional `WHERE` clause and any `GROUP` specification.

Using the `GROUP` keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the `AFTER GROUP OF` control block.

Input records for which the *expression* evaluates to `NULL` values are ignored.

By default, if all input record values are `NULL` , the result of the aggregate is `NULL`. However, you can control this behavior and force the runtime system to return zero instead of `NULL` with the `report.aggregateZero` FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the `GROUP` keyword and is used in any control block other than `ON LAST ROW`.

AVG

Calculates the average of a report parameter according to a condition.

Syntax

```
[GROUP] AVG(expression) [ WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

Usage

This aggregate report instruction evaluates as the average (that is, the arithmetic mean value) of expression among all records or among records qualified by the optional WHERE clause and any GROUP specification.

Using the GROUP keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the AFTER GROUP OF control block.

Input records for which the *expression* evaluates to NULL values are ignored.

By default, if all input record values are NULL , the result of the aggregate is NULL. However, you can control this behavior and force the runtime system to return zero instead of NULL with the report.aggregateZero FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the GROUP keyword and is used in any control block other than ON LAST ROW.

MIN

Calculates the minimum value of a report parameter according to a condition.

Syntax

```
[GROUP] MIN(expression) [ WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

Usage

For number, currency, and interval values, MIN(*expression*) aggregate report instruction returns the minimum value for *expression* among all records or among records qualified by the WHERE clause and any GROUP specification.

For DATETIME or DATE data values, greater than means later and less than means earlier in time. Character strings are sorted according to their first character. If your program is executed in the default (U.S. English) locale, for character data types, greater than means after in the ASCII collating sequence, where a> A> 1, and less than means before in the ASCII sequence, where 1< A< a.

Using the GROUP keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the AFTER GROUP OF control block.

Input records for which the *expression* evaluates to NULL values are ignored.

By default, if all input record values are NULL , the result of the aggregate is NULL. However, you can control this behavior and force the runtime system to return zero instead of NULL with the report.aggregateZero FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the GROUP keyword and is used in any control block other than ON LAST ROW.

MAX

Calculates the maximum value of a report parameter according to a condition.

Syntax

```
[GROUP] MAX(expression) [ WHERE condition]
```

1. *expression* is the expression to be computed.
2. *condition* is a boolean expression evaluated to compute the aggregate value.

Usage

For number, currency, and interval values, the MAX(*expression*) aggregate report instruction returns the maximum value for *expression* among all records or among records qualified by the WHERE clause and any GROUP specification.

For DATETIME or DATE data values, greater than means later and less than means earlier in time. Character strings are sorted according to their first character. If your program is executed in the default (U.S. English) locale, for character data types, greater than means after in the ASCII collating sequence, where a> A> 1, and less than means before in the ASCII sequence, where 1< A< a.

Using the GROUP keyword causes the aggregate instructions to include only data of the current group of records that have the same value for the variable that you specify in the AFTER GROUP OF control block.

Input records for which the *expression* evaluates to NULL values are ignored.

By default, if all input record values are NULL , the result of the aggregate is NULL. However, you can control this behavior and force the runtime system to return zero instead of NULL with the report.aggregateZero FGLPROFILE parameter.

This aggregate instruction makes a two-pass report when not using the GROUP keyword and is used in any control block other than ON LAST ROW.

Report engine configuration

Report engine behavior can be controlled with FGLPROFILE settings.

By default, aggregate instructions such as SUM() return a NULL value if all input record values are NULL.

You can force the report engine to return a zero decimal value with the following FGLPROFILE setting:

```
Report.aggregateZero = {true|false}
```

When this entry is set to true, the SUM() , AVG() , MAX() and MIN() aggregate functions return zero when all values are NULL.

Default value of the configuration parameter is false (i.e. aggregate functions evaluate to null if all items are null)

When using GROUP aggregates with this entry is set to true, the aggregate instruction will still return NULL in the first AFTER GROUP OF output of the report. Zero values will be returned starting from second group output. This behavior is expected, for backward compatibility with older versions.

You should not use the Report.aggregateZero entry if you don't need that specific behavior.

Programming tools

These topics cover programming with the Genero Business Development Language.

- Command line tools on page 1493
- Compiling source files on page 1508
- Source code edition on page 1516
- Source documentation on page 1517
- The preprocessor on page 1522
- The debugger on page 1531
- The profiler on page 1556
- Optimization on page 1559
- Logging options on page 1563

Command line tools

The different command line tools provided for BDL programming.

- fglrun on page 1493
- fglform on page 1495
- fgl2p on page 1496
- fglcomp on page 1497
- fgllink on page 1499
- fglmkmsg on page 1500
- fglmkext on page 1500
- fgldb on page 1501
- fgldbsch on page 1502
- fglmkstr on page 1503
- fglwsdl on page 1503
- fglpass on page 1506

fglrun

The fglrun tool is the runtime system program that executes p-code programs.

Syntax

```
fglrun [options] program [argument [...]]
```

1. *options* are described in Table 306: fglrun options on page 1493.
2. *program* is a .42r or .42m p-code program.
3. *argument* is an argument passed to the program

Options

Table 306: fglrun options

Option	Description
-V	Display version information for the tool.
-h	Displays options for the tool. Short help.

Option	Description
`-i { mbcs }`	Displays information. • `-i mbcs` displays information about multibyte character set settings.
`-d`	Start in debug mode. See The debugger on page 1531 for more details.
`-e extfile[,...]`	Specify a C extension module to be loaded. This option can take a comma-separated list of extensions.
`-l`	Link p-code modules together, see Compiling source files on page 1508.
`-o { progname.42r | libname.42x }`	Output file specification when using the `-l` link option, it can be a 42r program or a 42x library.
`-b`	Displays compiler version information of the module, see Compiling source files on page 1508.
`-p`	Generate profiling information to stderr (UNIX™ only). See The profiler on page 1556.
`-M`	Display a memory usage diagnostic when program ends. See Check runtime system memory leaks on page 1561.
`-m`	Check for memory leaks. If leaks are found, displays memory usage diagnostic and stops with status 1. See Check runtime system memory leaks on page 1561.
`--java-option=option`	Passes Java™ runtime options when initializing the JNI interface. See Java™ Interface for more details.
`--print-imports`	Loads the specified modules and prints all `IMPORT FGL` instructions that should be used in each module. See Compiling source files on page 1508.
`--start-guilog=logfile`	Log all GUI protocol exchange in a file. The GUI log file can then be replayed with the `--run-guilog` option.
`--run-guilog=logfile`	Replays a GUI log created with the `--start-guilog` option.
`--gui-listen=port`	Instructs the runtime system to listen to a TCP port for incoming GUI connections. For more details see Connecting with a front-end on page 755.
`--module-size module`	Show the amount of limited pcode size for a module.
`--program-size program`	Show the amount of limited pcode size for an entire program.

Usage

The `fglrun` command line tool executes p-code programs, for example:

```
fglrun myprogram.42r -x 123
```

The program file must contain the `MAIN` routine.

The arguments passed to the program can be queried with the arg_val() built-in function.

The .42r or .42m extension is optional:

```
fglrun myprogram -x 123
```

> **Note:** First `fglrun` tries to find the program file with the name provided in the command line. If the file is not found, the extension is removed (if it is present in the provided file name), and a new search is done by adding the `.42r` extension. If the file is still not found, `fglrun` tries with the `.42m` extension. As result, a program file `myprogram.42m` will be found and loaded, even if you pass `myprogram.42r` to `fglrun`. Specify no `.42r` or `.42m` extension, to avoid mistakes and simplify migration from `.42r` linked programs to `.42m`-only modules (using `IMPORT FGL`).

fglform

The fglform tool compiles form specification files into XML formatted files used by the programs.

Syntax

```
fglform [options] srcfile[.per]
```

1. *options* are described in Table 307: fglform options on page 1495.
2. *srcfile*.per is the form specification file.

Options:

Table 307: fglform options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-i { mbcs }`	Displays information. `-i mbcs` displays information about multibyte character set settings.
`-m`	Extract localized strings.
`-M`	Write error messages to standard output instead of creating a **.err** error file.
`-W { all }`	Display warning messages. Only `-W all` option is supported for now.
`-E`	Preprocess only.
`-p option`	Preprocessing control, where *option* can be one of: • `nopp`: Disable preprocessing. • `noli`: No line number information (only with -E option). • `fglpp`: Use # syntax instead of & syntax.

Option	Description
`-I path`	Provides a path to search for include files.
`-D ident`	Defines the macro 'ident' with the value 1.

Usage

The `fglform` command line tool compiles a .per form specification file into a .42f compiled version:

```
fglform custform.per
```

The .per extension is optional, if not used, fglform will automatically search for files with this extension.

The .42f compiled version is an XML formatted file used by programs when a form definition is loaded with the `OPEN FORM` or `OPEN WINDOW WITH FORM` instructions.

fgl2p

The fgl2p tool compiles source files and assembles p-code modules into a .42r program or a .42x library.

Syntax

To create a library:

```
fgl2p [options] -o outfile.42x { pcmod.42m | srcfile.4gl } [...]
```

To create a program:

```
fgl2p [options] -
o outfile.42r { pcmod.42m | srcfile.4gl | library.42x } [...]
```

1. *options* are described in Table 308: fgl2p options on page 1496.
2. *outfile*.42r is the name of the program to be created.
3. *outfile*.42x is the name of the library to be created.
4. *pcmod*.42m is a p-code module compiled with fglcomp.
5. *source*.4gl is a program source file.
6. *library*.42x is the name of a library to be linked.

Options

Table 308: fgl2p options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-o outfile.ext`	Output file specification, where *ext* can be 42r for a program or 42x for a library.
otheroption	Other options are passed to the linker or compiler.

Usage

The `fgl2p` command line tool can compile .4gl source files and link .42m p-code modules together, to create a .42x library or a .42r program file.

```
fgl2p -o myprog.42x module1.4gl module2.42m lib1.42x
```

This tool is provided for convenience, in order to create programs or libraries in one command line. It uses the fglcomp and the fgllink tools to compile and link modules together.

fglcomp

The `fglcomp` tool compiles `.4gl` source files into `.42m` p-code modules.

Syntax

```
fglcomp [options] srcfile[.4gl]
```

1. *options* are described in Table 309: fglcomp options on page 1497.
2. *srcfile* .4gl is the program source file.
3. The .4gl extension is optional.

Options

Table 309: fglcomp options

Option	Description
`-V` or `--version`	Display version information for the tool.
`-h` or `--help`	Display options for the tool. Short help.
`-i { mbcs }`	Display information. `-i mbcs` displays information about multibyte character set settings.
`-S`	Dump Static SQL statements found in the source to stdout.
`-m`	Extract %"string" localized strings from source to stdout.
`-M`	Write error messages to standard output instead of creating a .err error file.
`-W what`	Display warning messages. For a complete description, see Arguments for the -W option on page 1498.
`-E`	Preprocess only. See The preprocessor on page 1522 for more details.
`--timestamp`	Add compilation timestamp to build information in 42m header.
`-p option`	Preprocessing control, where *option* can be one of: • `nopp`: Disable preprocessing. • `noli`: No line number information (only with `-E` option). • `fglpp`: Use # syntax instead of & syntax.

Option	Description
`-G`	Produce .c and .h globals interface files for C-Extensions.
`-I` *`path`*	Provides a path to search for include files. See The preprocessor on page 1522 for more details.
`-D` *`ident`*	Defines the macro 'ident' with the value 1. See The preprocessor on page 1522 for more details.
`-U` *`ident`*	Undefines the macro 'ident'. See The preprocessor on page 1522 for more details.
`--build-doc`	Generate source documentation.
`--doc-private`	When using the `--build-doc` option, include `PRIVATE` symbols to the documentation.
`--build-rdd`	Generate the .rdd data definition during compilation.
`--verbose`	Print detailed compilation information.
`--implicit=`*`type`*	Specify whether or not to compile imported modules, where *type* can be one of: • `none`: Disable any implicit compilation. • `42m`: Compileimported modules if needed (the default).
`-r` or `--resolve-calls`	Throw an error on references to undeclared functions. Each external function must be made known to the compiler by `IMPORT FGL`. When using this option, the linking phase is no longer needed; a source (`4gl`) file compiled with this option must not be linked. See IMPORT FGL module on page 372 for more details.
`--java-option=`*`option`*	Passes Java™ runtime options when initializing the JNI interface. See Java™ Interface for more details.

Usage

The `fglcomp` command line tool compiles a .4gl into a .42m p-code module:

```
fglcomp customers.4gl
```

If a compilation error occurs, the compiler generates an error file with an `.err` extension. The error file contains the original source code with error messages. Use the option `-M` to display the error messages to standard error instead of producing the `.err` file.

Arguments for the -W option

The `-W` option can be used to check for wrong language usage, that must be supported for backward compatibility. When used, this option helps to write better source code.

The argument following `-W` option can be one of `return`, `unused`, `stdsql`, `print`, `error` or `all`.

- Using `-W all` enables all warning flags.
- Using `-W error` makes the compiler stop if any warning is raised, as if an error occurred.
- The `-W unused` option displays a message for all unused variables.
- The `-W return` option displays a warning if the same function returns different number of values with several `RETURN` statements.
- The `-W stdsql` option displays a message for all non-portable SQL statements or language instructions.
- The `-W print` option displays a message when the `PRINT` instruction is used outside a `REPORT`.
- The `-W implicit` option warns on references to undeclared functions. A function is undeclared if not defined in the current module or in any imported module. This warning is silently ignored if `IMPORT FGL` is not used.
- The `-W apidoc` option prints a warning for invalid source documentation tags when using the --build-doc option.

The `-W` option also supports the negative form of arguments by using the `no-` prefix as in: `no-return`, `no-unused`, `no-stdsql`. You might need to use these negative form in order to disable some warning when using the `-W all` option:

```
fglcomp -Wall -Wno-stdsql customers.4gl
```

Switches will be enabled/disabled in the order of appearance in the command line.

fgllink

The fgllink tool assembles p-code modules produced with fglcomp into a .42r program or a .42x library.

Syntax

To create a library:

```
fgllink [options] -o outfile.42x module.42m [...]
```

To create a program:

```
fgllink [options] -o outfile.42r { module.42m | library.42x } [...]
```

1. *options* are described in Table 310: fgllink options on page 1499.
2. *outfile*.42r is the name of the program to be created.
3. *outfile*.42x is the name of the library to be created.
4. *module*.42m is a p-code module compiled with fglcomp.
5. *library*.42x is the name of a library to be linked.

Options

Table 310: fgllink options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-e extfile[,...]`	Specify a C extension module to be loaded. This option can take a comma-separated list of extensions.
`-o { progname.42r | libname.42x }`	Output file specification, it can be a 42r program or a 42x library.

Option	Description
otheroption	Other options are passed to fglrun for linking.

Usage

The `fgllink` command line tool links .42m p-code modules together to create a .42x library or a .42r program file.

```
fgllink -o myprog.42x module1.42m module2.42m lib1.42x
```

> **Note:** `fgllink` is a wrapper calling fglrun with the **-l** option.

fglmkmsg

The fglmkmsg tool compiles .msg message files into a binary version used by programs.

Syntax

```
fglmkmsg [options] srcfile [outfile]
```

1. *options* are described in Table 311: fglmkmsg options on page 1500.
2. *srcfile* is the source message file.
3. *outfile* is the destination file.

Options

Table 311: fglmkmsg options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-r msgfile`	De-compiles a binary message file.

Usage

The `fglmkmsg` command line tool compiles a .msg message file into a .iem compiled version:

```
fglmkmsg mess01.msg
```

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

The .iem compiled version can be used by BDL programs, for example, when the `HELP` clause is used in a `MENU` or `INPUT` instruction.

fglmkext

The fglmkext tool compiles and links a user C Extension.

Syntax

```
fglmkext [options] source.c [...]
```

1. *options* are described in Table 312: fglmkext options on page 1501.
2. *source* is a C source file implementing C extension functions.

Options

Table 312: fglmkext options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-o` *`libname`*	Output file specification, defines the C Extension library name.

Usage

The `fglmkext` command line tool compiles and links a C Extension library.

The command can be used with a single source file, the name of the library will default to the name of the specified source:

```
fglmkext myext.c
```

If a single C source file is provided, must define the `usrFunctions` C extension interface structure as well as the functions to be used from a BDL program.

In order to specify a library name, use the `-o` option, several C source files can also be specified. For example, on a UNIX platform:

```
fglmkext -o mycext.so module_a.c module_b.c
```

fgldb

The fgldb tool is an interface program for remote debugging.

Syntax 1: Debugging an application running on a server

```
fgldb -p process-id
```

1. *process-id* is the process identifier of the fglrun-bin / fglrun.exe program.

Syntax 2: Debugging an app running on a mobile device

```
fgldb -m host[:port] ]
```

1. *host* is the host (or IP address) of the mobile device where the program executes, default is "localhost".
2. *port* is the TCP port number to connect to, default is 6400.

Options

Table 313: fgldb options

Option	Description
`-V`	Display version information for the tool.

Option	Description
`-h`	Displays options for the tool.
`-p process-id`	Attach to a running process to debug
`-m host[:port]`	Attach to a running process to debug

Usage

The `fgldb` command line tool is an interface for remote debugging, attaching to a Genero program running on a server or on a mobile device (or mobile emulator).

fgldbsch

The fgldbsch tool generates the database schema files from an existing database.

Syntax

```
fgldbsch -db dbname [options]
```

1. *dbname* is the name of the database from which the schema is to be extracted.
2. *options* are described in Table 314: fgldbsch options on page 1502.

Options

Table 314: fgldbsch options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-H`	Display long help.
`-v`	Enable verbose mode (display information messages).
`-ct`	Display data type conversion tables.
`-cx dbtype`	Display data type conversion table for the give database type.
`-db dbname`	Specify the database as *dbname*. This option is required to generate the schema files.
`-dv dbdriver`	Specify the database driver to be used.
`-un user`	Define the user name for database connection as *user*.
`-up pswd`	Define the user password for database connection as *pswd*.
`-ow owner`	Define the owner of the database tables as *owner*.
`-cv string`	Specify the data type conversion rules by character positions in *string*.
`-of name`	Specify output files prefix, default is database name.

Option	Description
`-tn` *tabname*	Extract the description of a specific table.
`-ie`	Ignore tables with columns having data types that cannot be converted.
`-cu`	Generate upper case table and column names.
`-cl`	Generate lower case table and column names.
`-cc`	Generate case-sensitive table and column names.
`-sc`	Extract shadow columns.
`-st`	Extract system tables.
`-om`	Run schema extractor in old fglschema mode (accepts -c and -r options)

Usage

The `fgldbsch` command line tool extracts the schema description for any database supported by the product.

The .sch schema file is mandatory to compiler forms or source modules using the `SCHEMA` instruction.

fglmkstr

The fglmkstr tool compiles .str localized string resource files.

Syntax

```
fglmkstr [options] source-file[.str]
```

1. *options* are described in Table 315: fglmkstr options on page 1503.
2. *source-file* is the .str string file. You can omit the file extension.

Options

Table 315: fglmkstr options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.

Usage

The `fglmkstr` command line tool is used to compile .str localized string files into .42s files.

fglwsdl

The fglwsdl tool produces web services stub files for client or server programs (from WSDL / XSD).

Syntax

```
fglwsdl command [options] parameter
```

1. *command* indicates what operation must be done by fglwsdl.

2. *options* are described in Commands and options on page 1504.
3. *parameter* depends on the *command* used.

Commands and options

Table 316: fglwsdl commands

Command	Description
`-V`	Display version information
`-h`	Display this help
`-l`	List services from a WSDL or variables from a XSD
`-c [`*`options`*`] `*`wsdl-spec`*	Generate client stub (default) to be used in a GWS client application. *wsdl-spec* is the name of a WSDL description file or the URL of a WSDL description for a published web service. Typically, `http://`*`host/service`*`?WSDL`. The *options* are listed in Table 317: WSDL Options on page 1504 and Table 319: Common options on page 1505.
`-s [`*`options`*`] `*`wsdl-spec`*	Generate server stub to be used in a GWS server application. *wsdl-spec* is the name of a WSDL description file or the URL of a WSDL description for a published web service. Typically, `http://`*`host/service`*`?WSDL`. The *options* are listed in Table 317: WSDL Options on page 1504 and Table 319: Common options on page 1505.
`-x [`*`options`*`] `*`xsd-spec`*	Generate BDL data types from a XML schema (XSD). *xsd-spec* is the name of an XML schema file or the URL of an XSD schema resource on the web. The *options* are listed in Table 318: XSD Options on page 1505 and Table 319: Common options on page 1505.
`-regex `*`regex value`*	Validate the *value* against the *regex* regular expression described in XML schema specification.

Table 317: WSDL Options

Options	Description
`-o `*`file`*	Specify a base name for the output files.
`-n `*`service port`*	Generate only for the given service name and port type.
`-b `*`binding`*	Generate only for the given binding.
`-prefix `*`name`*	Add *name* as the prefix of the generated web service functions, variables and types. (*name* can contain %s for servicename, %p for portname and %f for filename)
`-compatibility`	Generate a Genero 1.xx compatibility client stub.
`-fRPC`	Force RPC convention; use RPC Convention to generate the code, regardless of what the WSDL information contains.
`-disk`	Save WSDL and all dependencies from an URL on the disk. **Note:** To generate code at the same time, you must use the option -c, -s, or both. Otherwise, no code is generated.
`-domHandler`	Generates the use of DOM in the client stub and calls to callback handlers.

Options	Description
`-alias`	Generates FGLPROFILE Logical names in place of URLs for the client stub.
`-soap11`	Generates only client and server stubs supporting SOAP 1.1 protocol.
`-soap12`	Generates only client and server stubs supporting SOAP 1.2 protocol.
`-ignoreFaults`	Do not generate extra code to handle soap faults.
`-wsa <yes|no>`	Force support of WS-Addressing 1.0 if **yes**, disable support of WS-Addressing 1.0 if **no**, otherwise support WS-Addressing 1.0 according to the WSDL definition.

Table 318: XSD Options

Options	Description
`-o` *`file`*	Name of the output file. If *file* has no extension, .inc is added.
`-n` *`name [ns]`*	Generate only for the given variable name and namespace (if there is one).
`-prefix` *`name`*	Add *name* as the prefix of the generated data types.
`-disk`	Save XSD and all dependencies from an URL on the disk. **Note:** No code is generated.

Table 319: Common options

Options	Description
`-comment`	Add XML comments to the generation.
`-fArray`	Force XML array generation instead of XML list when possible. If the WSDL contains an XML definition of a BDL list, generate a BDL array matching the same definition.
`-fInheritance`	Force generation of XML choice records for all inheritance types found in the schemas, otherwise only for abstract types and elements.
`-fInlineTypes`	Force generation of TYPE definitions for all global inlined types found in the schemas.
`-noFacets`	Don't generate facet constraints restricting the value-space of simple data type.
`-legacyTypes`	Don't generate BIGINT, TINYINT and BOOLEAN data types.
`-ignoreMixed`	Ignore attribute mixed="true" in XML schemas when generating code.
`-ext` *`schema`*	Add an external schema. See option '-extDir'.
`-extDir` *`directory`*	Add all external schema files ending with .xsd in the directory. **Note:** External schemas for dependencies won't be included in the WSDL description or in the XSD schema if their location attributes are missing. Use this option to add a missing external schema for a WSDL or XSD dependency.
`-noValidation`	Disable XML schema validation warnings.
`-autoNsPrefix` *`nb`*	Automatic prefix generation for variables and types using a substring of the namespace by removing the nb first elements (-1 means only the last element). For example: If a variable belongs to the namespace `http://www.mycompany.com/Global/Service`, a value of -1 will give `Service` as a prefix, and a value of 1 will give `Global_Service` as a prefix.

Options	Description
`-nsPrefix` *`ns value`*	Add *value* as prefix of the generated variables and types belonging to namespace ns (supersede the -prefix and the -autoNsPrefix option, and can be called several times).

Table 320: Network options (when specifying an URL)

Options	Description
`-noHTTP`	Disable HTTP - search for the WSDL description or the XML schema and its dependencies on the client instead of the internet. Useful, for example, if a company has restricted access to the internet.
`-proxy` *`location`*	Connect via proxy where *location* is host[:port] or ip[:port].
`-pAuth` *`login pass`*	Proxy authentication login and password.
`-hAuth` *`login pass`*	HTTP authentication login and password.
`-cert` *`cert`*	File of the X509 PEM-encoded certificate for HTTPS purpose.
`-key` *`key`*	File of the PEM-encoded private key for HTTPS purpose.
`-wCert` *`cert`*	Certificate name in the Windows™ keystore for HTTPS purpose (Windows™ only).
`-CA` *`list`*	A filename with the list of concatenated X509 PEM-encoded certificate authorities. (On Windows™, if not set, the Certificate Authority list of the key store is used).

Usage

The `fglwsdl` command line tool produces the WSDL description of a web service that will be accessed by a GWS client application, or to define a WSDL description to creating a corresponding GWS server application. The tool generates the BDL data types from XML schemas (also known as XSD).

To access a remote web service, you must get the WSDL information from the service provider. Sample services can be found through UDDI registries (http://www.uddi.org) , or on other sites such as XMethods (http://www.xmethods.net).

fglpass

The fglpass tool allows you to encrypt passwords.

Syntax

```
fglpass [options]
```

1. *options* are described in fglpass options.

Options

Table 321: fglpass options

Command	Description
`-V`	Display version information
`-Vssl`	Display SSL version
`-h`	Display this help

Command	Description
`-e`	Encrypt the password with a RSA key or certificate and encode it in BASE64 form
`-d`	Decode the BASE64 form of the password and decrypt it with a RSA private key
`-w` *`cert`*	Windows™ certificate name to encrypt the password (Windows™ only)
`-c` *`cert`*	File of the PEM-encoded certificate to encrypt the password
`-k` *`key`*	File of the PEM-encoded private key to encrypt or decrypt the password
`-enc64` *`file`*	File to be BASE64 encoded (result to stdout)
`-dec64` *`file`*	BASE64 encoded file to be decoded (result to stdout)
`-agent:`*`port files`*	Start password agent on specified port to serve the list of private key files

Usage

The `fglpass` command line tool allows you to:

- Encrypt a password using a RSA key or X.509 certificate and encode it in BASE64 form.
- Run a password agent that returns (in a protected way) the passwords that grant access to the different private keys used in all your applications.
- Encode a file in BASE64 form and decode it back.

For security reasons, it is recommended to avoid storing clear passwords in a file, or leave private keys unprotected without a password. The fglpass command can be used to encrypt passwords.

fglWrt

The fglWrt tool is used to manage product licenses.

Syntax

```
fglWrt [options]
```

1. *options* are described in Table 322: fglmkstr options on page 1507.

Options

Table 322: fglmkstr options

Option	Description
`-V`	Display version information for the tool.
`-h`	Displays options for the tool. Short help.
`-l` *`license`*	Installs a license. **Note:** To escape when prompted to enter a license key, type "`stop`" at the prompt.

Option	Description
`-m` *key*	Maintenance key specification.
`-u`	Check for active users.
`-k` *key*	Installation key for license validation.
`-d`	Remove current installed license.
`-i`	Clears the list of registered user sessions.
`-a` *option*	Check or view options, possible options are: • ps : Shows processes on this machine. • cpu : Shows number of CPU in the computer. • hostname : Shows name of this machine. • info license : Shows license information. • info stat : Shows statitics of license server. • info users : Shows all registered active users. • info up : Shows if license server is up.

Usage

The `fglWrt` command line tool is used to install, upgrade or delete licenses.

If no license is installed, it is not possible to use Genero Business Development Language.

Compiling source files

Describes how to build the runtime files from source files.

- Compiling form files on page 1508
- Compiling message files on page 795
- Compiling source code on page 1510
- Importing modules on page 1511
- Linking libraries on page 1511
- Linking programs on page 1512
- Using makefiles on page 1515
- Module build information on page 1515

Compiling form files

Form specification files (with .per file extension) must be compiled to runtime form files (with .42f file extension) by using the fglform tool. Compiled form files are XML independent from the platform and processor architecture.

The following lines show a compilation in a UNIX™ shell session:

```
$ cat form.per
LAYOUT
GRID
{
[f01   ]
}
END
END
ATTRIBUTES
f01 = FORMONLY.field1;
```

```
END

$ fglform form.per

$ ls -s form.42f
   4 form.42f
```

If an error occurs, the compiler writes an error file with the .err extension.

```
$ cat form.per
LAYOUT
GRID
{
}

$ fglform form.per
The compilation was not successful.  Errors found: 1.
 The file 'form.err' has been written.

$ cat form.err
LAYOUT
GRID
{
}
# A grammatical error has been found at '}', expecting SCR_TEXT.
# See error number -6803.
```

With the `-M` option, you can force the compiler to display an error message instead of generating an .err error file (line break added for documentation readability):

```
$ fglform -M form.per
form.per:4:1:4:1:error:(-6803)
 A grammatical error has been found at '}', expecting SCR_TEXT.
```

By default, the compiler does not raise any warnings. You can turn on warnings with the `-W` option:

```
$ cat form.per
LAYOUT
GRID
{
[f01     ]
}
END
END
ATTRIBUTES
f01 = FORMONLY.field1, WIDGET="COMBO";
END

$ fglform -Wall form.per
form.per:9: warning (-8005) Deprecated feature: The WIDGET attribute is
 obsolete
```

Compiling message files

In order to use message files in a program, the message source files (with .msg extension) must be compiled with the fglmkmsg utility to produce compiled message files (with .iem extension).

The following command line compiles the message source file mess01.msg:

```
fglmkmsg mess01.msg
```

This creates the compiled message file mess01.iem.

For backward compatibility, you can specify the output file as second argument:

```
fglmkmsg mess01.msg mess01.iem
```

The .iem compiled version of the message file must be distributed on the machine where the programs are executed.

Compiling string files

The source string files (with `.str` extension) must be compiled to binary files (with `.42s` extension) in order to be loaded by the runtime system.

To compile a source string file, use the `fglmkstr` compiler.

```
$ fglmkstr filename.str
```

The `fglmkstr` tool generates a `.42s` file with the *filename* prefix.

> **Important:** When compiling a `.str` source string file, you must set the locale (character set) corresponding to the encoding used in the `.str` file.

Compiling source code

Source code modules (with .4gl file extension) must be compiled to p-code modules (with .42m file extension) by using the fglcomp tool. Compiled p-code modules are independent from the platform and processor architecture.

The following lines show a compilation in a UNIX™ shell session:

```
$ cat prog.4gl
MAIN
  DISPLAY "hello"
END MAIN

$ fglcomp prog.4gl

$ ls -s prog.42m
   4 prog.42m
```

If an error occurs, the compiler writes an error file with the .err extension.

```
$ cat prog.4gl
MAIN
  LET x = "hello"
END MAIN

$ fglcomp prog.4gl
Compilation was not successful. Errors found: 1.
 The file prog.4gl has been written.

$ cat prog.err
MAIN
  LET x = "hello"
| The symbol 'x' does not represent a defined variable.
| See error number -4369.
END MAIN
```

With the `-M` option, you can force the compiler to display an error message instead of generating an .err error file:

```
$ fglcomp prog.4gl
xx.4gl:2:8 error:(-4369) The symbol 'x' does not represent a defined
 variable.
```

By default, the compiler does not raise any warnings. You can turn on warnings with the `-W` option:

```
$ cat prog.4gl
MAIN
  DATABASE test1
  SELECT COUNT(*) FROM x, OUTER(y) WHERE x.k = y.k
END MAIN

$ fglcomp -W stdsql prog.4gl
xx.4gl:3: warning: SQL statement or language instruction with specific SQL
 syntax.
```

When a warning is raised, you can use the `-W error` option to force the compiler to stop as if an error was found. For more details about the `-W` option, see Arguments for the -W option on page 1498.

Consider also using the `--verbose` option of the compiler to get detailed information about the source compilation:

```
$ fglcomp --verbose main.4gl
[parsing main.4gl]
[compiling: fglcomp --import-by=main --verbose  mod1]
[parsing mod1.4gl]
[compiling: fglcomp --import-by=main,mod1 --verbose  mod2]
[parsing mod2.4gl]
[writing mod2.42m]
[loading mod2.42m]
[writing mod1.42m]
[loading mod1.42m]
[writing main.42m]
```

Importing modules

With the `IMPORT FGL` instruction, module symbols such as variables, types and constants can be referenced in the importing module.

The next source example imports the *myutils* and *account* modules, and uses the *init()* and *set_account()* functions of the imported modules. The first function call is qualified with the module name - this is optional but required to resolve ambiguities when the same function name is used by different modules:

```
IMPORT FGL myutils
IMPORT FGL account
MAIN
  CALL myutils.init()
  CALL set_account("CFX4559")
  ...
END MAIN
```

Linking libraries

Compiled 42m modules can be grouped in libraries by using the fgllink linker. The library files get the 42x extension. If none of the modules defines the `MAIN` block, the linker creates a library; if a `MAIN` block is present, the linker creates a program, that should use a 42r extension.

Note that linking is supported for backward compatibility, you should use `IMPORT FGL` instead.

Library linking is done with the fglrun tool by using the `-l` option. The fgllink tool can be used for convenience, it is a simple script calling `fglrun -l`.

The following lines show a link procedure to create a library in a UNIX™ shell session:

```
$ fglcomp fileutils.4gl
  $ fglcomp userutils.4gl
  $ fgllink -o libutils.42x fileutils.42m userutils.42m
```

When you create a library, all functions of the 42m modules used in the link command are registered in the 42x file.

Keep in mind that the 42x library file does not contain the 42m p-code. When deploying your application, you must provide all compiled 42m modules.

When creating a 42x library, all functions must be uniquely defined; otherwise, error -6203 will be returned by the linker.

The 42x libraries are typically used to link the final 42r programs:

```
$ fglcomp mymain.4gl
  $ fgllink -o myprog.42r mymain.42m libutils.42x
```

The 42r programs must be re-linked if the content of 42x libraries have changed. In this example, if a function of the userutils.4gl source file was removed, you must recompile userutils.4gl, re-link the libutils.42x library and re-link the myprog.42r program.

It is possible to create a library by referencing other 42x library files in the link command, as long as 42M modules can be found:

```
$ fglcomp module_1.4gl
  $ fglcomp module_2.4gl
  $ fgllink -o lib_A.42x module_1.42m
  $ fgllink -o lib_B.42x module_2.42m lib_A.42x $ fgllink -o myprog.42r
 lib_B.42x
    -- will hold functions of module_1 and module_2.
```

If you do not specify an absolute path for a file, the linker searches by default for .42m modules and .42x libraries in the current directory. You can specify a search path with the FGLLDPATH environment variable.

If you are using C-Extensions, you may need to use the `-e` option to specify the list of extension modules, if the `IMPORT` keyword is not used:

```
$ fgllink -e extlib,extlib2,extlib3 -o libutils.42x fileutils.42m
 userutils.42m
```

Linking programs

Genero programs are created by linking several .42m modules and/or .42x libraries together, where one of the modules defines a `MAIN` block. By convention, the resulting program file gets the 42r extension.

Note that linking is supported for backward compatibility, you should use `IMPORT FGL` instead. When using `IMPORT FGL` to import all modules used by a program, the link stage is no longer required (you can directly execute the 42m module containing the `MAIN` block).

Program linking is done with the fglrun tool by using the `-l` option. The fgllink tool can be used for convenience; it is a simple script calling `fglrun -l`.

The following lines show a link procedure to create a program in a UNIX™ shell session:

```
$ fglcomp main.4gl
```

```
$ fglcomp store.4gl
$ fgllink -o stores.42r main.42m store.42m
```

The purpose of the linking phase is to check for missing function symbols, and reference all the symbols in the resulting .42r program file. Any function used in the .42m modules specified in the link line must be provided. Missing symbols will result in a -1338 linker error. Note that this applies only to programs: When linking a 42x library, there can be references to undefined functions:

```
$ cat main.4gl
MAIN
  CALL myfunc()
END MAIN

$ fglcomp main.4gl
$ fgllink -o prog.42r main.42m
ERROR(-1338):The function 'myfunc' has not been defined in any module in the
 program.
```

The generated 42r program files do not contain the 42m p-code. When deploying your application, you must provide all 42m modules, as well as 42r program files. Since 42x library files are only used to build programs, you do not have to deploy 42x library files.

If you omit the -o option in the fgllink command, the default output file will have the .42x extension and the name of the module containing the `MAIN` block. The .42r file extension is used by convention to distinguish a program dictionary file from a library dictionary file.

When linking a 42r program by using 42x libraries, the modules defined in a library are included only if one of the symbols in the module is used by the program. However, all symbols of 42m modules specified in the command line will always be referenced in the resulting 42r program file. The same function symbols can be defined in distinct libraries; the linker will select the function of the first library that was specified in the command line.

All symbols referenced in a module must exist in the final 42r program dictionary file. If a symbol is not found, the runtime system stops with error -1338. This error is fatal and cannot be trapped with an exception handler.

When linking a 42r program, global symbols must be unique; otherwise, error -6203 will be returned by the linker. The same error will be returned when linking a 42x library by using modules defining the same functions.

If you do not specify an absolute path for a file, the linker searches by default for .42m modules and .42x libraries in the current directory. You can specify a search path with the FGLLDPATH environment variable:

```
$ FGLLDPATH=/usr/dev/lib/maths:/usr/dev/lib/utils
$ export FGLLDPATH
$ ls /usr/dev/lib/maths
mathlib1.42x
mathlib2.42x
mathmodule11.42m
mathmodule12.42m
mathmodule22.42m
$ ls /usr/dev/lib/utils
fileutils.42m
userutils.42m
dbutils.42m
$ fgllink -o myprog.42r mymodule.42m mathlib1.42x fileutils.42m
```

In this example the linker will find the specified files in the `/usr/dev/lib/maths` and `/usr/dev/lib/utils` directories defined in FGLLDPATH.

When creating a .42r program by linking .42m modules with .42x libraries, if the same function is defined in a .42m and in a module of a 42x library, the function of the specified .42m module will be selected by the linker, and the function of the library will be ignored. However, the linker will raise error -6203 if two .42m modules specified in the link command define the same function.

The link procedure searches recursively for the functions used by the program. For example, if the `MAIN` block calls function FA in module MA, and FA calls FB in module MB, all functions from module MA and MB will be included in the 42r program definition.

When linking a program with modules using the `IMPORT FGL` instruction, you do not have to specify the imported modules in the link line, since these modules will be loaded dynamically at runtime. However, any symbol used by the imported module must be resolved by the linker. Therefore, if the imported module uses functions that come from other modules which are not imported by this module, these non-imported modules must be specified in the link command line. For example, if the main module imports module MA to call the function FA, which in turn calls a function FB from module MB, but MA does not import MB, you will have to specify MB in the link line to have the linker resolve the FB function.

When linking a .42r program by using a .42x library, if none of the functions of a module in the .42x library are used in the program, the complete module is excluded by the linker. This may cause undefined function errors at runtime, such as when a function is only used in a dynamic call (an initialization function, for example.)

The following case illustrates this behavior:

```
$ cat x1.4gl
function fx11()
end function
function fx12()
end function

$ cat x2.4gl
function fx21()
end function
function fx22()
end function

$ cat prog.4gl
main
  call fx11()
end main

$ fglcomp x1.4gl
$ fglcomp x2.4gl
$ fglcomp prog.4gl

$ fgllink -o lib.42x x1.42m x2.42m

$ fgllink -o prog.42r prog.42m lib.42x
```

Here, module `x1.42m` (with functions `fx11` and `fx12`) will be referenced in the .42r program file, but functions of module `x2.42m` will not. At runtime, any dynamic call to functions `fx21()` or `fx22()` will fail with an un-trappable error -1338.

If you are using C-Extensions, you may need to use the -e option to specify the list of extension modules if the `IMPORT` keyword is not used:

```
$ fgllink -e extlib,extlib2,extlib3 -o stores.42r main.42m store.42m
```

Using makefiles

Most UNIX™ platforms provide the make utility program to compile projects. The make program is an interpreter of makefiles. These files contain directives to compile and link programs and/or generate other kind of files.

When developing on Microsoft™ Windows™ platforms, you may use the NMAKE utility provided with Visual C++, however this tool does not have the same behavior as the UNIX™ make program. To have a compatible make on Windows™, you can install a GNU make or third party UNIX™ tools such as Cygwin.

For more details about the make utility, see the platform-specific documentation.

The follow example shows a typical makefile for Genero applications:

```
#-----------------------------------------------------
# Generic makefile rules to be included in Makefiles
.SUFFIXES: .42s .42f .42m .42r .str .per .4gl .msg .hlp
FGLFORM=fglform -M
FGLCOMP=fglcomp -M
FGLLINK=fglrun -l
FGLMKMSG=fglmkmsg
FGLMKSTR=fglmkstr
FGLLIB=$$FGLDIR/lib/libfgl4js.42x
all::
.msg.hlp:
        $(FGLMKMSG) $*.msg $*.hlp
.str.42s:
        $(FGLMKSTR) $*.str $*.42s
.per.42f:
        $(FGLFORM) $*.per
.4gl.42m:
        $(FGLCOMP) $*.4gl
clean::
        rm -f *.hlp *.42? *.out
#----------------------------
# Makefile example
include Makeincl
FORMS=\
 customers.42f\
 orderlist.42f\
 itemlist.42f
MODULES=\
 customerInput.42m\
 zoomOrders.42m\
 zoomItems.42m
customer.42x: $(MODULES)
        $(FGLLINK) -o customer.42x $(MODULES)
all:: customer.42x $(FORMS)
```

Module build information

The compiler version used to build the 42m modules must be compatible to the runtime system used to execute the programs. The fglcomp compiler writes version information in the generated 42m files. This can be useful on site, if you need to check the version of the compiler that was used to build the 42m nodules.

To extract build information, run fglrun with the `-b` option:

```
$ fglrun -b mymodule.42m
  3.00.00 /home/devel/stores/mymodule.4gl 24
```

The output shows the following fields:

1. The Genero product version (`3.00.00`)
2. The full path of the source file (`/home/devel/stores/mymodule.4gl`)
3. The internal identifier of the p-code version (23)

fglrun -b can read the header of p-code modules compiled with older versions of fglcomp and display version information for such old modules. If fglrun cannot recognize a p-code module, it returns with an execution status that is different from zero.

When reading build information of a 42x or 42r file, fglrun scans all modules used to build the library or program. You will see different versions in the first column if the modules were compiled with different versions of fglcomp. However, it's not recommended that you mix versions on a production site:

```
$ fglrun -b myprogram.42r
  3.00.00 /home/devel/stores/mymodule1.4gl 24
  3.10.02 /home/devel/stores/mymodule2.4gl 24
  3.00.01 /home/devel/stores/mymodule3.4gl 24
```

To check if the version of the runtime system corresponds to the p-code version, run fglrun with the `-V` option:

```
$ fglrun -V
fglrun 3.00.00 internal-build-number
Genero virtual machine
Target l64xl212
...
```

If you need to write timestamp information in the p-code modules, you can use the `--timestamp` option of fglcomp:

```
$ fglcomp --timestamp mymodule.4gl
  $ fglrun -b mymodule.42m
  2008-12-24 11:22:33 2.11.05-1169.84 /home/devel/stores/mymodule.4gl 15
```

> **Important:** When using the `--timestamp` compiler option to write build timestamp information in p-code modules, you will not be able to easily compare 42m files (based on a checksum, for example): Without the timestamp, fglcomp generates exactly the same p-code module if the source file was not modified.

Source code edition

Simple helper to better render sources in configurable text editors.

These topics concern source code editing. You are free to use your preferred source code editor to write your programs.

- Choosing the correct locale on page 1516
- Avoid Tab characters in screen layouts on page 1517
- Code completion and syntax highlighting with VIM on page 1517

Choosing the correct locale

Before starting to edit source files, you must identify and configure the editor with the locale (character set) you want to use in your sources.

The language supports single-byte and multibyte character sets. When developing multilingual applications, we recommend that you write .per and .4gl source files in ASCII, and externalize language-dependent messages in string resource files.

Avoid Tab characters in screen layouts

When editing .per form files, avoid using Tab characters in sources, especially in the LAYOUT or SCREEN sections of forms. Different kinds of text or source code editors can expand Tab characters differently, according to the configuration settings. As a result, if two programmers are using different Tab expansion settings, the form layout will display in different ways. If used in a grid area, a Tab character will be interpreted as 8 blanks by fglform.

It is legal to use Tab characters in the rest of the .per file or .4gl sources (for example, to indent the code).

Code completion and syntax highlighting with VIM

If you are using the vim editor, automatic code completion and syntax highlighting is supported by fglcomp and fglform compilers.

In order to use auto completion with vim, you need at least vim version 7 with the Omni Completion feature.

To get the benefit of this feature with the vim editor, do this:

> **Note:** In the next lines, *user-vim-dir* is `~/.vim` on Unix platforms, and `%USERPROFILE%\vimfiles` on Windows platforms. And *home-dir* is `$HOME` on Unix platforms, and `%USERPROFILE%` on Windows platforms.

1. Copy `$FGLDIR/lib/fglcomplete.vim` into the *`user-vim-dir`*`/autoload` directory.
2. Copy `$FGLDIR/lib/fgl.vim` into the *`user-vim-dir`*`/syntax` directory.
3. Copy `$FGLDIR/lib/per.vim` into the *`user-vim-dir`*`/syntax` directory.
4. Add the following lines to your VIM resrouces file (*`home-dir`*`/.vimrc` file):

```
autocmd Filetype fgl setlocal omnifunc=fglcomplete#Complete
autocmd Filetype per setlocal omnifunc=fglcomplete#Complete
syntax on
au BufNewFile,BufRead *.per setlocal filetype=per
```

You can now use automatic code completion; open a .4gl or .per file, start to edit the file, and when you are in vim insert mode, press `CTRL-X` + `CTRL-O`, to get a list of language elements to complete the instruction syntax or expression.

> **Note:** For convenience, `TAB` can also be used to get the completion list as with the `CTRL-X` + `CTRL-O` key combinations. However, `TAB` will only show the completion list, if the edit cursor in after a keyword: At the beginning of the line, `TAB` adds indentation characters.

For more details about vim, see http://www.vim.org.

Source documentation

Explains how to automatically generate documentation from your sources.

- Understanding source code documentation on page 1518
- Prerequisites for source documentation generation on page 1518
- Documentation structure on page 1518
- Adding comments to sources on page 1519
  - Commenting a function on page 1519
  - Commenting a module on page 1521
  - Commenting a package on page 1521
  - Commenting a project on page 1521
- Run the documentation generator on page 1521

Understanding source code documentation

Documenting sources is an important task in software development, to share the code among applications and achieve better re-usability.

Source documentation must be concise, clear, and complete. However, documenting sources can be boring and subject to mistakes if large repetitive documentation sections have to be written by hand.

Source documentation can be produced automatically with the fglcomp compiler. The compiler can generate source documentation from the .4gl files of your project with minimum effort. The resulting source documentation is generated in simple HTML format and can be published on a web server.

Source documentation is generated with the `--build-doc` option of fglcomp. To extract documentation from a .4gl source:

```
fglcomp --build-doc filename.4gl
```

You can generate default documentation from the existing sources. For a better description of the code, add special #+ comments in your sources to describe code elements such as functions, function parameters, and return values.

By default, only `PUBLIC` symbols are documented. If you want to include `PRIVATE` symbols, use the `--doc-private` option:

```
fglcomp --build-doc --doc-private filename.4gl
```

Prerequisites for source documentation generation

To generate the HTML pages, fglcomp first generates `.xa` files which must be converted to `.html` files. The conversion from `.xa` to `.html` is done with an XSLT processor using the `.xsl` style sheets files provided in `FGLDIR/lib/fgldoc/`

You must have an XSLT processor installed on the machine where the documentation is generated.

- On UNIX™, fglcomp runs the `FGLDIR/lib/fgldoc/Transform.sh` script to convert `.xa` files to `.html` files. Therefore you need the `xsltproc` command line XSLT processor (from the libxml package).
- On Windows™, fglcomp runs the `FGLDIR\lib\fgldoc\Transform.js` script to convert `.xa` files to `.html` files. To run the `Transform.js` script, you must have `cscript.exe` installed with the Microsoft™.XMLDOM class (this is the case on recent Windows™ versions).

  Note: If the default result of the transformation does not fit your needs, the style sheets provided in `FGLDIR/lib/fgldoc` can be adapted to generate different HTML files.

Documentation structure

The source documentation structure is based on the well-known Java-doc technique. The generated documentation reflects the structure of your sources; in order to have nicely structured source documentation, you must have a nicely structured source tree.

The source documentation elements are structured as follows (elements in *italic* must be created by hand, others are generated files):

- Top/root directory (the root of your project)
  - *`overview.4gl`* (description of the project)
  - `overview-summary.html`
  - `overview-frame.html`
  - `allclasses-frame.html`
  - `index-all.html`
  - `index.html`

- `fgldoc.css`
- *sub-directory1* (package)
  - *sub-directory11* (package)
  - ...
  - *sub-directory12* (package)
    - *sub-directory121* (package)
      - *`package-info.4gl`* (description of the package/directory)
      - `package-summary.html`
      - `package-frame.html`
      - *source1211.4gl* (module)
      - `source1211.html`
      - *source1212.4gl* (module)
        - `#+ Module 1212 comment` (description of the module)
        - `#+ Function 12121 comment` (description of the function)
        - function12121
        - `#+ Function 12121 comment`
        - function12122
        - ...
      - `source1212.html`
      - *source1213.4gl* (module)
      - `source1213.html`
      - ...
- *sub-directory2* (package)
- ...
- *sub-directory3* (package)
- ...

First create a file named `overview.4gl` in the top directory of the project. This file contains the overall description of the project. In that directory, the documentation generator creates the files `overview-summary.html`, `overview-frame.html`, `allclasses-frame.html`, `index-all.html`, `index.html` and `fgldoc.css` .

The documentation generator can scan sub-directories to build the documentation for a whole project; each source directory defines a package. For each directory (i.e. package), the generator creates a `package-summary.html` and a `package-frame.html` file. If a file with the name `package-info.4gl` exists, it will be scanned to complete the `package-summary.html` file with the package description.

The documentation generator creates a *filename* .html file for each .4gl source module, seen as a *class* in the documentation.

Adding comments to sources

Commenting a function

To comment a function, add some lines starting with #+, before the function body. The comment body is composed of paragraphs separated by a blank line. The first paragraph of the comment is a short description of the function. This description will be placed in the function summary table. The next paragraph is long text describing the function in detail. Other paragraphs must start with a tag to identify the type of the paragraph; a tag starts with the @ "at" sign.

Table 323: Supported @ tags

Tag	Description
`@code`	Indicates that the next lines show a code example using the function.
`@param` *`name description`*	Defines a function parameter identified by *name*, explained by a *description*. *name* must match the parameter name in the function declaration.
`@returnType` *`fglType`* `[,...]`	Defines the data type of the value returned by the function. If the function returns several values, write a comma-separated list of types.
`@return` *`description`*	Describes the values returned by the function. Several @return comment lines can be written.

Example

```
#+ Compute the amount of the orders for a given customer
#+
#+ This function calculates the total amount of all orders for
 the
#+ customer identified by the cust_id number passed as
 parameter.
#+
#+ @code
#+ DEFINE total DECIMAL(10,2)
#+ CALL total = ordersTotal(r_customer.cust_id)
#+
#+ @param cid Customer identifier
#+
#+ @returnType DECIMAL(10,2)
#+ @return The total amount as DECIMAL(10,2)
#+
FUNCTION ordersTotal(cid)
   DEFINE cid INTEGER
   DEFINE ordtot DECIMAL(10,2)
   SELECT SUM(ord_amount) INTO ordtot
        FROM orders WHERE orders.cust_id = cid
   RETURN ordtot
END FUNCTION
```

Commenting a report

To comment a report, add some lines starting with #+, before the report body. The comment body is composed of paragraphs separated by a blank line. The first paragraph of the comment is a short description of the report. This description will be placed in the function summary table. The next paragraph is long text describing the report in detail. Other paragraphs must start with a tag to identify the type of the paragraph; a tag starts with the @ "at" sign.

Table 324: Supported @ tags

Tag	Description
`@code`	Indicates that the next lines show a code example using the report.
`@param` *`name description`*	Defines a report parameter identified by *name*, explained by a *description*. *name* must match the parameter name in the report declaration.

Commenting a module

To comment a .4gl module, you can add #+ lines at the beginning of the source, before module element declarations such as module variable definitions.

Example

```
#+ This module implements customer information handling
#+
#+ This code uses the 'customer' and 'custdetail' database
 tables.
#+ Customer input, query and list handling functions are defined
 here.
#+
DEFINE r_cust RECORD
   cust_id INTEGER,
   cust_name VARCHAR(50),
   cust_address VARCHAR(200)
END RECORD
```

Commenting a package

To describe a complete directory (i.e. package), you must create a `package-info.4gl` file in the directory and add a #+ comment in the file. The comment will be added to the `package-summary.html` file.

Commenting a project

In the top directory of your sources, you must create a `overview.4gl` file with a #+ comment describing the project. This file is mandatory in order to generate the tree of HTML pages for an entire project, as it is used as the starting point by fglcomp.

Run the documentation generator

Follow this procedure to produce the source documentation.

Follow this procedure to produce the source documentation.

1. Go to the top directory of your sources.
2. Create a file named `overview.4gl`, with a #+ comment describing your project.
3. Go to the subdirectories and create files named `package-info.4gl` with a #+ comment describing the package.
4. Edit the 4gl modules to add #+ comments to functions that must be documented.
5. Go back to the top directory of your sources.
6. Run `fglcomp --build-doc overview.4gl`

Use the `-W apidoc` compiler option to get warnings for invalid comment tags. For example, when a @param tag is missing for a function parameter.

7. To test the result, load the generated `index.html` file in your preferred browser.

The preprocessor

A typical preprocessor like in the C language.

- Understanding the preprocessor on page 1522
- Compilers command line options on page 1522
- File inclusion on page 1523
- Simple macro definition on page 1525
- Function macro definition on page 1527
- Stringification operator on page 1529
- Concatenation operator on page 1529
- Predefined macros on page 1530
- Undefining a macro on page 1530
- Conditional compilation on page 1530

Understanding the preprocessor

The preprocessor is used to transform your sources before compilation. It allows you to include other files and to define macros that will be expanded when used in the source. It behaves similar to the C preprocessor, with some differences.

> **Important:** The preprocessor should be avoided if there is an alternative in the native language. For example, instead of defining program constants with a &define macro, use the `CONSTANT` instruction. Other language features such as `IMPORT FGL` increase code readability and modular programming, without the need of a preprocessor. The preprocessor might be desupported in a future version.

The preprocessor transforms files as follows:

- The source file is read and split into lines.
- Continued lines are merged into one long line if it is part of a preprocessor definition.
- Comments are not removed unless they appear in a macro definition.
- Each line is split into a list of lexical tokens.

The preprocessor implements the following features:

1. File inclusion
2. Conditional compilation
3. Macro definition and expansion

There are two kind of macros:

1. Simple macros
2. Function macros

If a preprocessing directive is invalid, the compilers will generate a .err file with the preprocessing error included in the source file at the line position where the problem exists. When using the `-M` option, preprocessor errors will be printed to stderr, like regular compiler errors.

Compilers command line options

Preprocessor options can be used with fglcomp and fglform compilers.

File inclusion path

The `-I` option defines a path used to search files included by the `&include` directives:

```
-I path
```

Macro definition

The `-D` option defines a macro with the value 1, so that it can be used conditional directives like `&ifdef`:

```
-D identifier
```

The `-U` option undefines a macro. The macro will not be defined, even if it is defined with the `-D` option later in the command line, or when it is defined in the code with a `&define` directive:

```
-U identifier
```

However, predefined macros such as `__LINE__` can't be undefined with the `-U` option.

Preprocessing only

```
-E
```

By using the `-E` option, only the preprocessing phase is done by the compilers. Result is dumped in standard output.

Preprocessing options

```
-p [nopp|noln|fglpp]
```

When using option `-p nopp`, it disables the preprocessor phase.

By using option `-p noln` with the `-E` preprocessing-only option, you can remove line number information and unnecessary empty lines.

By default, the preprocessor expects an ampersand '&' as preprocessor symbol for macros. The option `-p fglpp` enables the old syntax, using the sharp '#' as preprocessor symbol. The sharp '# ' syntax is not compatible with single-line comments.

Examples

```
fglcomp -E -D DEBUG -I /usr/sources/headers program.4gl
```

```
fglcomp -E -p fglpp -I /usr/sources/headers program.4gl
```

```
fglcomp -E -p nopp -I /usr/sources/headers program.4gl
```

File inclusion

The `&include` directive instructs the preprocessor to include a file.

Syntax

```
&include "filename"
```

1. *filename* is searched first in the directory containing the current file, then in the directories listed in the include path. (-I option). The file name can be followed by spaces and comments.

Usage

The included file will be scanned and processed before continuing with the rest of the current file.

Source: File A

```
First line
&include "B"
Third line
```

Source: File B

```
Second line
```

Result:

```
& 1 "A"
First line
& 1 "B"
Second line
& 3 "A"
Third line
```

These preprocessor directives inform the compiler of its current location with special preprocessor comments, so the compiler can provide the right error message when a syntax error occurs.

The preprocessor-generated comments use the following format:

```
& number "filename"
```

where:

- *number* is the current line in the preprocessed file
- *filename* is the current file name

Recursive inclusions

Recursive inclusions are not allowed. Doing so will fail and output an error message.

The following example is incorrect:

Source: File A

```
&include "B"
```

Source: File B

```
HELLO
&include "A"
```

```
fglcomp -M A.4gl output
```

```
B.4gl:2:1:2:1:error:(-8029) Multiple inclusion of the source file 'A'.
```

Including the same file several times is allowed:

Source: File A

```
&include "B"
&include "B"  -- correct
```

Source: File B

```
HELLO
```

Result:

```
& 1 "A"
& 1 "B"
HELLO
& 2 "A"
& 1 "B"
HELLO
```

Simple macro definition

A simple macro is identified by its name and body.

Syntax

```
&define identifier body
```

1. *identifier* is the name of the macro. Any valid identifier can be used.
2. *body* is any sequence of tokens until the end of the line.

After substitution, the macro definition is replaced with blank lines.

Usage

As the preprocessor scans the text, it substitutes the macro body for the name identifier.

The following example show macro substitution with 2 simple macros:

Source: File A

```
&define MAX_TEST 12
&define HW "Hello world"

MAIN
  DEFINE i INTEGER
  FOR i=1 TO MAX_TEST
    DISPLAY HW
  END FOR
END MAIN
```

Result:

```
& 1 "A"



MAIN
  DEFINE i INTEGER
  FOR i=1 TO 12
    DISPLAY "Hello world"
  END FOR
```

```
END MAIN
```

The macro definition can be continued on multiple lines, but when the macro is expanded, it is joined to a single line as follows:

Source: File A

```
&define TABLE_VALUES 1, \
                     2, \
                     3
DISPLAY TABLE_VALUES
```

Result:

```
& 1 "A"



DISPLAY 1, 2, 3
```

The source file is processed sequentially, so a macro takes effect at the place it has been written:

Source: File A

```
DISPLAY X
&define X "Hello"
DISPLAY X
```

Result:

```
& 1 "A"
DISPLAY X

DISPLAY "Hello"
```

The macro body is expanded only when the macro is applied:

Source: File A

```
&define AA BB
&define BB 12
DISPLAY AA
```

Result:

```
& 1 "A"


DISPLAY 12
```

- AA is first expanded to BB.
- The text is re-scanned and BB is expanded to 12.
- When the macro AA is defined, BB is not known yet; but it is known when the macro AA is used.

In order to prevent infinite recursion, a macro cannot be expanded recursively.

Source: File A

```
&define A B
&define B A
&define C C
```

```
A C
```

Result:

```
& 1 "A"



A C
```

- A is first expanded to B.
- B is expanded to A.
- A is not expanded again as it appears in its own expansion.
- C expands to C and can not be expanded further.

Note: It is also possible to define a macro with the `-D` command line option of compilers.

Function macro definition

Function macros are preprocessor macros which can take arguments.

Syntax

```
&define identifier( arglist ) body
```

1. *identifier* is the name of the macro. Any valid identifier can be used.
2. *body* is any sequence of tokens until the end of the line.
3. *arglist* is a list of identifiers separated with commas and optionally whitespace.
4. There must be no space or comment between the macro name and the opening parenthesis. Otherwise the macro is not a function macro, but a simple macro.

Usage

Function macros take arguments that are replaced in the body by the preprocessor.

Source: File A

```
&define function_macro(a,b) a + b
&define simple_macro (a,b) a + b
function_macro( 4 , 5 )
simple_macro (1,2)
```

Result:

```
& 1 "A"


4 + 5
(a,b) a + b (1,2)
```

A function macro can have an empty argument list. In this case, parenthesis are required for the macro to be expanded. As we can see in the next example, the third line is not expanded because it there is no '`()`' after `foo`. The function macro cannot be applied even if it has no arguments.

Source: File A

```
&define foo() yes
foo()
foo
```

Result:

```
& 1 "A"

yes
foo
```

The comma separates arguments. Macro parameters containing a comma can be used with parenthesis. In this example, the second line has been substituted, but th third line produced an error, because the number of parameters is incorrect.

Source: File A

```
&define one_parameter(a) a
one_parameter((a,b))
one_parameter(a,b)
```

```
fglcomp -M output
```

```
source.4gl:3:1:3:1:error:(-8039) Invalid number of parameters
 for macro one_parameter.
```

Macro arguments are completely expanded and substituted before the function macro expansion.

A macro argument can be left empty.

Source: File A

```
&define two_args(a,b) a b
two_args(,b)
two_args(,)
two_args()
two_args(,,)
```

```
fglcomp -M output
```

```
source.4gl:4:1:4:1:error:(-8039) Invalid number of parameters
 for macro two_args.
source.4gl:5:1:5:1:error:(-8039) Invalid number of parameters
 for macro two_args.
```

Macro arguments appearing inside strings are not expanded.

Source: File A

```
&define foo(x) "x"
foo(toto)
```

Result:

```
& 1 "A"

"x"
```

Stringification operator

Transforms a preprocessor macro element to a string.

Syntax

```
#param
```

1. *param* is a parameter of the macro

Usage

The stringification operator # converts a preprocessor macro parameter to a string.

When a macro parameter is used with a preceding #, it is replaced by a string containing the literal text of the argument.

The argument is not macro expanded before the substitution.

Source: File A

```
&define disp(x) DISPLAY #x
disp(abcdef)
```

Result:

```
& 1 "A"
DISPLAY "abcdef"
```

Concatenation operator

Concatenates two parameters of a preprocessor macro.

Syntax

```
token1 ## token2
```

1. *token1* is a parameter of the macro or a simple token.
2. *token2* is a parameter of the macro or a simple token.

Usage

The double-sharp operator ## can be used to merge two tokens while expanding a macro and create a single token.

All tokens can not be merged. Usually these tokens are identifiers, or numbers.

The concatenation result produces an identifier.

Source: File A

```
&define COMMAND(NAME) #NAME, NAME ## _command
COMMAND(quit)
```

Result:

```
& 1 "A"

"quit", quit_command
```

Predefined macros

The preprocessor predefines 2 macros:

1. `__LINE__` expands to the current line number. Its definition changes with each new line of the code.
2. `__FILE__` expands to the name of the current file as a string constant. Ex: "subdir/file.inc"

These macros are often used to generate error messages.

An `&include` directive changes the values of `__FILE__` and `__LINE__` to correspond to the included file.

Undefining a macro

Un-defines a preprocessor macro.

Syntax

```
&undef identifier
```

1. *identifier* is a preprocessor constant.

Usage

If a macro is redefined without having been undefined previously, the preprocessor issues a warning and replaces the existing definition with the new one. First un-define a macro with the `&undef` directive.

Source: File A

```
&define HELLO "hello"
DISPLAY HELLO
&undef HELLO
DISPLAY HELLO
```

Result:

```
& 1 "A"

DISPLAY "hello"
DISPLAY HELLO
```

> **Note:** It is also possible to undefine a macro with the `-U` command line option of compilers. However, predefined macros can't be undefined with this option.

Conditional compilation

Integrate code lines conditionally.

Syntax 1

```
&ifdef identifier
...
[&else
...]
&endif
```

Syntax 2

```
&ifndef identifier
...
[&else
...]
&endif
```

1. *identifier* is a preprocessor constant.

Usage

The `&ifdef` and `&ifndef` preprocessor macros can be used to integrate code lines conditionally according to the existence of a preprocessor constant.

The constant is defined with a `&define` or with the `-D` option in the command line.

Even if the condition is evaluated to false, the content of the `&ifdef` block is still scanned and tokenized. Therefore, it must be lexically correct.

Sometimes it is useful to use some code if a macro is not defined. You can use `&ifndef`, that evaluates to true if the macro is not defined.

Source: File A

```
&define IS_DEFINED
&ifdef IS_DEFINED
DISPLAY "The macro is defined"
&endif  /* IS_DEFINED */
```

Result:

```
& 1 "A"


DISPLAY "The macro is defined"
```

The debugger

Describes the command-line debugger to find easily bugs in your programs.

- Understanding the debugger on page 1531
- Prerequisites to run the debugger on page 1532
- Starting fglrun in debug mode on page 1532
- Attaching to a running program on page 1533
- Debugging on a mobile device on page 1534
- Stack frames in the debugger on page 1535
- Setting a breakpoint programmatically on page 1536
- Expressions in debugger commands on page 1536
- Debugger commands on page 1537

Understanding the debugger

The debugger is a feature built in the runtime system (`fglrun`) that allows you to control the execution of a program step by step, so that you can find logical and runtime errors.

There are three debug modes possible with the Genero runtime system:

1. Start the fglrun program from the command line with the `-d` option. For more details, see Starting fglrun in debug mode on page 1532.
2. Attaching with the `fgldb` tool, to a running fglrun process, for debugging through a TCP socket. For more details, see Attaching to a running program on page 1533.
3. Connect directly with the `fgldb` tool, to the debug TCP port of a runtime system running on a mobile device in standalone mode. For more details, see Debugging on a mobile device on page 1534.

The debugger supports a subset of the standard GNU C/C++ debugger called gdb.

In command line mode, the debugger shows the following prompt

```
(fgldb)
```

A command is a single line of input. It starts with a command name, which may be followed by arguments whose meaning depends on the command name. For example, the command `step` accepts as an argument the number of times to step:

```
(fgldb) step 5
```

You can use command abbreviations. For example, the 'step' command abbreviation is 's':

```
(fgldb) s 5
```

Possible command abbreviations are shown in the command's syntax.

A blank line as input to the debugger (pressing just the RETURN or ENTER keys) usually causes the previous command to repeat. However, commands whose unintentional repetition might cause problems will not repeat in this way.

Prerequisites to run the debugger

Before starting the debugger, make sure you have properly set the FGLSOURCEPATH environment variable, to let the debugger find the source files.

UNIX™ example:

```
$ FGLSOURCEPATH="/usr/app/source:/home/scott/sources"
$ export FGLSOURCEPATH
```

Windows™ example:

```
C:\> set FGLSOURCEPATH=C:\app\sources;C:\scott\sources
```

By default, if FGLSOURCEPATH is not defined, the debugger searches for sources in the current directory and in directories defined by FGLLDPATH.

Make sure that the following FGLPROFILE entry is not define or defined as false:

```
fglrun.ignoreDebuggerEvent = false
```

Starting fglrun in debug mode

To start the fglrun runtime system in debug mode, use the `-d` option of fglrun, for example:

```
fglrun -d myprog
```

This mode is typically used in development environments when using the command line tools.

The debugger can be used from the command line shell, but can also be called from a graphical debugging tool that understands the debugging commands of `fglrun -d`. The syntax of the commands is similar to the gdb debugger.

The debugger can for example be used alone in the command line mode or with a graphical shell compatible with gdb such as `ddd`:

```
ddd --debugger "fglrun -d myprog"
```

Attaching to a running program

Basics

Use the `fgldb` command with the `-p` option to switch the runtime system into debug mode when an application is running on a server.

> **Note:** The `fgldb` command must be executed on the machine where the fglrun process executes.

The `fgldb` command line tool takes the fglrun-bin (Unix) / fglrun.exe (Windows) process id as value for the `-p` argument.

> **Note:** Before starting a debug session, make sure that you fulfill the prerequisites for debugging.

Debug a program running on a UNIX server

First, identify the process id of the fglrun-bin or fglrun.exe program running on your server.

For example, on a Unix platform, use the `ps` command:

```
$ ps a | grep fglrun-bin
10646 pts/0    S+     0:00 /opt/myapp/fgl/lib/fglrun-bin stockinfo.42m
```

> **Note:** Inspect the GAS log files to find the id of an fglrun process running behind a GAS application server. Enable full log reports in the GAS to get detailed information about process execution.

You may want to debug processes that use a lot of machine resources (processor, memory or open files). Use a system utility to find a process id by resources used (e.g., the `top` command on Linux).

Execute the fgldb tool with the process id of the program you want to attach to:

```
$ fgldb -p 10646
108     DISPLAY ARRAY contlist TO sr.*
(fgldb)
```

The `(fgldb)` prompt indicates that you are now connected to the fglrun process, and the program flow is suspended. To continue with the program flow, enter the "`continue`" debugger command:

```
(fgldb) continue
Continuing.
```

The application will then resume. To suspend the program again and enter debugger commands, press CTRL-C in the debug console. `fgldb` will display the interrupt message and return control to the debugger:

```
...
Continuing.
^CINTERRUPT
108     DISPLAY ARRAY contlist TO sr.*
(fgldb)
```

At this point, you can enter debugger commands. For example, set a break point and continue until the break point is reached:

```
(fgldb) b 427
Breakpoint 2 at 0x00000000: file contacts.4gl, line 427.
(fgldb) continue
Continuing.
Breakpoint 2, edit_contact() at contacts.4gl:427
427     IF new THEN
(fgldb)
```

To finish the debug session, close the connection with the "`detach`" debugger command:

```
(fgldb) detach
Connection closed by foreign host.
```

Debugging on a mobile device

Basics

When an app was created with debug mode and is running in on a device, it is possible to switch the runtime system in debug mode, by using the `fgldb` command tool with the `-m` option.

> **Important:** The app must have be created in debug mode. Apps created in release mode cannot be debugged with the `fgldb` tool. For more details, check how to build mobile apps with debug mode in the Deploying mobile apps on page 2572 section.
>
> **Important:** On iOS devices, after installing the app, you need to enable the debug port in the app settings, otherwise the app will not listen to the debug port.

The `fgldb` command line tool takes two arguments: The host (or IP address) of the mobile device, and an optional TCP port number to connect to. For mobile devices, the debug TCP port is `6400`. Note that this is the same port the mobile front-end is listening to for GUI connection, when working in GUI client/server mode.

> **Note:** Before starting a debug session, make sure that you fulfill the prerequisites for debugging.

Debugging an app running on a physical device

Considering the mobile device IP address is "`192.168.1.23`", and the application is running locally on a physical mobile device, you can open a debug session from the development machine as follows:

```
$ fgldb -m 192.168.1.23:6400
108     DISPLAY ARRAY contlist TO sr.*
(fgldb)
```

The `(fgldb)` prompt indicates that you are now connected to the fglrun process on mobile device, and the program flow is suspended. To continue with the program flow, enter the "`continue`" debugger command:

```
(fgldb) continue
Continuing.
```

The application will then resume on the mobile device. To suspend the program again and enter debugger commands, press CTRL-C in the debug console: `fgldb` will show the interrupt message and give you the control back:

```
...
Continuing.
```

```
^CINTERRUPT
108     DISPLAY ARRAY contlist TO sr.*
(fgldb)
```

At this point, you can for example set a break point and continue until the break point is reached:

```
(fgldb) b 427
Breakpoint 2 at 0x00000000: file contacts.4gl, line 427.
(fgldb) continue
Continuing.
Breakpoint 2, edit_contact() at contacts.4gl:427
427     IF new THEN
(fgldb)
```

To finish the debug session, close the connection with the "`detach`" debugger command:

```
(fgldb) detach
Connection closed by foreign host.
```

Debugging an app running on an Android device emulator

When the mobile application is executing on an Android device emulator in the same machine as the development environment, you must first redirect the 6400 TCP port.

First you must connect to the emulator terminal, using the telnet TCP port 5554:

```
$ telnet localhost 5554
```

When connected on the device emulator, redirect the port 6400 as follows:

```
$ redir add tcp:6400:6400
$ quit
```

You may also want to redirect the port 6480, to be able to show GMA service debug information from a browser with the http://localhost:6480 URL:

```
$ redir add tcp:6480:6480
```

Finally, quit the device emulator telnet session with:

```
$ quit
```

Stack frames in the debugger

Each time your program performs a function call, information about the call is saved in a block of data called a *stack frame*. Each frame contains the data associated with one call to one function.

The stack frames are allocated in a region of memory called the *call stack*. When your program is started, the stack has only one frame, that of the function `main`. This is the initial frame, also known as the *outermost frame*. As the debugger executes your program, a new frame is made each time a function is called. When the function returns, the frame for that function call is eliminated.

The debugger assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are assigned by the debugger to allow you to designate stack frames in commands.

Each time your program stops, the debugger automatically selects the currently executing frame and describes it briefly. You can use the frame command to select a different frame from the current call stack.

Setting a breakpoint programmatically

You can set a breakpoint in the program source code with the BREAKPOINT instruction. If the program flow encounters this instruction, the program stops as if the break point was set by the break command:

```
MAIN
  DEFINE i INTEGER
  LET i=123
  BREAKPOINT
  DISPLAY i
END MAIN
```

The BREAKPOINT instruction is simply ignored when running in normal mode.

Expressions in debugger commands

Some debugger commands such as display take an expression as argument. The Genero debugger supports a reduced syntax for command expressions described in this section. For a detailed description of comparison operators, constant values and operands, see Expressions.

Syntax

```
  variable
| char-const
| int-const
| dec-const
| NULL
| TRUE
| FALSE
| expression IS [NOT] NULL
| expression = expression
| expression == expression
| expression <= expression
| expression => expression
| expression < expression
| expression > expression
| expression + expression
| expression - expression
| expression * expression
| expression / expression
| expression OR expression
| expression AND expression
| NOT expression
| - expression
| ( expression )
```

Note:

1. *variable* is a program variable name.
2. *char-const* is character string literal delimited by single or double quotes.
3. *int-const* is an integer literal.
4. *dec-const* is a decimal number literal.
5. *expression* is a combination of one or more listed syntax elements.

Example

```
(fgldb) display a + 1000
1: a = 1140.50
```

Debugger commands

Table 325: Summary of debugger commands

Command	Description
`backtrace / where`	Print a summary of how your program reached the current state (back trace of all stack frames).
`break`	Set a break point at the specified line or function.
`call`	Call a function in the program.
`clear`	Clear breakpoint at some specified line or function.
`continue`	Continue program being debugged.
`define`	Define a new command name.
`delete`	Delete some breakpoints or auto-display expressions.
`detach`	Closes a remote debug connection.
`disable`	Disable some breakpoints.
`display`	Print the values of expression *EXP* each time the program stops.
`down`	Select and print the function called by the current function.
`echo`	Print the specified text.
`enable`	Re-activate breakpoints that have previously been disabled.
`finish`	Execute until selected stack frame returns.
`frame`	Select and print a stack frame.
`help`	Print list of debugger commands.
`ignore`	Set ignore-count of a breakpoint number N to COUNT.
`info`	Provide information about the status of the program.
`list`	List specified function or line.
`next`	Step program; continue with the next source code line at the same level.
`output`	Print the current value of the specified expression; do not include value history and do not print newline.
`print`	Print the current value of the specified expression.
`ptype`	Print the type of a variable
`quit`	Exit the debugger.
`run`	Start the debugged program.

Command	Description
set	Evaluate an expression and assign the result to a variable.
source	Execute a file of debugger commands.
signal	Continue program giving it the signal specified by the argument.
step	Step program until it reaches a different source line.
tbreak	Set a temporary breakpoint.
tty	Set terminal for future runs of program being debugged.
undisplay	Cancel some expressions to be displayed when the program stops.
until	Continue running until a specified location is reached.
up	Select and print the function that called the current function.
watch	Set a watchpoint for an expression. A watchpoint stops the execution of your program whenever the value of an expression changes.
whatis	Prints the data type of a variable.

backtrace / where
The `backtrace` commands prints a summary of how your program reached the current state.

Syntax

```
backtrace
```

Usage

The `backtrace` command prints a summary of your program's entire stack, one line per frame. Each line in the output shows the frame number and function name.

`bt` and `where` are aliases for the `backtrace` command.

Example

```
(fgldb) backtrace
#1 addcount() at mymodule.4gl:6
#2 main() at mymodule.4gl:2
(fgldb)
```

break
The `break` command defines a break point to stop the program execution at a given line or function.

Syntax

```
break [ { [module.]function
```

```
      | [module:]line } ]
          [ if condition ]
```

1. *function* is a function name.
2. *module* is the name of a specific source file, without extension.
3. *line* is a source code line.
4. *condition* is an expression evaluated dynamically.

Usage

The `break` command sets a break point at a given position in the program.

When the program is running, the debugger stops automatically at breakpoints defined by this command.

If a *condition* is specified, the program stops at the breakpoint only if the *condition* evaluates to TRUE.

If you do not specify any location (function or line number), the breakpoint is created for the current line. For example, if you write "break if var = 1", the debugger adds a conditional breakpoint for the current line, and the program will only stop if the variable is equal to 1 when reaching the current line again.

Example

```
(fgldb) break mymodule:5
Breakpoint 2 at 0x00000000: file mymodule.4gl, line 5.
```

call

The `call` command calls a function in the program.

Syntax

```
call function-name ( [ expression [,...] ] )
```

1. *function-name* is the name of the function to call.
2. *expression* is a combination of variables, constants and operators.

Usage

The `call` command invokes a function of the program and returns the control to the debugger.

The return values of the function are printed as a comma-separated list delimited by curly braces.

Example

```
MAIN
  DEFINE i INTEGER

  LET i = 1
  DISPLAY i
END MAIN

FUNCTION hello ()
  RETURN "hello", "world"
END FUNCTION
```

```
(fgldb) br main
Breakpoint 1 at 0x00000000: file t.4gl, line 4.
(fgldb) run
Breakpoint 1, main() at t.4gl:4
```

```
4         LET i = 1
(fgldb) call hello()
$1 =  { "hello" , "world" }
(fgldb)
```

clear
The `clear` command clears the breakpoint at a specified line or function.

Syntax

```
clear [ { function | [ module: ] line } ]
```

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.

Usage

With the `clear` command, you can delete breakpoints according to where they are in your program.

Use the `clear` command with no arguments to delete any breakpoints at the next instruction to be executed in the selected stack frame.

Use the delete command to delete individual breakpoints by specifying their breakpoint numbers.

Example

```
(fgldb) clear mymodule:5
Deleted breakpoint 2
(fgldb)
```

continue
The `continue` command continues the execution of the program after a breakpoint.

Syntax

```
continue [ignore-count]
```

1. *ignore-count* defines the number of times to ignore a breakpoint at this location.

Usage

The `continue` command continues the execution of the program until the program completes normally, another breakpoint is reached, or a signal is received.

`c` is an alias for the `continue` command.

Example

```
(fgldb) continue
...
(program output)
...
Program exited normally.
```

define
The `define` command allows you to specify a user-defined sequence of commands.

Syntax

```
define command-name
command
 [...]
end
```

1. *command-name* is the name assigned to the command sequence.
2. *command* is a valid debugger command.
3. **end** indicates the end of the command sequence.

Usage

The `define` command allows you to create a user-defined command by assigning a command name to a sequence of debugger commands that you specify. You may then execute the command that you defined by entering the command name at the debugger prompt.

User commands may accept up to ten arguments separated by white space.

Example

```
(fgldb) define myinfo
> info breakpoints
> info program
> end
(fgldb)
```

delete
The `delete` command allows you to remove breakpoints that you have specified in your debugger session.

Syntax

```
delete breakpoint
```

1. *breakpoint* is the number assigned to the breakpoint by the debugger.

Usage

The `delete` command allows you to remove breakpoints when they are no longer needed in your debugger session.

If you prefer you may disable the breakpoint instead, see the `disable` command.

`d` is an alias for the `delete` command.

Example

```
(fgldb) delete 1
(fgldb) run
Program exited normally.
(fgldb)
```

detach
The `detach` command closes the TCP connection of a remove debug session.

Syntax

```
detach
```

Usage

The `detach` command must be used to terminate a remove debug session, by closing the debug TCP connection.

Example

```
(fgldb) detach
```

disable
The `disable` command disables the specified breakpoint.

Syntax

```
disable breakpoint
```

1. *breakpoint* is the number assigned to the breakpoint by the debugger.

Usage

The `disable` command instructs the debugger to ignore the specified breakpoint when running the program.

Use the `enable` command to reactivate the breakpoint for the current debugger session.

Example

```
(fgldb) disable 1
(fgldb) run
Program exited normally.
(fgldb)
```

display
The `display` command displays the specified expression's value each time program execution stops.

Syntax

```
display expression
```

1. *expression* is a combination of variables, constants and operators.

Usage

The `display` command allows you to add an expression to an automatic display list. The values of the expressions in the list are printed each time program execution stops. Each expression in the list is assigned a number to identify it.

This command is useful in tracking how the values of expressions change during the program's execution.

Example

```
(fgldb) display a
1: a = 6
(fgldb) display i
2: i = 1
(fgldb) step
2: i = 1
1: a = 6
16       for i = 1 to 10
(fgldb) step
2: i = 2
1: a = 6
17       let a = a+1
(fgldb)
```

down
The down command moves down in the call stack.

Syntax

```
down
```

Usage

The down command moves the focus of the debugger down from the frame currently being examined, to the frame of its callee.

The command selects and prints the function called by the current function.

See stack frames for a brief description of frames.

```
(fgldb) down
#0 query_cust() at custquery.4gl:22
22    CALL cleanup()
(fgldb)
```

echo
The echo command prints the specified text as prompt.

Syntax

```
echo text
```

1. *text* is the specific text to be output.

Usage

The echo command allows you to generate exactly the output that you want.

Special characters can be included in text using C escape sequences, such as '\n' to print a newline. No newline is printed unless you specify one. In addition to the standard C escape sequences, a backslash followed by a space stands for a space. A backslash at the end of text can be used to continue the command onto subsequent lines.

Example

```
(fgldb) echo hello\n
hello
(fgldb)
```

enable
The `enable` command enables breakpoints that have previously been disabled.

Syntax

```
enable breakpoint
```

1. *breakpoint* is the number assigned to the breakpoint by the debugger.

Usage

The `enable` command allows you to reactivate a breakpoint in the current debugger session.

The breakpoint must have been disabled using the `disable` command.

Example

```
(fgldb) disable 1
(fgldb) run
Program exited normally.
(fgldb) enable 1
(fgldb) run
Breakpoint 1, at mymodule.4gl:5
```

finish
The `finish` command continues the execution of a program until the current function returns normally.

Syntax

```
finish
```

Usage

The `finish` command instructs the program to continue running until just after the function in the selected stack frame returns, and then stop.

The returned value, if any, is printed.

Example

```
(fgldb) finish
Run till exit myfunc() at module.4gl:10
Value returned is $1 = 123
(fgldb)
```

frame
The `frame` command selects and prints a stack frame.

Syntax

```
frame [ number ]
```

1. *number* is the stack frame number of the frame that you wish to select.

Usage

The `frame` command allows you to move from one stack frame to another, and to print the stack frame that you select. Each stack frame is associated with one call to one function within the currently executing program. Without an argument, the current stack frame is printed.

See stack frames for a brief discussion of frames.

Example

```
(fgldb) frame
#0 query_cust() at testquery.4gl:42
(fgldb)
```

help
The `help` command provides information about debugger commands.

Syntax

```
help [command]
```

1. *command* is the name of the debugger command for which you wish information.

Usage

The `help` command displays a short explanation of a specified command.

Enter the `help` command with no arguments to display a list of debugger commands.

Example

```
(fgldb) help delete
Delete some breakpoints or auto-display expressions
```

ignore
The `ignore` command defines the number of times a breakpoint must be ignored.

Syntax

```
ignore breakpoint count
```

1. *breakpoint* is the breakpoint number.
2. *count* is the number of times the breakpoint will be ignored.

Usage

The `ignore` command defines the number of times a breakpoint is ignored when the program flow reaches that breakpoint.

The next *count* times the breakpoint is reached, the program execution will continue, and no breakpoint condition is checked.

You can specify a *count* of zero to make the breakpoint stop the next time it is reached.

When using the continue command to resume the execution of the program from a breakpoint, you can specify a an ignore count directly as an argument.

Example

```
(fgldb) br main
Breakpoint 1 at 0x00000000: file t.4gl, line 4.
(fgldb) ignore 1 2
Will ignore next 2 crossings of breakpoint 1.
(fgldb) run           1
Program exited normally.
(fgldb) run           1
Program exited normally.
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4         LET i = 1
(fgldb)
```

info

The `info` command describes the current state of your program.

Syntax

```
info { breakpoints
|sources
| program
|  variables
|  locals
| files
| line { function
|        | module:line }
}
```

1. *function* is a function name of the program.
2. *module:line* defines a source code line in a module.

Usage

The `info` command describes the state of your program.

- `info breakpoints` lists the breakpoints that you have set.
- `info sources` prints the names of all the source files in your program.
- `info program` displays the status of your program.
- `info variables` displays global variables.
- `info locals` displays the local variables of the current function.
- `info files` lists the files from which symbols were loaded.
- `info line` *`function`* prints the program addresses for the first line of the function named *function*.
- `info line` *`module:line`* prints the starting and ending addresses of the compiled code for the source line specified. See the list command for all the ways that you can specify the source code line.

Example

```
(fgldb) info sources
Source files for which symbols have been read in:
mymodule.4gl, fglwinexec.4gl, fglutil.4gl, fgldialog.4gl,
 fgldummy4js.4gl
(fgldb)
```

list
The `list` command prints source code lines of the program being executed.

Syntax

```
list [ function
     | [module:]line]
```

Usage

The `list` command prints source code lines of your program, by default it begins with the current line.

Example

```
(fgldb) run
Breakpoint 1, at mymodule.4gl:5
5    CALL addlist()
(fgldb) list
5    CALL add_customer(cust_rec.*)
6    MESSAGE "Customer record was added"
...
14  END FUNCTION
(fgldb)
```

next
The `next` command continues running the program by executing the next source line in the current stack frame, and then stops.

Syntax

```
next [count]
```

1. *count* defines the number of lines to execute before stopping.

Usage

The `next` command allows you to execute your program one line of source code at a time. The `next` command is similar to `step`, but function calls that appear within the line of code are executed without stepping into the function code.

When the next line of code at the original stack level that was executing when you gave the `next` command is reached, execution stops.

Using a *count* parameter will repeat the `step` command *count* times.

After reaching a breakpoint, the `next` command can be used to examine a troublesome section of code more closely.

n is an alias for the `next` command.

Example

```
(fgldb) next
5    CALL add_customer(cust_rec.*)
(fgldb) next
6    MESSAGE "Customer record was added"
(fgldb) next 2
8    RETURN TRUE
```

output
The `output` command prints only the value of the specified expression, suppressing any other output.

Syntax

```
output expression
```

1. *expression* is a combination of variables, constants and operators.

Usage

The `output` command prints the current value of the expression and nothing else, no newline character, no "expr=", etc.

The usual output from the debugger is suppressed, allowing you to print only the value.

Example

```
(fgldb) output cust_rec.cust_id
87324(fgldb)
```

print
The `print` command displays the current value of the specified expression.

Syntax

```
print expression
```

1. *expression* is a combination of variables, constants and operators.

Usage

The `print` command allows you to examine the data in your program.

It evaluates and prints the value of the specified expression from your program, in a format appropriate to its data type.

p is an alias for the `print` command.

Example

```
(fgldb) print cust_rec.cust_id
$1 = 87324
(fgldb)
```

ptype
The `ptype` command prints the data type or structure of a variable.

Syntax

```
ptype variable-name
```

1. *variable-name* is the name of the variable.

Example

```
(fgldb) ptype cust_rec
type = RECORD
    cust_num INTEGER,
    cust_name VARCHAR(10),
    cust_address VARCHAR(200)
END RECORD
```

quit
The `quit` command terminates the debugger session.

Syntax

```
quit
```

Usage

The `quit` command allows you to exit the debugger.

`q` is an alias for the `quit` command.

Example

```
(fgldb) quit
```

run
The `run` command starts the program.

Syntax

```
run [argument [...] ]
```

1. *argument* is an argument to be passed to the program.

Usage

The `run` command causes your program to execute until a breakpoint is reached or the program terminates normally.

Example

```
(fgldb) run a b c
Breakpoint 1, at mymodule.4gl:3
3      CALL add_cust(cust_rec.*)
```

```
(fgldb)
```

set
The `set` command allows you to configure your debugger session and change program variable values.

Syntax

```
set { annotate {1|0}
    | environment envname[=value]
    | prompt ptext
    | set print elements elemcount
    | variable varname=expression
    | verbose {on|off}
    }
```

1. *ptext* is the string to which the prompt should be set.
2. *varname* is the program variable to be set to *expression*.
3. *expression* is a combination of variables, constants and operators.
4. *envname* is the environment variable to be set to *value*.
5. *elemcount* is the number of elements to define.

Usage

The `set` command allows to change program variables and/or debug environment settings.

`set variable` sets an program variable, to be taken into account when continuing program execution. The right operand can be an expression.

`set prompt` changes the prompt text. The text can be set toady string. A space is not automatically added after the prompt string, allowing you to determine whether to add a space at the end of the prompt string.

`set environment` sets an environment variable, where *value* may be any string. If the *value* parameter is omitted, the variable is set to a null value. The variable is set for your program, not for the debugger itself.

`set verbose on` forces the debugger to display additional messages about its operations, allowing you to observe that it is still working during lengthy internal operations.

`set annotate 1` switches the output format of the debugger to be more machine readable (this command is used by GUI front-ends like ddd or xxgdb)

`set print elements elemcount` defines the maximum number of array elements to be printed by the debugger when displaying a program array.

Example

```
(fgldb) set prompt ($)
($)
```

On UNIX™ systems, if your SHELL variable names a shell that runs an initialization file, any variables you set in that file affect your program. You may wish to move setting of environment variables to files that are only run when you sign on, such as .login or .profile.

source
The `source` command executes a file of debugger commands.

Syntax

```
source cmdfile
```

1. *cmdfile* is the name of the file containing the debugger commands.

Usage

The `source` command allows you to execute a command file of lines that are debugger commands.

The lines in the file are executed sequentially.

The commands are not printed as they are executed, and any messages are not displayed.

Commands are executed without asking for confirmation.

An error in any command terminates execution of the command file.

Example

Using the text file `cmdfile.txt`, which contains the single line with a `break` command:

```
$ cat cmdfile.txt
break 10

$ fglrun -d myprog
(fgldb) source cmdfile.txt
Breakpoint 2 @ 0x00000000: file mymod.4gl, line 10.
(fgldb)
```

signal
The `signal` command sends an interruption signal to the program.

Syntax

```
signal signal
```

Usage

The `signal` comment resumes execution where your program stopped, but immediately give it the signal *signal*.

signal can be the name or the number of a signal.

For example, on many systems signal 2 and signal SIGINT are both ways of sending an interrupt signal. The `signal SIGINT` command resumes execution of your program where it has stopped, but immediately sends an interrupt signal. The source line that was current when the signal was received is displayed.

Note: The current version only allows then signal `SIGINT`.

Example

```
(fgldb) signal SIGINT
Program exited normally.
16      for i = 1 to 10
```

```
(fgldb)
```

step
The `step` command continues running the program by executing the next line of source code, and then stops.

Syntax

```
step [count]
```

1. *count* defines the number of lines to execute before stopping.

Usage

The `step` command allows you to "step" through your program, executing one line of source code at a time.

When a function call appears within the line of code, that function is also stepped through.

A common technique is to set a breakpoint prior to the section or function that is causing problems, run the program till it reaches the breakpoint, and then step through it line by line.

Using a *count* parameter will repeat the `step` command *count* times.

`s` is an alias for the `step` command.

Example

```
(fgldb) step
4     CALL add_customer(cust_rec.*)
(fgldb) step 2
6     MESSAGE "Customer record was added"
```

tbreak
The `tbreak` command sets a temporary breakpoint.

Syntax

```
tbreak [ { function | [ module: ] line } ] [ if condition ]
```

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.
4. *condition* is an expression evaluated dynamically.

Usage

The `tbreak` command sets a breakpoint for one stop only.

The breakpoint is set in the same way as with the `break` command, but the breakpoint is automatically deleted after the first time your program stops there.

If a *condition* is specified, the program stops at the breakpoint only if the *condition* evaluates to true.

If you do not specify any location (function or line number), the breakpoint is created for the current line. For example, if you write "`tbreak if var = 1`", the debugger adds a conditional breakpoint for the current line, and the program will only stop if the variable is equal to 1 when reaching the current line again.

Example

```
(fgldb) tbreak 12
Breakpoint 2 at 0x00000000: file custmain.4gl, line 12.
(fgldb)
```

tty
The `tty` command resets the default program input and output for future run commands.

Syntax

```
tty filename
```

1. *filename* is the file which is to be the default for program input and output.

Usage

The `tty` command instructs the debugger to redirect program input and output to the specified file for future run commands.

The redirection is for your program only; your terminal is still used for debugger input and output.

Example

```
(fgldb) tty /dev/ttyS0
(fgldb)
```

undisplay
The `undisplay` command cancels expressions to be displayed when the program execution stops.

Syntax

```
undisplay itemnum
```

1. *itemnum* is the number of the expressions for which the display is cancelled.

Usage:

When the `display` command is used, each expression displayed is assigned an item number.

The `undisplay` command allows you to remove expressions from the list to be displayed, using the item number to specific the expression to be removed.

Example

```
(fgldb) step
2: i = 2
1: a = 20
9     FOR  i = 1 TO 10
(fgldb) undisplay 2
(fgldb) step
1: a = 20
10   LET cont = TRUE
(fgldb)
```

until
The `until` command continues running the program until the specified location is reached.

Syntax

```
until [ { function | [ module: ] line } ]
```

1. *function* is a function name.
2. *module* is a specific source file.
3. *line* is a source code line.

Usage

The `until` command continues running your program until either the specified location is reached, or the current stack frame returns.

This command can be used to avoid stepping through a loop more than once.

Example

```
(fgldb) until add_customer()
```

up
The `up` command selects and prints the function that called this one, or the function specified by the frame number in the call stack.

Syntax

```
up [frames]
```

1. *frames* says how many frames up to go in the stack. The default is 1.

Usage

The `up` command moves towards the outermost frame, to frames that have existed longer. To print the function that called the current function, use the `up` command without an argument.

See stack frames for a brief description of frames.

Example

```
(fgldb) up
#1 main() at customain.4gl:14
14    CALL query_cust()
(fgldb)
```

watch
The `watch` command sets a watchpoint for an expression.

Syntax

```
watch expression [if boolean-expression]
```

1. *expression* is a combination of variables, constants and operators.
2. *boolean-expression* is an optional boolean expression.

Usage

The `watch` command stops the program execution when the value of the expression changes.

If *boolean-expression* is provided, the `watch` command stops the execution of the program if the expression value has changed and the *boolean-expression* evaluates to true.

The watchpoint cannot be set if the program is not in the context where *expression* can be evaluated. Before using a watchpoint, you typically set a breakpoint in the function where the *expression* makes sense, then you run the program, and then you set the watchpoint. This example illustrates this procedure.

Example

```
MAIN
   DEFINE i INTEGER

   LET i = 1
   DISPLAY i
   LET i = 2
   DISPLAY i
   LET i = 3
   DISPLAY i

 END MAIN
```

```
(fgldb) break main
breakpoint 1 at 0x00000000: file test.4gl, line 4
(fgldb) run
Breakpoint 1, main() at test.4gl:4
4      LET i = 1
(fgldb) watch i if i >= 3
Watchpoint 1:  i
(fgldb) continue
1
2
Watchpoint 1:  i

Old value = 2
New value = 3
main() at t.4gl:9
9         DISPLAY i
(fgldb)
```

whatis

The `whatis` command prints the data type of a variable.

Syntax

```
whatis variable-name
```

1. *variable-name* is the name of the variable.

Usage

The `whatis` command can be used to show the data type of a program variable.

The program variable must exist in the current scope.

Example

```
(fgldb) run
Breakpoint 1, main() at t.4gl:4
4         LET i = 1
(fgldb) whatis i
type = INTEGER
(fgldb)
```

The profiler

Find out what function is the bottleneck in your program.

- Syntax of the program profiler on page 1556
- Usage on page 1556
  - Understanding the profiler on page 1556
  - Profiler output: Flat profile on page 1556
  - Profiler output: Call graph on page 1557
- Example on page 1558

Syntax of the program profiler

Start the fglrun tool with the `-p` option in order to activate the program profiler.

```
fglrun -p program[.42r] [argument [...]]
```

1. *program* is the name of the BDL program.
2. *argument* is a command line argument passed to the program.

Profiling statistics will be collected during program execution, and printed when the program ends.

Usage

Understanding the profiler

The profiler is a tool built in the runtime system that allows you to know where your program spends time, and which function calls which function.

The profiler can help to identify pieces of your program that are slower than expected.

In order to enable the profiler during the execution of a program, you must start fglrun with the **-p** option, for example:

```
fglrun -p myprog
```

When the program ends, the profiler dumps profiling information to standard error.

The times reported by the profiler can change from one execution to the other, depending on the available system resources. You better execute the program several times to get an average time.

The profiler does not support parent/child recursive calls, when a child function calls its parent function (i.e. Function P calls C which calls P again). In this case the output will show negative values, because the time spend in the parent function is subtracted from the time spend in the child function.

Profiler output: Flat profile

The section "flat profile" contains the list of the functions called while the programs was running. It is presented as a five-column table.

Table 326: Flat profile columns

Column Name	Description
`count`	number of calls for this function
`%total`	Percentage of time spent in this function. Includes time spent in subroutines called from this function.
`%child`	Percentage of time spent in the functions called from this function.
`%self`	Percentage of time spent in this function excluding the time spent in subroutines called from this function.
`name`	Function name

Note: 100% represents the program execution time.

Profiler output: Call graph

The section "Call graph" provides for each function:

1. The functions that called it, the number of calls, and an estimation of the percentage of time spent in these functions.
2. The functions called, the number of calls, and an estimation of the time that was spent in the subroutines called from this function.

Table 327: Call graph columns

Column name	Description
`index`	Each function has an index which appears at the beginning of its primary line.
`%total`	Percentage of time spent in this function. Includes time spent in subroutines called from this function.
`%self`	Percentage of time spent in this function excluding the time spent in subroutines called from this function.
`%child`	Percentage of time spent in the functions called from this function.
`calls/of`	Number of calls / Total number of calls
`name`	Function name

Output example:

```
index  %total  %self  %child  calls/of  name
```

```
...
      1.29    0.10    1.18      1/2        <-- main
     24.51    1.18   23.33      1/2        <-- fb
[4]     25.80    1.29   24.51        2        *** fc
     24.51    1.43   23.08      7/8        --> fa
```

Description:

- The three stars `***` indicate the function that is analyzed: `fb`.
- `fc` consumed 25.80% of the CPU time, 24.51% was in the called functions, 1.29% in the `fc` function code.
- `fc` has been called two times (one time by `main` and a second time by `fb`)
- `fc` has called the `fa` function 7 times.
- `fa` has been called 8 times in the program.

Example

Sample program

```
MAIN
  DISPLAY "Profiler sample"
  CALL fB()
  CALL fC(2)
END MAIN

FUNCTION fA(s,n_a)
  DEFINE s STRING
  DEFINE n_a,i INTEGER
  FOR i=1 TO n_a
    DISPLAY "fA "||s||" n:"||i
  END FOR
END FUNCTION

FUNCTION fB()
  CALL fA("fB",10)
  CALL fC(5)
END FUNCTION

FUNCTION fC(n_c)
  DEFINE n_c INTEGER
  WHILE n_c > 0
    CALL fA("fC",2)
    LET n_c=n_c-1
  END WHILE
END FUNCTION
```

Running the profiler

```
Flat profile (order by self)
   count  %total  %child   %self name
       25    88.0     0.0    88.0  rts_display
       72     6.3     0.0     6.3  rts_Concat
        8    85.4    82.0     3.4  fa
        2    25.8    24.5     1.3  fc
        8     0.3     0.0     0.3  rts_forInit
        1    85.6    85.4     0.2  fb
        1    99.9    99.6     0.3  main
```

```
Call gr
```

```
index   %total  %self  %child  calls/of  name
     12.69   12.69    0.00      1/25       <-- main
     75.29   75.29    0.00     24/25       <-- fa
[1]    87.98   87.98    0.00        25       *** rts_display
-------------------
     6.35    6.35    0.00     72/72       <-- fa
[2]     6.35    6.35    0.00        72       *** rts_Concat
-------------------
    60.90    2.02   58.88      1/8        <-- fb
     24.51    1.43   23.08      7/8        <-- fc
[3]    85.41    3.45   81.96        8        *** fa
     75.29   75.29    0.00     24/25        --> rts_display
      6.35    6.35    0.00     72/72        --> rts_Concat
      0.33    0.33    0.00      8/8         --> rts_forInit
-------------------
     1.29    0.10    1.18      1/2        <-- main
     24.51    1.18   23.33      1/2         <-- fb
[4]    25.80    1.29   24.51        2        *** fc
     24.51    1.43   23.08      7/8         --> fa
-------------------
     0.33    0.33    0.00      8/8        <-- fa
[5]     0.33    0.33    0.00        8        *** rts_forInit
-------------------
    85.61    0.20   85.41      1/1        <-- main
[6]    85.61    0.20   85.41        1        *** fb
     24.51    1.18   23.33      1/2         --> fc
     60.90    2.02   58.88      1/8         --> fa
-------------------
    99.94    0.35   99.59      1/1        <-- <top>
[7]    99.94    0.35   99.59        1        *** main
      1.29    0.10    1.18      1/2         --> fc
     85.61    0.20   85.41      1/1         --> fb
     12.69   12.69    0.00      1/25        --> rts_display
-------------------
```

Optimization

Programming tips and tricks to make your programs run faster.

- Runtime system basics on page 1560
  - Dynamic module loading on page 1560
  - Elements shared by multiple programs on page 1560
  - Elements shared by multiple modules on page 1560
  - Objects private to a program on page 1560
- Check runtime system memory leaks on page 1561
- Optimize your programs on page 1561
  - Finding program bottlenecks on page 1561
  - Optimizing SQL statements on page 1561
  - Passing small CHAR parameters to functions on page 1561
  - Compiler removes unused variables on page 1562
  - Saving memory by splitting modules on page 1562
  - Saving memory by using STRING variables on page 1562
  - Saving memory by using dynamic arrays on page 1562

Runtime system basics

Dynamic module loading

A Genero Business Development Language program is made of several 42m modules. Modules a linked together, or the dependency is defined with the `IMPORT FGL` instruction.

Except when using the debugger, modules are loaded dynamically when a module element (.i.e symbol) is required by the caller. For example, when executing a `CALL` instruction, the runtime system checks if the module of the function is already in memory. If not, the module is first loaded, then module variables are instantiated, and then the function is called.

Running programs are not affected by file replacements and will continue to run with an image of the module file that was originally loaded. However, replacing program modules during execution should be used with care: Since `.42m` modules are loaded dynamically on demand (when a symbol of the module is referenced), some modules may not yet be loaded, even if the program instance is already started. When replacing a module while programs are running, invalid symbol errors can occur if the module to be loaded does not correspond to the rest of the program modules that were loaded before the file replacements. See following scenario:

1. Program starts with V1 of `main.42m`, needing V1 of module `libutil.42m` (loaded later on demand).
2. Administrator upgrades application and installs `main.42m` and `libutil.42m` version V2.
3. Program running with V1 copy of `main.42m` calls a function from `libutil.42m`: runtime loads V2 of that module, while V1 is expected.

When live application updates are mandatory, consider installing new program and resource files (V2) in a different directory as the currently running version (V1), and use the FGLLDPATH and FGLRESOURCEPATH environment variables to point to the new files when starting a new (V2) program instance.

Note that on Windows™ platforms, program files currently in use cannot be overwritten, because of Windows™ OS memory mapping limitations. You need to turn off memory mapping with the FGLPROFILE entry `fglrun.mmapDisable`.

Elements shared by multiple programs

The (`.42m`) p-code module instructions and other elements such as constants are shared among several programs running on the same machine.

Localized string resource files (.42s) are also shared among all fglrun processes running on a computer.

These files are loaded with the system memory mapping facility, which allows multiple processes to access the same unique memory area.

Elements shared by multiple modules

By definition, global variables are visible to all modules of a program, and thus shared among all modules of the program. While global variables are an easy way to share data among multiple modules, it is not recommended that you use too many global variables.

The data type definitions are only defined once in memory and shared by all modules of a program instance. By data type definition we mean the type descriptions, not the data itself. This applies only to the equivalent data types used in different modules.

Objects private to a program

Program objects such as global variables, module variables as well as resources used by the user interface and SQL connections and cursors, are private to a program.

This implies that each of these objects requires private memory to be allocated. If memory is an issue, do not allocate unnecessary resources. For example, don't create windows / load forms or declare / prepare cursors until these are really needed by the program. When the resource is not longer needed, consider freeing them (`CLOSE WINDOW`, `FREE cursor`).

Check runtime system memory leaks

To improve the quality of the runtime system, fglrun supports the `-M` / `-m` options to count the creation of built-in class objects and some internal objects. This allows to check for memory leaks in the runtime system: The runtime system counts the object creations and destructions for each class. The right-most column of the output is the different between created and destroyed objects, it must show a zero for all type of objects.

The options described here are provided for debugging purpose only. The output format is subject of changes. These option can also be removed in a next version of the product.

```
$ fglrun -M stores.42r
FunctionI         :       10 -      10 =      0
Module            :        3 -       3 =      0
...
FieldType         :       19 -      19 =      0
```

The `-M` option displays memory counters at the end of the program execution.

The `-m` option checks for memory leaks, and displays memory counters at the end of the program execution if leaks were found.

Each line shows the number of objects allocated, and the number of objects freed. If the difference is not zero, there is a memory leak.

If you are doing automatic regression tests, we recommend that you run all your programs with `fglrun -m` to check for memory leaks in the runtime system.

Optimize your programs

This section lists some programming tips and tricks to optimize the execution of your application.

Finding program bottlenecks

The best way to find out why a program is slow (and also, to optimize an already fast-running program), it to use the profiler.

This tool is included in the runtime system, and generates a report that shows what function in your program is the most time-consuming.

Optimizing SQL statements

SQL statement execution is often the code part of the program that consumes a lot of processor, disk and network resources. Therefore, it is critical to pay attention to SQL execution.

Advice for this can be found in SQL Programming.

Passing small CHAR parameters to functions

Function parameters of most data types are passed by value (i.e. the value of the caller variable is copied on the stack, and then copied back into a local variable of the called function.) When large data types are used, this can introduce a performance issue.

For example, the following code defines a logging function that takes a CHAR(2000) as parameter:

```
FUNCTION log_msg( msg )
  DEFINE msg CHAR(2000)
  CALL myLogChannel.writeLine(msg)
END FUNCTION
```

If you call this function with a string having 19 bytes:

```
CALL log_msg( "Start processing..." )
```

When doing this call, the runtime system copies 19 characters on the stack, calls the function, and then copies the value into the local variable. Since the values in CHAR variables must always have a length matching the variable definition size, the runtime system fills the remaining 1981 positions with blanks. As result, each time you call this function, a 2000 characters long variable is created on the stack.

By using a VARCHAR(2000) (or a STRING) data type in this function, you optimize the execution because no trailing blanks need to be added.

Compiler removes unused variables

When declaring a large static array without any reference to that variable in the rest of the module, you will not see the memory grow at runtime. The compiler has removed its definition from the 42m module.

To get the defined variable in the 42m module, you must at least use it once in the source (for example, with a LET statement). Note that memory might only be allocated when reaching the lines using the variable.

Saving memory by splitting modules

Program modules (42m) are loaded dynamically on demand. If a program only needs some independent functions of a given module, all module resources will be allocated just to call these functions. By independent, we mean functions that do not use module objects such as variables defined outside function or SQL cursors. To avoid unnecessary resource allocation, you can extract these independent functions into another module and save a lot of memory at runtime.

If you are using 42x libraries, it is recommended that you create libraries with the 42m modules that belong to the same functionality group. For example, group all accounting modules together in an accounting library. By doing this, programmers using the 42x libraries are not dependent from module reorganizations.

Libraries are supported for backward compatibility, you should consider using the IMPORT FGL instruction to define module dependency and get modules loaded dynamically when needed.

Saving memory by using STRING variables

The CHAR and VARCHAR data types are provided to hold string data from a database column. When you define a CHAR or VARCHAR variable with a length of 1000, the runtime system must allocate the entire size, to be able to fetch SQL data directly into the internal string buffer.

For character string data that is not stored in the database, consider using the STRING data type. The STRING type is similar to VARCHAR, except that you don't need to specify a maximum length and the internal string buffer is allocated dynamically as needed. Thus, by default, a STRING variable initially requires just a bunch of bytes, and grows during the program life time, with a limitation of 65534 bytes.

A STRING variable should typically be used to build SQL statements dynamically, for example from a CONSTRUCT instruction. You may also use the STRING type for utility function parameters, to hold file names for example.

After a large STRING variable is used, it should be cleared with a LET or a INITIALIZE TO NULL instruction. However, this is only needed for STRING variables declared as global or module variables. The variables defined in functions will be automatically destroyed when the program returns from the function.

The base.StringBuffer build-in class should be used for heavy string manipulation and modifications. String data is not copied on the stack when an object of this class is passed to a function, or when the string is modified with class methods. This can have a big impact on performance when very large strings are processed.

Saving memory by using dynamic arrays

The language supports both static arrays and dynamic arrays. For compatibility reasons, static arrays must be allocated in their entirety. This can result in huge memory usage when big structures are declared, such as:

```
DEFINE my_array ARRAY[100,50] OF RECORD
```

```
      id CHAR(200),
      comment1 CHAR(2000),
      comment2 CHAR(2000)
END RECORD
```

If possible, replace such static arrays with dynamic arrays:

```
DEFINE my_array DYNAMIC ARRAY OF RECORD
      id CHAR(200),
      comment1 CHAR(2000),
      comment2 CHAR(2000)
END RECORD
```

However, be aware that dynamic arrays have a slightly different behavior than static arrays.

Logging options

Logging solutions allow to display exchanges between components when a program executes.

Genero provides several options to get debug information, as well as logging features, to ease regression test implementation:

- Logging the runtime errors in a file with STARTLOG().
- Getting the stack trace with `base.Application.getStackTrace()`
- Displaying the GUI protocol exchange in stderr with FGLGUIDEBUG.
- Log front-end protocol exchange with fglrun --start-guilog option.
- Displaying the SQL statements execution in stderr with FGLSQLDEBUG.

Extending the language

These topics cover extending Genero Business Development Language with other languages and external components.

- The Java interface on page 1564
- C-Extensions on page 1597
- User-defined front calls on page 1615
- Web Components on page 1636

The Java™ interface

The *Java™ interface* allows you to import Java classes and instantiate Java objects in your programs.

The Java interface gives access to the huge standard Java libraries, as well as commercial libraries for specific purposes.

The methods of Java objects can be called with other Java objects referenced in program, as well as with native language data types such as `INTEGER`, `DECIMAL`, `CHAR`.

The Java interface of Genero has the following limitations:

1. It is not possible to use Java generic types such as java.util.Vector<E>, with a type parameter (for ex: `Vector<MyClass> v = new Vector<MyClass>()`). However, it is possible to instanciate these classes without a type parameter (for ex: `Vector v = new Vector()`).
2. Database connections cannot be shared between Java and Genero programs.
3. Java graphical objects cannot be used in Genero forms.

   Note: On Android™ mobile devices, some system functions can only be accessed in the context of a JVM. Use the Java™ interface with the `com.fourjs.gma.vm.FglRun` class to access such system specifics.

- Prerequisites and installation on page 1565
- Getting started with the Java interface on page 1566
  - Import a Java class on page 1566
  - Define an object reference variable on page 1566
  - Instantiate a Java class on page 1567
  - Calling a method of a class on page 1567
  - Calling a method of an object on page 1567
- Advanced programming on page 1568
  - Using JVM options on page 1568
  - Case sensitivity with Java on page 1568
  - Method overloading in Java on page 1568
  - Passing Java objects to functions on page 1569
  - Using the method return as an object on page 1570
  - Ignorable return of Java methods on page 1570
  - Static fields of Java classes on page 1570
  - Mapping native and Java data types on page 1571
  - Using the DATE type on page 1572
  - Using the DATETIME type on page 1573
  - Using the INTERVAL type on page 1577
  - Using the DECIMAL type on page 1575

  - Using the BYTE type on page 1576
  - Using the TEXT type on page 1575
  - Identifying Genero data types in Java code on page 1579
  - Using Genero records on page 1580
  - Formatting data in Java code on page 1582
  - Character set mapping on page 1583
  - Using Java arrays on page 1583
  - Passing variable arguments (varargs) on page 1584
  - The CAST operator on page 1586
  - The INSTANCEOF operator on page 1586
  - Java exception handling on page 1587
  - Executing Java code with GMA on page 1587
- Examples on page 1592
  - Example 1: Using the regex package on page 1592
  - #unique_2611
  - Example 3: Using Java on Android on page 1593

Prerequisites and installation

Learn about Java™ and OOP

Before starting with the Java™ interface, if you are not familiar with Java™ and Object Oriented Programming, we strongly recommend that you learn more about this language from the different tutorials and courses you can find on the internet.

Java software requirements

In order to use the Java™ Interface in your application programs, you need the Java software installed and properly configured.

- Install a Java™ Development Kit on development sites (if you need to compile your own Java classes)
- Install a Java™ Runtime Environment on production sites (on the server where your programs are running)

The Java™ classes defined by Genero (`com.fourjs.fgl.lang.*`) are compiled with `javac -source 1.5 -target 1.5`, to be Java™ **1.5+** compatible. Therefore the minimum theoretical Java™ version is **1.5**. However, according to the platform, the minimum required version is **Java™ 1.6** or **1.7**.

The version of the installed Java software can be shown with the command:

```
java --version
```

In order to execute Java byte code, the Genero runtime system uses the JNI interface. The JVM is loaded as a shared library and its binary format must match the binary format of the Genero runtime system. For example, a 64-bit Genero package requires a 64-bit JVM.

When implementing Java classes for Genero Mobile for Android (GMA), check the JDK version required by the Android™ SDK. For more information, see the Android Studio web site.

How to set up Java™

This short procedure describes how to set up a Java™ environment to be used with Genero.

1. Download the latest JDK from your preferred Java™ provider. On production sites, you only need a Java™ Runtime Environment (JRE).
2. Install the package on your platform by following the vendor installation notes.

3. Set the PATH environment variable to the directory containing the Java™ compiler (javac), and to the Java™ Virtual Machine (java).
4. Configure your environment to let the dynamic linker find the `libjvm.so` shared library on UNIX™ or the `JVM.DLL` on Microsoft™ Windows™. For example, on a Linux/Intel you add `$JAVA_HOME/lib/i386/server` to LD_LIBRARY_PATH.

   Note: On Microsoft™ Windows™ platforms, make sure that the PATH envrionment variable does not contain double quotes arround the path to the `JVM.DLL` dynamic library, otherwise the DLL loader will fail to load the JVM. On Mac OS X, the JVM lib can be found from the JAVA_HOME directory, For more details, see Platform-specific notes for the JVM on page 1566
5. Set the CLASSPATH or pass the `--java-option=-Djava.class.path=<pathlist>` option to fglrun with the directories of the Java™ packages you want to use. You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with language specific classes such as `com.fourjs.fgl.lang.FglDecimal` or `com.fourjs.fgl.lang.FglRecord`.
6. Try your JDK by compiling a small java sample and executing it.

Platform-specific notes for the JVM

On some platforms like HP-UX® and AIX®, you must pay attention to additional configuration settings in order to use the Java™ Interface. For more details, see the OS specific notes in the installation guide.

On Microsoft™ Windows™ platforms, make sure that the PATH envrionment variable does not contain double quotes arround the path to the `JVM.DLL` dynamic library, otherwise the DLL loader will fail to load the JVM.

On Android™ devices, Java classes must be part of the `.apk` package and can be used without any further configuration.

On Mac OS X, the usage of DYLD_LIBRARY_PATH is strongly discouraged, especially since OS X 10.11 this environment variable is no longer exported in sub processes. In order to load the JVM, the runtime system will first try a regular dlopen("libjvm"). If this system call fails, the runtime system will lookup for the `libjvm.dylib` library under the typical `$JAVA_HOME/jre` directories (for example, `$JAVA_HOME/jre/bin/client`).

Note: In order to find JAVA_HOME on Mac OS X, use the `/usr/libexec/java_home` tool:

```
export JAVA_HOME=`/usr/libexec/java_home`
```

Getting started with the Java™ interface

Import a Java™ class

In order to use a Java™ class in your program code, you must first import the class with the `IMPORT JAVA` instruction:

```
IMPORT JAVA java.util.regex.Pattern
```

This will import the specified Java™ class into the current program module. Object references can now be defined for this class.

Define an object reference variable

Before creating a Java™ object in your program, you must declare a program variable to reference the object. The type of the variable must be the name of the Java™ class, and can be fully qualified if needed:

```
IMPORT JAVA java.util.regex.Pattern
MAIN
  DEFINE p1 Pattern
  DEFINE p2 java.util.regex.Pattern
```

```
END MAIN
```

The variables declared with a class are only the handles to reference an object (i.e. the object is not yet created).

Instantiate a Java™ class

To create a new Java™ object, use `ClassName.create()`, and assign the value returned by the `create()` method to a program variable declared with the Java™ class name:

```
IMPORT JAVA java.lang.StringBuffer
MAIN
    DEFINE sb StringBuffer
    LET sb = StringBuffer.create()
END MAIN
```

If the Java™ class constructor uses parameters, pass the parameters to the `create()` method:

```
IMPORT JAVA java.lang.StringBuffer
MAIN
    DEFINE sb1, sb2 StringBuffer
    -- Next code line uses StringBuffer(String str) constructor
    LET sb1 = StringBuffer.create("abcdef")
    -- Next code line uses StringBuffer(int capacity) constructor
    LET sb2 = StringBuffer.create(2048)
END MAIN
```

Calling a method of a class

Class methods (static method in Java™) can be called without instantiating an object of the class. Static method invocation must be prefixed with the class name. In the next example, the `compile()` class method of Pattern class returns a new instance of a Pattern object:

```
IMPORT JAVA java.util.regex.Pattern
MAIN
    DEFINE p Pattern
    LET p = Pattern.compile("[,\\s]+")
END MAIN
```

If you define a variable with the same name as a Java™ class, you must fully qualify the class when calling static methods, as shown in this example:

```
IMPORT JAVA java.util.regex.Pattern
IMPORT JAVA java.util.regex.Matcher
MAIN
    DEFINE Pattern Pattern
    DEFINE Matcher Matcher
    -- static method, needs full qualifier
    LET Pattern = java.util.regex.Pattern.compile("[a-z]+")
    -- regular instance method, Pattern resolves to variable
    LET Matcher = Pattern.matcher("abcdef")
END MAIN
```

Note that in Genero, program variables are case-insensitive (Pattern = pattern).

Calling a method of an object

Once the class has been instantiated as an object, and the object reference has been assigned to a variable, you can call a method of the Java™ object by using the variable as the prefix:

```
IMPORT JAVA java.util.regex.Pattern
```

```
IMPORT JAVA java.util.regex.Matcher
MAIN
    DEFINE p Pattern
    DEFINE m Matcher
    LET p = java.util.regex.Pattern.compile("[a-z]+")
    LET m = p.matcher("abcdef")
    DISPLAY m.matches()
END MAIN
```

In this example, the last line of the `MAIN` module calls an object method that returns a boolean value that is converted to an `INTEGER` and displayed.

Advanced programming

Using JVM options

When using the Java™ interface, you can instruct fglrun or fglcomp to pass Java™ VM specific options during JNI initialization, by using the `--java-option` command line argument.

In the next example, fglrun will pass `-verbose:gc` to the Java™ Virtual Machine:

```
$ fglrun --java-option=-verbose:gc myprog.42r
```

If you want to pass several options to the JVM, repeat the `--java-option` argument as in this example:

```
$ fglrun --java-option=-verbose:gc --java-option=-esa myprog.42r
```

You may want to pass the Java™ class path as command line option to fglrun with `-Djava.class.path` option as in the next example:

```
$ fglrun --java-option=-Djava.class.path=$FGLDIR/lib/fgl.jar:$MYCLASSPATH
 myprog.42r
```

Regarding class path specification, the `java` runtime or `javac` compiler provide the `-cp` or `-classpath` options but when loading the JVM library from fglrun or fglcomp, only `-Djava.class.path` option is supported by the JNI interface.

Case sensitivity with Java™

The Java™ language is case-sensitive. Therefore, when you write the name of a Java™ package, class or method in a .4gl source, it must match the exact name as if you were writing a Java™ program. The fglcomp compiler takes care of this, and writes case-sensitive class and method names in the .42m p-code modules.

```
IMPORT JAVA java.util.regex.Pattern
MAIN
  DEFINE p java.util.regex.PATTERN   -- Note the case error
END MAIN
```

With this code example, fglcomp will raise error -6622 at line 3, because the "java/util/PATTERN" name cannot be found.

Method overloading in Java™

The Java™ language allows method overloading; the parameter count and the parameter data types of a method are part of the method identification. Thus, the same method name can be used to implement different versions of the Java™ method, taking different parameters:

```
DEFINE int2 SMALLINT, int4 INTEGER, flt FLOAT
```

```
-- Next call invokes method display(short) of the Java class
CALL myobj.display(int2)

-- Next call invokes method display(int) of the Java class
CALL myobj.display(int4)

-- Next call invokes method display(double) of the Java class
CALL myobj.display(flt)

-- Next call invokes method display(short,int) of the Java class
CALL myobj.display(int2,int4)
```

Passing Java™ objects to functions

Java™ objects must be instantiated and referenced by a program variable. The object reference is stored in the variable and can be passed as a parameter or returned from a program function. The Java™ objects are passed by reference to functions. This means that the called function does not get a clone of the object, but rather a handle to the original object. The function can then manipulate and modify the original object provided by the caller:

```
IMPORT JAVA java.lang.StringBuffer

MAIN
  DEFINE x java.lang.StringBuffer
  LET x = StringBuffer.create()
  CALL change(x)
  DISPLAY x.toString()
END MAIN

FUNCTION change(sb)
  DEFINE sb java.lang.StringBuffer
  CALL sb.append("abc")
END FUNCTION
```

Similarly, Java™ object references can be returned from functions:

```
IMPORT JAVA java.lang.StringBuffer

MAIN
  DEFINE x java.lang.StringBuffer
  LET x = build()
  DISPLAY x.toString()
END MAIN

FUNCTION build()
  DEFINE sb java.lang.StringBuffer
  LET sb = StringBuffer.create()   -- Creates a new object.
  CALL sb.append("abc")
  RETURN sb  -- Returns the reference to the object, not a copy/clone.
END FUNCTION
```

Garbage collection of unused objects

Java™ objects do not need to be explicitly destroyed; as long as an object is referenced by a variable, on the stack or in an expression, it will remain. When the last reference to an object is removed, the object is destroyed automatically.

The next example shows how a unique object can be referenced twice, using two variables:

```
FUNCTION test()
  -- Declare 2 variables to reference a StringBuffer object
```

```
  DEFINE sb1, sb2 java.lang.StringBuffer
  -- Create object and assign reference to variable
  LET sb1 = StringBuffer.create()
  -- Same object is now referenced by 2 variables
  LET sb2 = sb1
  -- Object is modified through first variable
  CALL sb1.append("abc")
  -- Object is modified through second variable
  CALL sb2.append("def")
  -- Shows content of StringBuffer object
  DISPLAY sb1.toString()
  -- Same output as previous line
  DISPLAY sb2.toString()
  -- Object is only referenced by second variable
  LET sb1 = NULL
  -- sb2 removed from stack, object is no longer referenced and is
 destroyed.
END FUNCTION
```

Using the method return as an object

If a Java™ method returns an object, you can use the method call directly as an object reference to call another method:

```
IMPORT JAVA java.util.regex.Pattern
MAIN
  DEFINE p Pattern
  LET p = Pattern.compile("a*b")
  IF p.matcher("aaaab").matches() THEN
    DISPLAY "It matches..."
  END IF
END MAIN
```

In this code example, the `matcher()` method of object p is invoked and returns an object of type `java.util.regex.Matcher`. The object reference returned by the `matcher()` method can be directly used to invoke the `matches()` method of the `Matcher` class.

Ignorable return of Java™ methods

Java™ allows you to ignore the return value of a method (as in C/C++):

```
StringBuffer sb = new StringBuffer;
sb.append("abc");  -- returns a new StringBuffer object but is ignored
```

In programs, you can call a Java™ method and ignore the return value:

```
IMPORT JAVA java.util.lang.StringBuffer
MAIN
  DEFINE sb StringBuffer
  LET sb = StringBuffer.create()
  LET sb = sb.append("abc")
  CALL sb.append("def")  -- typical usage
END MAIN
```

Static fields of Java™ classes

Java™ classes can have object and class ("static") fields. Java™ static class fields can be declared as "final" (read-only). It is not possible to change the object or class fields in programs, even if the field is not declared as "static final"; you can however read from it:

```
IMPORT JAVA java.lang.Integer
```

```
MAIN
  DISPLAY Integer.MAX_VALUE
END MAIN
```

Mapping native and Java™ data types

Java™ and Genero have different built-in data types. Unlike Genero, Java™ is a strongly typed language: You cannot call a method with a `String` if it was defined to get an `int` parameter. To call a Java™ method, Genero native typed values need to be converted to/from Java™ types such as `byte`, `int`, `short`, `char` or data objects such as `java.lang.String`. If possible, the fglrun runtime system will do this conversion implicitly.

The fglcomp compiler will raise the error -6606, if the native data type does not match the Java™ (primitive) type, using Widening Primitive Conversions. For example, passing a Genero `DECIMAL` when a Java™ `double` is expected will fail, but passing a `SMALLFLOAT` (equivalent to Java™ `float`) when a Java™ `double` is expected will compile and run.

Genero has advanced native data types such as `DECIMAL`, which do not have an equivalent primitive type or class in Java™. For such Genero types, you need to use a specific Java™ class provided in the `FGLDIR/lib/fgl.jar` package, like `com.fourjs.fgl.lang.FglDecimal`. You can then manipulate the Genero specific value in the Java™ code.

Genero also implements structured types with `RECORD` definitions, converted to `com.fourjs.fgl.lang.FglRecord` objects for Java™.

The Genero arrays cannot be used to call Java™ methods. You must use a native Java™ arrays instead.

In some cases you need to explicitly cast with the new `CAST( )` operator. See the section about `CAST( )` operator for more details.

This table shows the implicit conversions done by the runtime system when a Java™ method is called, or when a Java™ method returns a value or object reference:

Table 328: Implicit conversions by Genero with Java™ method / Java™ method returns

Genero data type	Java™ equivalent
`CHAR`	`java.lang.String`
`VARCHAR`	`java.lang.String`
`STRING`	`java.lang.String`
`DATE`	`com.fourjs.fgl.lang.FglDate`
`DATETIME`	`com.fourjs.fgl.lang.FglDateTime`
`INTERVAL`	`com.fourjs.fgl.lang.FglInterval`
`BIGINT`	`long` (64-bit signed integer)
`INTEGER`	`int` (32-bit signed integer)
`SMALLINT`	`short` (16-bit signed integer)
`TINYINT`	`tinyint` (8-bit signed integer)
`FLOAT`	`double` (64-bit signed floating point number)
`SMALLFLOAT`	`float` (32-bit signed floating point number)
`DECIMAL`	`com.fourjs.fgl.lang.FglDecimal`
`MONEY`	`com.fourjs.fgl.lang.FglDecimal`

Genero data type	Java™ equivalent
BYTE	`com.fourjs.fgl.lang.FglByteBlob`
TEXT	`com.fourjs.fgl.lang.FglTextBlob`
RECORD structure	`com.fourjs.fgl.lang.FglRecord`
Java™ Array	This is a native Java™ Array

Table 329: Native language data types that cannot be converted to Java™ types

Genero data type
ARRAY structures
Built-in classes

Using the DATE type

When calling a Java™ method with an expression evaluating to a DATE, the runtime system converts the DATE value to an instance of the `com.fourjs.fgl.lang.FglDate` class implemented in `FGLDIR/lib/fgl.jar`. You can then manipulate the date from within the Java™ code.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglDate` class.

The `com.fourjs.fgl.lang.FglDate` class implements following:

Table 330: Methods of the com.fourjs.fgl.lang.FglDate class

Method	Description
`String toString()`	Converts the DATE value to a String object representing the date in format: YYYY-MM-DD
`static void valueOf( String val)`	Creates a new `FglDate` object from a String object representing a date in the format YYYY-MM-DD.

In the Java™ code, you can convert the `com.fourjs.fgl.lang.FglDate` to a `java.util.Calendar` object as in this example:

```
public static void useDate(FglDate d) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Calendar cal = Calendar.getInstance();
    cal.setTime( sdf.parse(d.toString()) );
    ...
}
```

If you need to create an `com.fourjs.fgl.lang.FglDate` object in your program, you can use the `valueOf()` class method as in this example:

```
IMPORT JAVA com.fourjs.fgl.lang.FglDate
MAIN
  DEFINE d com.fourjs.fgl.lang.FglDate
  LET d = FglDate.valueOf("2008-12-23")
  DISPLAY d.toString()
END MAIN
```

Using the DATETIME type

When calling a Java™ method with an expression evaluating to a DATETIME, the runtime system converts the DATETIME value to an instance of the com.fourjs.fgl.lang.FglDateTime class implemented in FGLDIR/lib/fgl.jar. You can then manipulate the DATETIME from within the Java™ code.

You must add FGLDIR/lib/fgl.jar to the class path in order to compile Java™ code with com.fourjs.fgl.lang.FglDateTime class.

The com.fourjs.fgl.lang.FglDateTime class implements following:

Table 331: Fields of the com.fourjs.fgl.lang.FglDateTime class

Field	Description
`final static int YEAR`	Time qualifier for year
`final static int MONTH`	Time qualifier for month
`final static int DAY`	Time qualifier for day
`final static int HOUR`	Time qualifier for hour
`final static int MINUTE`	Time qualifier for minute
`final static int SECOND`	Time qualifier for second
`final static int FRACTION`	Time qualifier for fraction (start qualifier)
`final static int FRACTION1`	Time qualifier for fraction(1) (end qualifier)
`final static int FRACTION2`	Time qualifier for fraction(2) (end qualifier)
`final static int FRACTION3`	Time qualifier for fraction(3) (end qualifier)
`final static int FRACTION4`	Time qualifier for fraction(4) (end qualifier)
`final static int FRACTION5`	Time qualifier for fraction(5) (end qualifier)

Table 332: Methods of the com.fourjs.fgl.lang.FglDateTime class

Method	Description
`String toString()`	Converts the `DATETIME` value to a String object representing a datetime in the format `YYYY-MM-DD hh:mm:ss.fff`.
`static void valueOf( String val)`	Creates a new `FglDateTime` object from a String object representing a datetime value in the format: `YYYY-MM-DD hh:mm:ss.fff`
`static void valueOf( String val, int startUnit, int endUnit)`	Creates a new `FglDateTime` object from a String object representing a datetime value in the format `YYYY-MM-DD hh:mm:ss.fff`, using the qualifiers passed as parameter.
`static int encodeTypeQualifier( int startUnit, int endUnit)`	Returns the encoded type qualifier for a datetime with to datetime qualifiers passed: encoded qualifier = (length * 256) + (startUnit * 16) + endUnit Where *length* defines the total number of significant digits in this time data. For example, with `DATETIME YEAR TO MINUTE`: startUnit = YEAR length = 12 (YYYYMMDDhhmm) endUnit = MINUTE

In the Java™ code, you can convert the `com.fourjs.fgl.lang.FglDateTime` to a `java.util.Calendar` object as in this example:

```
public static void useDatetime(FglDateTime dt) throws ParseException {
   SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
   Calendar cal = Calendar.getInstance();
   cal.setTime( sdf.parse(dt.toString()) );
   ...
}
```

If you need to create an `com.fourjs.fgl.lang.FglDateTime` object in your program, you can use the `valueOf()` class method as in this example:

```
IMPORT JAVA com.fourjs.fgl.lang.FglDateTime
MAIN
  DEFINE dt com.fourjs.fgl.lang.FglDateTime
  LET dt = FglDateTime.valueOf("2008-12-23 11:22:33.123")
  LET dt = FglDateTime.valueOf("11:22:33.123",
               FglDateTime.HOUR, FglDateTime.FRACTION3)
  DISPLAY dt.toString()
END MAIN
```

The `valueOf()` method expects a string representing a complete date-time specification, from year to milliseconds, equivalent to a `DATETIME YEAR TO FRACTION(3)` data type.

Using the DECIMAL type

When calling a Java™ method with an expression evaluating to a `DECIMAL` , the runtime system converts the `DECIMAL` value to an instance of the `com.fourjs.fgl.lang.FglDecimal` class implemented in `FGLDIR/lib/fgl.jar`. You can then manipulate the `DECIMAL` from within the Java™ code.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglDecimal` class.

The `com.fourjs.fgl.lang.FglDecimal` class implements following:

Table 333: Methods of the com.fourjs.fgl.lang.FglDecimal class

Method	Description
`String toString()`	Converts the `DECIMAL` value to a String object.
`static void valueOf(String val)`	Creates a new `FglDecimal` object from a `String` object representing a decimal value.
`static void valueOf(int val)`	Creates a new `FglDecimal` object from an `int` value.

`static int encodeTypeQualifier(`
`  int precision,`
`  int scale)`
	Returns the encoded type qualifier for this decimal according to precision and scale.
	encoded qualifier = (precision * 256) + scale
	Use 255 as scale for floating point decimal.

In the Java™ code, you can convert the `com.fourjs.fgl.lang.FglDecimal` to a `java.lang.BigDecimal` as in following example:

```
public static FglDecimal divide(FglDecimal d1, FglDecimal d2){
    BigDecimal bd1 = new BigDecimal(d1.toString());
    BigDecimal bd2 = new BigDecimal(d2.toString());
    BigDecimal res = bd1.divide(bd2, BigDecimal.ROUND_FLOOR);
    return FglDecimal.valueOf(res.toString());
}
```

If you need to create an `com.fourjs.fgl.lang.FglDecimal` object in your program, you can use the `valueOf()` class method as in this example:

```
IMPORT JAVA com.fourjs.fgl.lang.FglDecimal
MAIN
  DEFINE jdec com.fourjs.fgl.lang.FglDecimal
  LET jdec = FglDecimal.valueOf("123.45")
  DISPLAY jdec.toString()
END MAIN
```

Using the TEXT type

When calling a Java™ method with an expression evaluating to a `TEXT`, the runtime system converts the `TEXT` handle to an instance of the `com.fourjs.fgl.lang.FglTextBlob` class implemented in `FGLDIR/lib/fgl.jar`. You can then manipulate the LOB from within the Java™ code.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglTextBlob` class.

The `com.fourjs.fgl.lang.FglTextBlob` class implements following:

Table 334: Methods of the com.fourjs.fgl.lang.FglTextBlob class

Method	Description
`String toString()`	Converts the large text data to a simple `String`.
`static void valueOf( String val)`	Creates a new `FglTextBlob` object from a `String`.

In the Java™ code, you can pass a `com.fourjs.fgl.lang.FglTextBlob` object as in this example:

```
public static void useByte(FglTextBlob t) throws ParseException {
    String s = t.toString();
    ...
}
```

If you need to create an `com.fourjs.fgl.lang.FglTextBlob` object in your program, you can use the `valueOf()` class method as in this example:

```
IMPORT JAVA com.fourjs.fgl.lang.FglTextBlob
MAIN
  DEFINE jtext com.fourjs.fgl.lang.FglTextBlob
  LET jtext = FglTextBlob.valueOf("abcdef..........")
  DISPLAY jtext.toString()
END MAIN
```

Using the BYTE type

When calling a Java™ method with an expression evaluating to a BYTE, the runtime system converts the BYTE handle to an instance of the `com.fourjs.fgl.lang.FglByteBlob` class implemented in `FGLDIR/lib/fgl.jar`. You can then manipulate the LOB from within the Java™ code.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglByteBlob` class.

The `com.fourjs.fgl.lang.FglByteBlob` class implements following:

Table 335: Methods of the com.fourjs.fgl.lang.FglByteBlob class

Method	Description
`String toString()`	Returns the HEX string representing the binary data.
`static void valueOf( String val)`	Creates a new `FglByteBlob` object from a String object representing the binary data in HEX format.

In the Java™ code, you can pass a `com.fourjs.fgl.lang.FglByteBlob` object as in this example:

```
public static void useByte(FglByteBlob b) throws ParseException {
    String s = b.toString();
    ...
}
```

If you need to create an `com.fourjs.fgl.lang.FglByteBlob` object in your program, you can use the `valueOf()` class method as in this example:

```
IMPORT JAVA com.fourjs.fgl.lang.FglByteBlob
MAIN
  DEFINE jbyte com.fourjs.fgl.lang.FglByteBlob
  LET jbyte = FglByteBlob.valueOf("0FA5617BDE")
  DISPLAY jbyte.toString()
END MAIN
```

Using the INTERVAL type

When calling a Java™ method with an expression evaluating to an `INTERVAL`, the runtime system converts the `INTERVAL` value to an instance of the `com.fourjs.fgl.lang.FglInterval` class implemented in `FGLDIR/lib/fgl.jar`. You can then manipulate the `INTERVAL` from within the Java™ code.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglInterval` class.

The `com.fourjs.fgl.lang.FglInterval` class implements following:

Table 336: Fields of the com.fourjs.fgl.lang.FglInterval class

Field	Description
`final static int YEAR`	Time qualifier for year
`final static int MONTH`	Time qualifier for month
`final static int DAY`	Time qualifier for day
`final static int HOUR`	Time qualifier for hour
`final static int MINUTE`	Time qualifier for minute
`final static int SECOND`	Time qualifier for second
`final static int FRACTION`	Time qualifier for fraction (start qualifier)
`final static int FRACTION1`	Time qualifier for fraction(1) (end qualifier)
`final static int FRACTION2`	Time qualifier for fraction(2) (end qualifier)
`final static int FRACTION3`	Time qualifier for fraction(3) (end qualifier)
`final static int FRACTION4`	Time qualifier for fraction(4) (end qualifier)
`final static int FRACTION5`	Time qualifier for fraction(5) (end qualifier)

Table 337: Methods of the com.fourjs.fgl.lang.FglInterval class

Methods	Description
`String toString()`	Converts the `INTERVAL` value to a `String` object representing an interval in default format.
`static void valueOf(` `  String val)`	Creates a new `FglInterval` object from a `String` object representing an interval value in format: DD hh:mm:ss.fff

`static void valueOf(`
`  String val,`
`  int startUnit,`
	Creates a new `FglDateTime` object from a `String` object representing an interval value in standard format, using the qualifiers and precision passed as parameter.

Methods	Description
`int endUnit)`	

`static int encodeTypeQualifier(`
`  int startUnit,`
`  int length,`
`  int endUnit))`
	Returns the encoded type qualifier for an interval with to interval qualifiers and length passed:
	encoded qualifier = (*length* * 256) + (*startUnit* * 16) + *endUnit*
	Where *length* defines the total number of significant digits in this time data.
	For example, with INTERVAL DAY(5) TO FRACTION3:
	startUnit = DAY
	length = 13 (DDDDhhmmssfff)
	endUnit = FRACTION3

In the Java™ code, you can pass a `com.fourjs.fgl.lang.FglInterval` object as in this example:

```
public static void useInterval(FglInterval inv) throws ParseException {
    String s = inv.toString();
    ...
}
```

If you need to create an `com.fourjs.fgl.lang.FglInterval` object in your program, you can use the `valueOf()` class method as in this example:

```
IMPORT JAVA com.fourjs.fgl.lang.FglInterval
MAIN
  DEFINE inv com.fourjs.fgl.lang.FglInterval
  LET inv = FglInterval.valueOf("-510 12:33:45.123")
  DISPLAY inv.toString()
END MAIN
```

Identifying Genero data types in Java™ code

Java™ data types and Genero data types are different. To identify Genero types in Java™ code, you can use the `com.fourjs.fgl.lang.FglTypes` class implemented in `FGLDIR/lib/fgl.jar`.

You can for example identify the data type of a member of an FglRecord object.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglType` class.

The `com.fourjs.fgl.lang.FglTypes` class implements following:

Table 338: Fields of the com.fourjs.fgl.lang.FglTypes class

Field	Corresponding data type
`final static int BYTE`	BYTE
`final static int CHAR`	CHAR
`final static int DATE`	DATE
`final static int DATETIME`	DATETIME
`final static int DECIMAL`	DECIMAL
`final static int FLOAT`	FLOAT
`final static int INT`	INTEGER
`final static int SMALLFLOAT`	SMALLFLOAT
`final static int SMALLINT`	SMALLINT
`final static int VARCHAR`	VARCHAR
`final static int STRING`	STRING
`final static int RECORD`	RECORD structure
`final static int ARRAY`	ARRAY object

Using Genero records

When passing a `RECORD` to a Java™ method, the runtime system converts the `RECORD` to an instance of the `com.fourjs.fgl.lang.FglRecord` class implemented in `FGLDIR/lib/fgl.jar`.

The `FglRecord` object is a copy of the `RECORD` variable: Structure and members of the `FglRecord` object can be read within the Java™ code, but cannot be modified.

You must add `FGLDIR/lib/fgl.jar` to the class path in order to compile Java™ code with `com.fourjs.fgl.lang.FglRecord` class.

The `com.fourjs.fgl.lang.FglRecord` class implements the following methods:

Table 339: Methods of the com.fourjs.fgl.lang.FglRecord class

Method	Description
`int getFieldCount()`	Returns the number of record members.
`String getFieldName(int p)`	Returns the name of the record member at position *p*.
`FglTypes getType(int p)`	Returns the `FglTypes` constant of the record member at position *p*.
`String getTypeName(int p)`	Returns the string representation of the data type of the record member at position *p*.
`int getTypeQualifier(int p)`	Returns the encoded type qualifier of the record member at position *p*.
`int getInt(int p)`	Returns the int value of the record member at position *p*.
`int getFloat(int p)`	Returns the float value of the record member at position *p*.
`double getDouble(int p)`	Returns the double value of the record member at position *p*.
`String getString(int p)`	Returns the String representation of the value of the record member at position *p*.
`FglDecimal getDecimal(int p)`	Returns the `FglDecimal` value of the record member at position *p*.
`FglDate getDate(int p)`	Returns the `FglDate` value of the record member at position *p*.
`FglDateTime getDateTime(int p)`	Returns the `FglDateTime` value of the record member at position *p*.
`FglInterval getInterval(int p)`	Returns the `FglInterval` value of the record member at position *p*.
`FglByteBlob getByteBlob(int p)`	Returns the `FglByteBlob` value of the record member at position *p*.
`FglTextBlob getTextBlob(int p)`	Returns the `FglTextBlob` value of the record member at position *p*.

In the Java™ code, use the query methods of the `com.fourjs.fgl.lang.FglRecord` to identify the members of the `RECORD`:

```
public static void showMemberTypes(FglRecord rec){
    int i;
    int n = rec.getFieldCount();
    for (i = 1; i <= n; i++)
        System.out.println( String.valueOf(i) + ":" +
```

```
        rec.getFieldName(i) + " / " + rec.getTypeName(i) );
}
```

When assigning a RECORD to a com.fourjs.fgl.lang.FglRecord, *widening conversion* applies implicitly. But when assigning a com.fourjs.fgl.lang.FglRecord to a RECORD, *narrowing conversion* applies and you must explicitly CAST the original object reference to the type of the RECORD. The next example shows how to return an FglRecord object from a Java™ method:

```
-- PassRecord.4gl
IMPORT JAVA com.fourjs.fgl.lang.FglRecord
IMPORT JAVA UseRecord
MAIN
    TYPE type1 RECORD
               id INTEGER,
               name VARCHAR(50)
           END RECORD
    DEFINE rec1, rec2 type1
    LET rec1.id = 123
    LET rec1.name = "McFly"
    LET rec2 = CAST(UseRecord.getRecord(rec1) AS type1)
END MAIN

-- UseRecord.java
import com.fourjs.fgl.lang.FglRecord;
public class UseRecord{
    public static FglRecord getRecord(FglRecord rec){
        ...
        return rec;
    }
}
```

Formatting data in Java™ code

To format numeric and date-time data in Java™ code, use the com.fourjs.fgl.lang.FglFormat class implemented in FGLDIR/lib/fgl.jar.

You must add FGLDIR/lib/fgl.jar to the class path in order to compile Java™ code with com.fourjs.fgl.lang.FglFormat class.

The com.fourjs.fgl.lang.FglFormat class provides an interface to the data formatting functions of the runtime system. This class is actually an equivalent of the USING operator in the language.

The com.fourjs.fgl.lang.FglFormat class implements the following:

Table 340: Methods of the com.fourjs.fgl.lang.FglFormat class

Method	Description
`static String format( int v, String fmt)`	Formats the integer value provided as Java™ int, according to fmt. Here fmt must specify a numeric format with [$@*#&<()+-] characters, as with the USING operator.
`static String format( double v, String fmt)`	Formats the FLOAT value provided as Java™ double, according to fmt. Here fmt must specify a numeric format with [$ @*#&<()+-. ,] characters, as with the USING operator.
`static String format( FglDate v,`	Formats the DATE value provided as FglDate, according to fmt. Here fmt must specify a date

Method	Description
`String fmt)`	format with `[mdy]`characters., as with the `USING` operator.
`static String format( FglDecimal v, String fmt)`	Formats the `DECIMAL` value provided as `FglDecimal`, according to fmt. Here fmt must specify a numeric format with `[ $ @*# $<()+-. ,]` characters, as with the `USING` operator.

Example of Java™ code using the `com.fourjs.fgl.lang.FglFormat` class:

```
public static void formatDecimal(FglDecimal dec){
    System.out.println( FglFormat.format(dec,"$#####&.&&" );
}
```

Character set mapping

Application programs use a given locale and character set, while Java™ uses its own charset internally for the `char` Java™ type (16- bit UNICODE).

When passing character strings to/from Java™ methods or when assigning program strings to `java.lang.String`, the runtime system handles character set conversion.

Using Java™ arrays

Java™ arrays and Genero arrays are different. In order to interface with Java™ arrays, the Genero language has been extended with a new kind of arrays, called "Java™ arrays".

Java™ arrays have to be created with a given length. Like native Java™ arrays, the length cannot be changed after the array is created.

To create a Java™ array in Genero, you must define a `TYPE` in order to call the `create()` type method of Java™ arrays. The type of the elements in a Java™ array must be one of the language types that have a corresponding primitive type in Java™ (such as INTEGER (int), FLOAT (double)), or it must be a Java™ class such as java.lang.String.

The Java™ arrays are passed to Java™ methods by reference, so the elements of the array can be manipulated in Java™. Further, Java™ arrays can be created in Java™ code and returned to the Genero program.

This example shows how to create a Java™ array in Genero, to instantiate a Java™ Array of `INTEGER` elements:

```
MAIN
  TYPE int_array_type ARRAY[] OF INTEGER
  DEFINE ja int_array_type
  LET ja = int_array_type.create(100)
  LET ja[10] = 123
  DISPLAY ja[10], ja[20]
  DISPLAY ja.getLength()
END MAIN
```

The next example shows a program creating a Java™ array of Java™ strings:

```
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
```

```
    LET names[1] = "aaaaaaa"
    DISPLAY names[1]
END MAIN
```

To create a Java™ array of structured `RECORD` elements, use the `com.fourjs.fgl.lang.FglRecord` class:

```
IMPORT JAVA com.fourjs.fgl.lang.FglRecord
MAIN
  TYPE record_array_type ARRAY[]
       OF com.fourjs.fgl.lang.FglRecord
  DEFINE ra record_array_type
  TYPE r_t RECORD
       id INTEGER,
       name VARCHAR(100)
      END RECORD
  DEFINE r r_t
  LET ra = record_array_type.create(100)
  LET r.id = 123
  LET r.name = "McFly"
  LET ra[10] = r
  INITIALIZE r TO NULL
  LET r = CAST (ra[10] AS r_t)
  DISPLAY r.*
END MAIN
```

Java™ arrays of Java™ classes can be defined. The next example introspects the `java.lang.String` class by using Java™ array of `java.lang.reflect.Method` to query the list of methods from the `java.lang.String` class:

```
IMPORT JAVA java.lang.Class
IMPORT JAVA java.lang.reflect.Method
MAIN
  DEFINE c java.lang.Class
  DEFINE ma ARRAY[] OF java.lang.reflect.Method
  DEFINE i INTEGER
  LET c = Class.forName("java.lang.String")
  LET ma = c.getMethods()
  FOR i = 1 TO ma.getLength()
     DISPLAY ma[i].toString()
  END FOR
END MAIN
```

Java™ arrays can be created in the Java™ code, to be returned from a method and assigned to a program variable:

```
public static int [] createIntegerArray(int size) {
    return new int[size];
}
```

Passing variable arguments (varargs)

Java™ supports variable arguments in method definitions with the ellipsis notation, allowing callers to pass a different number of arguments according to the need. A typical example is a message print method:

```
import java.lang.String;

public class MyClass {
    public static void ShowStrings( String... sl ) {
        for ( String s : sl )
            System.out.println(s);
```

```
    }
}
```

In order to call such a method from the Genero program, create a Java™ array of the type of the variable argument, fill the array with objects and call the method with that array:

```
IMPORT JAVA java.lang.String
IMPORT JAVA MyClass

MAIN
    TYPE sl_t ARRAY[] OF java.lang.String
    DEFINE sl ARRAY[] OF java.lang.String
    LET sl = sl_t.create(2)
    LET sl[1] = "Value 1"
    LET sl[2] = "Value 2"
    CALL MyClass.ShowStrings(sl)
END MAIN
```

Since Java arrays have a static size, you must create the Java array with the exact number of variable arguments to be passed to the method.

If the Java class cannot be modified, consider implementing a function to wrap calls to the Java method, with a varying number of arguments. It can for example take a BDL dynamic array as parameter, to simplify the callers code:

```
IMPORT JAVA java.lang.String
IMPORT JAVA MyClass

MAIN
    DEFINE a DYNAMIC ARRAY OF STRING
    LET a[1] = "Value 1"
    LET a[2] = "Value 2"
    LET a[3] = "Value 3"
    CALL my_show_strings(a)
    LET a[4] = "Value 1"
    LET a[5] = "Value 2"
    CALL my_show_strings(a)
END MAIN

FUNCTION my_show_strings(sa)
    TYPE sl_t ARRAY[] OF java.lang.String
    DEFINE sa DYNAMIC ARRAY OF STRING
    DEFINE sl ARRAY[] OF java.lang.String
    DEFINE i INTEGER
    LET sl = sl_t.create(sa.getLength())
    FOR i=1 TO sa.getLength()
        LET sl[i] = sa[i]
    END FOR
    CALL MyClass.ShowStrings(sl)
END FUNCTION
```

If the Java class can be modified, a good practice is to write overloaded methods, using a static number of arguments:

```
public class MyClass {
    private static void _ShowStrings( String... sl ) {
        for ( String s : sl )
            System.out.println(s);
    }
    public static void ShowStrings(String s1) {
        _ShowStrings(s1);
    }
```

```
    public static void ShowStrings(String s1, String s2) {
        _ShowStrings(s1, s2);
    }
    public static void ShowStrings(String s1, String s2, String s3) {
        _ShowStrings(s1, s2, s3);
    }
}
```

The CAST operator

Important consideration has to be taken when assigning object references to different target types or classes. A *Widening Reference Conversion* occurs when an object reference is converted to a superclass that can accommodate any possible reference of the original type or class. A *Narrowing Reference Conversion* occurs when an object reference of a superclass is converted to a subtype or subclass of the original object reference. For example, in a vehicle class hierarchy with `Vehicle` and `Car` classes, `Car` is a subclass that inherits from the `Vehicle` superclass. When assigning a `Car` object reference to a `Vehicle` variable, Widening Reference Conversion takes place. When assigning a `Vehicle` object reference to a `Car` variable, Narrowing Reference Conversion occurs.

While widening conversion does not require casts and will not produce compilation or runtime errors, narrowing conversion needs the `CAST` operator to convert to the target type or class:

```
CAST( object_reference AS type_or_class )
```

The next example creates a `java.lang.StringBuffer` object, and assigns the reference to a `java.lang.Object` variable (implying Widening Reference Conversion); then the object reference is assigned back to the `java.lang.StringBuffer` variable (implying Narrowing Reference Conversion and `CAST` operator usage):

```
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE o java.lang.Object
  DEFINE sb java.lang.StringBuffer
  LET sb = StringBuffer.create()
  -- Widening Reference Conversion
  LET o = sb
  -- Narrowing Reference Conversion needs CAST()
  LET sb = CAST( o AS StringBuffer )
END MAIN
```

The INSTANCEOF operator

When manipulating an object reference with a variable defined with a superclass of the real class used to instantiate the object, you sometimes need to identify the real class of the object.

This is possible with the `INSTANCEOF` operator.

This operator checks whether the left operand is an instance of the type or class specified by the right operand:

```
object_reference INSTANCEOF type_or_class
```

This example creates a `java.lang.StringBuffer` object, assigns the reference to a `java.lang.Object` variable, and tests whether the class type of the object reference is a `java.lang.StringBuffer`:

```
IMPORT JAVA java.lang.Object
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE o java.lang.Object
```

```
  LET o = StringBuffer.create()
  DISPLAY o INSTANCEOF StringBuffer    -- Shows 1 (TRUE)
END MAIN
```

Java exception handling

In order to catch Java™ exceptions within programs, use a TRY/CATCH block.

When a Java exception occurs, the runtime system sets the STATUS variable to the error code -8306.

The Java exception details (i.e. the name of the exception) can be found with the ERR_GET(STATUS) built-in function.

> **Important:** To get the Java exception type with ERR_GET(), do not execute other instructions before querying for the error message, otherwise the STATUS variable might be reset to zero and the Java exception details would be lost.

To easily identify the type of the Java exceptions in your code, consider writing a library function based on ERR_GET(), that recognizes most common Java exceptions, and converts them to integer codes:

```
IMPORT JAVA java.lang.StringBuffer
MAIN
  DEFINE sb java.lang.StringBuffer
  LET sb = StringBuffer.create("abcdef")
  TRY
    CALL sb.deleteCharAt(50) -- out of bounds!
  CATCH
    DISPLAY err_get(STATUS)
    EXIT PROGRAM 1
  END TRY
END MAIN
```

> **Note:** As a general pattern, do not use TRY/CATCH or WHENEVER ERROR CONTINUE exception handlers if no exception is supposed to occur. By default the program will then stop and display the Java exception details.

Executing Java™ code with GMA

On Android™ devices running GMA apps, the Genero language can be extended with the Java interface.

The GMA executes a program in a JVM process and therefore does not require more resources to execute Java code.

We distinguish the following use cases where the Java interface of Genero can be used in GMA:

- Use classes from the standard Java or Android Java library.
- Implement and use user-defined Java classes, requiring GMA packaging.
- Implement and execute a user-defined Android activity, requiring GMA packaging.

Java may also be used to extend the GMA front-end with user-defined front calls. For details, see Implement front call modules for GMA on page 1620.

Standard Java™ and Android™ library usage

You can use Java classes that are part of the standard Java library and Android Java library.

Using standard Java within the GMA

Java classes provided in the standard Java library and in the Android Java library can be used directly by including the IMPORT JAVA *classname* keywords in the Genero code:

```
IMPORT JAVA java.lang.Runtime
IMPORT JAVA android.os.Build

MAIN
```

```
  DEFINE rtm Runtime, msg STRING

  LET rtm = java.lang.Runtime.getRuntime()

  LET msg = SFMT("Device:[%1] %2 - %3 (%4 procs)",
                 android.os.Build.MANUFACTURER,
                 android.os.Build.MODEL,
                 android.os.Build.SERIAL,
                 rtm.availableProcessors() )

  MENU "Test" ATTRIBUTES(STYLE="dialog", COMMENT=msg)
    ON ACTION ok
      EXIT MENU
  END MENU

END MAIN
```

The Android Java library does not include all the classes of a regular JRE. User interface classes are specific to the Android user interface framework. The list of standard Android Java packages can be found at http://developer.android.com/reference/packages.html.

Only non-interactive classes can be used in this context. To get a graphical user interface, you must implement an Android Activity, as described in Implement Android activities in GMA on page 1589.

Because Android apps are Java-based, the JVM and standard Java library is directly available. There is no need to bundle the Java library with your Genero program files when you deploy your app as .apk package.

When executing the Genero program on a computer in development mode, it is not possible to use classes that are specific to the Android Java library, because the Android Java library is not available in development mode at runtime.

You must compile your app code and deploy it on an Android device for execution. To compile your app code on the development platform, you need to setup the Java SDK environment and the CLASSPATH to the Android SDK library (android.jar).

> **Note:** For compilation, `JDK_HOME` can point to a 32-bit or 64-bit Java Development Kit installation, to match the Genero BDL architecture. However, the Android SDK is only available in 32-bit.

JVM context-dependent Android API calls

On an Android device, the GMA executes a Genero program in a JVM process. Some Android system APIs cannot be directly called from the Genero runtime system context; they must be called from the JVM context.

In order to call such APIs, you must import the `com.fourjs.gma.vm.FglRun` class and get the Android JVM thread context by calling the `getContext()` method of the `FglRun` class.

The `getContext()` method will return an instance of the `android.content.Context` class. For more details, see http://developer.android.com/reference/android/content/Context.html

> **Note:** To use this Android JVM interface, you must add the `android.jar` library (from the Android SDK) to the class path.

The `com.fourjs.gma.vm.FglRun` class implements the following methods:

Table 341: Methods of the com.fourjs.gma.vm.FglRun class

Method	Description
`Context getContext()`	Returns the Android JVM context object of the runtime system.

In the program code, use the `getContext()` method to get the JVM context and call specific Android APIs:

```
IMPORT JAVA android.app.Service
IMPORT JAVA android.content.Context
IMPORT JAVA android.util.DisplayMetrics
IMPORT JAVA android.view.WindowManager

IMPORT JAVA com.fourjs.gma.vm.FglRun

MAIN
    DEFINE w, h, d INT
    MENU "Java"
       ON ACTION test
          CALL android_screen_metrics() RETURNING w, h, d
          MESSAGE "Width: ", w, "\nHeight: ", h, "\nDensity: ", d
    END MENU
END MAIN

FUNCTION android_screen_metrics()
    DEFINE ctx android.content.Context,
           dm android.util.DisplayMetrics,
           wm android.view.WindowManager

    LET ctx = com.fourjs.gma.vm.FglRun.getContext()
    LET dm = android.util.DisplayMetrics.create()
    LET wm = CAST ( ctx.getSystemService("window")
                    AS android.view.WindowManager )
    CALL wm.getDefaultDisplay().getMetrics(dm)

    RETURN dm.widthPixels,
           dm.heightPixels,
           dm.densityDpi
END FUNCTION
```

Using front calls instead of pure Java

For maximum portability, consider implementing Android-specific extensions as custom front calls. When using the front call technology, apps can be executed in development (app running on the server) and in deployed mode (app running on the mobile device) with the same Genero code.

Implement Java user extensions in GMA

A GMA app can execute custom Java code.

In order to execute Java user code on the mobile device, the compiled Java classes need to be available to the Genero runtime system. They can then be imported with the `IMPORT JAVA` *`classname`* instruction.

When executing the Genero program on a computer in development mode, define the CLASSPATH to your .jar files. This allows the JVM loaded by the Genero runtime system find the appropriate Java classes.

When executing the Genero program on a mobile device, the compiled user Java classes must be included in the mobile app Android™ package (.apk), which is created in the Genero Studio deployment procedure.

Implement Android™ activities in GMA

Android activities can be bundled with your GMA app and called from the Genero code.

A Java-based extension that interacts with the end user must be implemented as an Android Activity, by using the `android.app.Android` class.

In order to use your Android Activity from the program, it must be integrated in the mobile app Android package (.apk), which is created in the Genero Studio deployment procedure.

This code example implements a simple Android Activity:

```
package com.myextension;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class MyActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Button button = new Button(this);
        button.setText("Quit");
        setContentView(button);
        button.setOnClickListener(
             new View.OnClickListener() {
             public void onClick(View v) {
                 int resultCode = 0;
                 Intent resultData = new Intent();
                 resultData.putExtra("MyKey", "MyValue");
                 setResult(resultCode, resultData);
                 finish();
             }
        });
    }
}
```

In order to execute this activity from a Genero app, use the `startActivity` front call:

```
MAIN
    DEFINE data, extras STRING
    MENU
        ON ACTION activity ATTRIBUTES(TEXT="Call bundled activity")
            CALL ui.Interface.frontCall("android", "startActivityForResult",
                 ["android.intent.action.VIEW", NULL, NULL, NULL,
                  "com.myextension.MyActivity"],
                 [ data, extras ])
                 MESSAGE "data=",data," / extras=",extras
        ON ACTION quit
            EXIT MENU
    END MENU
END MAIN
```

> **Note:** The component name (fifth parameter) of the startActivity front call does normally take the APK package name followed by the Java Activity class name (*`apk-package-name/java-class-name`*). The APK Android package name can be defined for the application project in the Genero Studio. When using an user-defined activity that is part of the GMA binary archive, do not specify the APK package in the component parameter, because the Java Activity class will be included in the current APK package. This is true when using the customized GMA front-end in development mode, and in the final application that is deployed on the device. For more details about the component parameter, see startActivity (Android) on page 1942.

Packaging custom Java™ extensions for GMA

Custom Java extension must be integrated in the GMA to run on Android™ devices.

Genero Mobile apps for Android are created from Genero Studio, or from the command-line with gmabuildtool on page 2580: You need to provide the custom GMA binary archive containing your Java extensions, to Genero Studio or to gmabuildtool.

- Genero Studio finds the GMA binary archive from the GMADIR variable defined in the configuration settings.

- The gmabuildtool requires the Android Studio project directory used to build the custom GMA, to be specified with the `--build-project-folder-path` option.

Along with the GMA binary archive, you must provide the `.jar` files of your Java extensions, that will be used to compile Genero application code on the development machine, as well as the `.apk` Android packages of GMA, to deploy the front-end part on the device for client/server development (typically with user-defined front calls).

The original GMA binary archive is a zip file containing several `.aar` Android libraries. A customized GMA binary archive contains the `.aar` files from the GMA core libraries, the Genero runtime system core libraries, and custom `.aar` files build from your own Java libraries. The custom `.aar` libs are created from Android Studio. The minimum Android Studio version is 0.8.9.

To create a new GMA binary archive, the `extension.jar` file, and the `.apk` packages, including your Java extensions, perform the steps described in Custom GMA binary archive build on page 1591.

After completing these steps:

- When compiling application code, Genero Studio can find your `.jar` libraries to resolve Java symbols.
- When deploying the front-end only for client/server development, Genero Studio will find the .apk packages to be installed on the device.
- When building an Android app in Genero Studio, it will be created from the custom GMA binary archive that includes your Java extensions.
- When building an Android app with gmabuildtool, it can be created by specifying the custom GMA Android project directory with the `--build-project-folder-path` option.

Custom GMA binary archive build
Building a GMA binary archive with custom Java classes using Android™ Studio.

Android Studio must be installed, and minimum Android development skills are required.

1. Locate the original GMA binary archive on your computer. When using Genero Studio, the GMA binary archive is defined by the GMADIR variable in configuration settings. When not using Genero Studio, the GMA binary archive is provided as a separate package.

   The GMA binary archive consist of a set of files:

   - `fjs-gma-*-android-scaffolding.zip` : This file contains the original GMA core binary without custom extensions.
   - `fjs-gma-*-android-extension-project.zip` : This file contains the Android Studio project template to build your custom GMA.
2. Unzip the `fjs-gma-*-android-extension-project.zip` archive into a directory of your choice (*my_project_dir*).
3. From Android Studio, open the project from *my_project_dir*.
4. In the project definition, under the "`extension`" library, find `AndroidManifest.xml` and modify the "`package`" attribute of the "`manifest`" node. This package name identifies your extension library, it will not be used to build a final app in GST.
5. Add your Java sources to this Android Studio project, under the "`extension`" library.
6. Locate the `testapp` app in the project.

   This is a sample activity that can be customized, to test your GMA extension directly within the Android Studio environment.
7. Modify `testapp`, to call your extension (see code for details), build and run for testing.
8. Build the project in release mode.

   This creates the `extension.jar` file and a `fjs-gma-.*-scaffolding.zip` archive, containing the `.aar` Android libraries. The files are created under the *my_project_dir*`/extension/build/outputs` directory.
9. Unzip the new GMA binary archive (`fjs-gma-.*-scaffolding.zip`) into a temporary directory (*my_tmp_dir*).

10. Locate the `gradlew` script from the unzipped GMA binary archive (*my_tmp_dir*).
11. Execute the command `gradlew build`.

    This will create the new `.apk` packages of GMA. The `.apk` files are created in the *`my_tmp_dir`*`/app/build/outputs` directory.
12. Create a new directory (*my_gma_dir*).
13. Copy the new `*-scaffolding.zip` file, the `extension.jar` and the `.apk` packages to *my_gma_dir*.
14. Modify the CLASSPATH configuration variable in Genero Studio, to find the `extension.jar` file.

    This is required to let the Genero compiler find your Java classes.
15. When using Genero Studio, modify the GMADIR configuration variable to point to the *my_gma_dir* directory.This is required to let Genero Studio use your customized GMA binary to build apps. When using gmabuildtool to build apps from the command line, provide the custom GMA Android project directory with the `--build-project-folder-path` option.
16. Deploy the new GMA apk on the device, for client/server development purpose.

    When using Genero Studio, the IDE will find the .apk packages to be installed on the device from GMADIR.

Examples

Example 1: Using the regex package

```
IMPORT JAVA java.util.regex.Pattern
IMPORT JAVA java.util.regex.Matcher
MAIN
  DEFINE p Pattern
  DEFINE m Matcher
  LET p = Pattern.compile("[a-z]+,[a-z]+")
  DISPLAY p.pattern()
  LET m = p.matcher("aaa,bbb")
  IF m.matches() THEN
    DISPLAY "The string matches the pattern..."
  ELSE
    DISPLAY "The string does not match the pattern..."
  END IF
END MAIN
```

Example 2: Using the Apache POI framework

This example shows how to create an XLS file, using the Apache POI framework. You must download and install the Apache POI JAR file and make the CLASSPATH environment variable point to the POI JAR in order to compile and run this example. After execution, you should find a file named "itemlist.xls" in the current directory, which can be loaded with Microsoft™ Excel or Open Office Calc:

```
IMPORT JAVA java.io.FileOutputStream
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFWorkbook
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFSheet
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFRow
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFCell
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFCellStyle
IMPORT JAVA org.apache.poi.hssf.usermodel.HSSFFont
IMPORT JAVA org.apache.poi.ss.usermodel.IndexedColors

MAIN
    DEFINE fo FileOutputStream
    DEFINE workbook HSSFWorkbook
    DEFINE sheet HSSFSheet
```

```
    DEFINE row HSSFRow
    DEFINE cell HSSFCell
    DEFINE style HSSFCellStyle
    DEFINE headerFont HSSFFont
    DEFINE i, id INTEGER, s STRING

    LET workbook = HSSFWorkbook.create()

    LET style = workbook.createCellStyle()
    CALL style.setAlignment(HSSFCellStyle.ALIGN_CENTER)
    CALL style.setFillForegroundColor(
                IndexedColors.LIGHT_CORNFLOWER_BLUE.getIndex());
    CALL style.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);
    LET headerFont = workbook.createFont()
    CALL headerFont.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD)
    CALL style.setFont(headerFont);

    LET sheet = workbook.createSheet()

    LET row = sheet.createRow(0)
    LET cell = row.createCell(0)
    CALL cell.setCellValue("Item Id")
    CALL cell.setCellStyle(style)
    LET cell = row.createCell(1)
    CALL cell.setCellValue("Name")
    CALL cell.setCellStyle(style)

    FOR i=1 TO 10
        LET row = sheet.createRow(i)
        LET cell = row.createCell(0)
        CALL cell.setCellType(HSSFCell.CELL_TYPE_NUMERIC)
        LET id = 100 + i
        CALL cell.setCellValue(id)
        LET cell = row.createCell(1)
        LET s = SFMT("Item #%1",i)
        CALL cell.setCellValue(s)
    END FOR

    LET fo = FileOutputStream.create("itemlist.xls");
    CALL workbook.write(fo);
    CALL fo.close();

END MAIN
```

Example 3: Using Java on Android™

This example shows how to access Android™ components throught Java, it includes:

- Access to the JDK API to get the number of cores on your device.
- Access to Android™ APIs to get the screen dimension, the device manufacturer and model (with no need for any additional authorization)
- Access to the Bluetooth stack to list the paired devices.

> **Note:** In your GM project, you need to ask for BLUETOOTH authorization.

Form file `formJavaStandard.per`:

```
LAYOUT (TEXT="Access to Android API")
GROUP group1(TEXT="Using standard JDK API...")
GRID grid1
{
[l1                       |f1                                    ]
```

```
}
END
END

ATTRIBUTES
LABEL l1 : label1, TEXT="Number of processors available";
LABEL f1 = FORMONLY.nb_proc;
END
```

Form file `formAndroidSimple.per`:

```
LAYOUT (TEXT="Access to Android API")
GROUP group1(TEXT="Using simple Android API...")
GRID grid1
{
[l1                        |f1                                    ]
[l2                        |f2                                    ]
[l3                        |f3                                    ]
[l4                        |f4                                    ]
}
END
END

ATTRIBUTES
LABEL l1 : label1, TEXT="Device manufacturer";
LABEL f1 = FORMONLY.manufacturer;
LABEL l2 : label2, TEXT="Device model";
LABEL f2 = FORMONLY.model;
LABEL l3 : label3, TEXT="Device serial number";
LABEL f3 = FORMONLY.serial;
LABEL l4 : label4, TEXT="Device screen dimension";
LABEL f4 = FORMONLY.diagonal;
END
```

Form file `formAndroidBluetooth.per`:

```
LAYOUT (TEXT="Access to Android API")
GROUP group1(TEXT="Using Bluetooth Android API...")
GRID grid1
{
[l1                           |f1                                  ]
<TABLE t                                                           >
[c1                           |c2                                  ]
[c1                           |c2                                  ]
[c1                           |c2                                  ]
<                                                                  >
}
END
END

ATTRIBUTES
LABEL l1 : label1, TEXT="Bluetooth adapter name";
LABEL f1 = FORMONLY.ba_name;
LABEL c1 = FORMONLY.name;
LABEL c2 = FORMONLY.comment;
END

INSTRUCTIONS
SCREEN RECORD list(FORMONLY.name, FORMONLY.comment);
END
```

Program file:

```
IMPORT util

IMPORT JAVA java.lang.Runtime
IMPORT JAVA java.util.Iterator
IMPORT JAVA java.lang.Class
IMPORT JAVA java.lang.Math

IMPORT JAVA android.bluetooth.BluetoothAdapter
IMPORT JAVA android.bluetooth.BluetoothDevice
IMPORT JAVA android.content.Context
IMPORT JAVA android.os.Build
IMPORT JAVA android.util.DisplayMetrics
IMPORT JAVA android.view.WindowManager

IMPORT JAVA com.fourjs.gma.vm.FglRun

MAIN
 MENU "Samples"
  COMMAND "Android API access"
   CALL androidApiAccess()
  COMMAND "Quit"
   EXIT MENU
  ON ACTION close
   EXIT MENU
 END MENU
END MAIN

FUNCTION androidApiAccess()

 MENU "Android API access"
  COMMAND "Accessing Java standard API"
   CALL androidApiAccess_java_standard()
  COMMAND "Accessing simple android information"
   CALL androidApiAccess_android_simple()
  COMMAND "Accessing sophisticated APIs : bluetooth"
   CALL androidApiAccess_bluetooth()
  ON ACTION CANCEL
   EXIT MENU
 END MENU
END FUNCTION

FUNCTION androidApiAccess_java_standard()
 DEFINE r Runtime

 OPEN WINDOW w WITH FORM "formJavaStandard"

 LET r = java.lang.Runtime.getRuntime()
 DISPLAY r.availableProcessors() TO nb_proc

 MENU
  ON ACTION QUIT
   EXIT MENU
  ON ACTION close
   EXIT MENU
 END MENU

 CLOSE WINDOW w
END FUNCTION

FUNCTION androidApiAccess_android_simple()
 DEFINE s STRING
```

```
 DEFINE dm DisplayMetrics
 DEFINE c Context
 DEFINE width, height, dens, wi, hi, x, y FLOAT
 DEFINE screenInches FLOAT
 DEFINE wm android.view.WindowManager

 OPEN WINDOW w WITH FORM "formAndroidSimple"

 LET s = android.os.Build.MANUFACTURER
 DISPLAY s TO manufacturer
 LET s = android.os.Build.MODEL
 DISPLAY s TO model
 LET s = android.os.Build.SERIAL
 DISPLAY s TO serial

 # Get the FglRun Context
 LET c = com.fourjs.gma.vm.FglRun.getContext()

 # Compute display dimension (diagonal)
 LET dm = android.util.DisplayMetrics.create()
 LET wm = CAST ( c.getSystemService("window") AS
 android.view.WindowManager )
 CALL wm.getDefaultDisplay().getMetrics(dm)
 LET width = dm.widthPixels
 LET height = dm.heightPixels
 LET dens = dm.densityDpi
 LET wi = width/dens
 LET hi = height/dens
 LET x = util.Math.pow(wi,2)
 LET y = util.Math.pow(hi,2);
 LET screenInches = util.Math.sqrt(x+y);

 DISPLAY screenInches TO diagonal
 MENU
  ON ACTION QUIT
   EXIT MENU
  ON ACTION close
   EXIT MENU
 END MENU

 CLOSE WINDOW w
END FUNCTION

FUNCTION androidApiAccess_bluetooth()
 DEFINE ba  BluetoothAdapter
 DEFINE sbd Iterator
 DEFINE bd  BluetoothDevice
 DEFINE bds DYNAMIC ARRAY OF RECORD
  name STRING,
  comment STRING
 END RECORD
 DEFINE i INTEGER
 DEFINE s STRING

 OPEN WINDOW w WITH FORM "formAndroidBluetooth"

 LET ba = android.bluetooth.BluetoothAdapter.getDefaultAdapter()
 LET s = ba.getName()
 DISPLAY s TO ba_name

 LET sbd = ba.getBondedDevices().iterator()
 LET i = 0
 WHILE sbd.hasNext()
  LET bd = CAST(sbd.next() AS BluetoothDevice)
```

```
  LET i = i + 1
  LET bds[i].name = bd.getName()
  LET bds[i].comment = bd.getBluetoothClass().toString()
 END WHILE

 DISPLAY ARRAY bds TO list.*
  ON ACTION QUIT
   EXIT DISPLAY
  ON ACTION close
   EXIT DISPLAY
 END DISPLAY

 CLOSE WINDOW w
END FUNCTION
```

C-Extensions

With *C-Extensions*, you can bind your own C libraries in the runtime system, to call C function from the application code.

- Understanding C-Extensions on page 1597
- Header files for ESQL/C typedefs on page 1598
- Creating C-Extensions on page 1598
- Creating Informix ESQL/C Extensions on page 1599
- The C interface file on page 1600
- Loading C-Extensions at runtime on page 1601
- Runtime stack functions on page 1602
- C-Extension data types and structures on page 1606
- Calling program functions from C on page 1610
- Sharing global variables on page 1611
- Simple C-Extension example on page 1612
- Implementing C-Extensions for GMI on page 1613

Understanding C-Extensions

With *C-Extensions*, you can bind your own C libraries in the runtime system, to call C function from the application code. This feature allows you to extend the language with custom libraries, or existing standard libraries, by writing some 'wrapper functions' to interface with the Genero language.

On regular platforms, C-Extensions are implemented with shared libraries, that will be loaded by the fglrun program on demand.

> **Note:** Platforms such as iOS mobile devices deny to load shared libraries. In this case, you must re-link the virtual machine. For more details, see Implementing C-Extensions for GMI on page 1613.

Function parameters and returned values are passed/returned on the runtime stack, using pop/push functions. Be sure to pop and push the exact number of parameters/returns expected by the caller; otherwise, a fatal stack error will be raised at runtime.

In order to use a C-Extension in your program, you typically specify the library name with the `IMPORT` instruction at the beginning of the module calling the C-Extension functions. The compiler can then check for function existence and the library will be automatically loaded at runtime.

> **Note:**
>
> - The C code written in C-Extensions is usually platform specific and does not ease the migration of your application to a different operating system, especially when doing a lot of system calls.

Additionally, C data types that are defined differently according to the processor architecture (32 / 64 bits issues) can also be an issue.

- Make sure that the functions defined in your C-Extensions do not conflict with program functions. In case of conflict, you will get a compiler or a runtime error, according to the loading technique used.

Header files for ESQL/C typedefs

To compile C-Extensions using complex data types such as `DECIMAL`, `DATETIME/INTERVAL` or `BYTE/TEXT`, you need IBM® Informix® ESQL/C data type structure definitions such as `dec_t`, `dtime_t`, `intrvl_t`, as well as macros like `DECLEN()` or `TU_ENCODE()`. These definitions are not required if you use standard C types such as short, `int` or `char[]`.

The definition of the ESQL/C structures like `dec_t` are property of IBM®. However, a copy of the ESQL/C header files used during the port of Genero are distributed in `FGLDIR/include/esql`, with agreement from IBM®.

Some ESQL/C type definitions are platform specific. For example, the `mlong` typedef is different on 32-bit and 64-bit machines.

Creating C-Extensions

Custom C-Extensions must be provided to the runtime system as Shared Objects (.so) on UNIX™, and as Dynamically Loadable Libraries (.DLL) on Windows™.

In order to create a C-Extension, you must:

1. Define the list of user functions in the C interface file, by including the `fglExt.h` header file.
2. Compile the C interface file with your C compiler.
3. Modify your C source modules by including the `fglExt.h` header file.
4. Compile the C interface file and the C modules with the position-independent code option.
5. Create the shared library with the compiled C interface file and C modules by linking with the `libfgl` runtime system library.

Include the `fglExt.h` header file in the following way:

```
#include "f2c/fglExt.h"
```

When migrating from IBM® Informix® 4GL, it is possible that existing C-Extension sources include Informix® specific headers like `sqlhdr.h` or `decimal.h`. You can either remove or keep the original includes, but if you want to keep them, the Informix® specific header files must be included before the `fglExt.h` header file, in order to let `fglExt.h` detect that typedefs such as `dec_t` or `dtime_t` are already defined by Informix® headers. If you include Informix® headers after `fglExt.h`, you will get a compilation error. As `fglExt.h` defines all Informix-like typedef structures, you can remove the inclusion of Informix® specific header files.

The C functions that are implemented in the C-Extension libraries must be known by the runtime system. To do so, each C-Extension library must publish its functions in a `UsrFunction` array, which is read by the runtime system when the module is loaded. The `UsrFunction` array describes the user functions by specifying the name of the function, the C function pointer, the number of parameters and the number of returned values. You typically define the `UsrFunction` array in the C interface file.

After compiling the C sources, you must link them together with the `libfgl` runtime system library.

Carefully read the man page of the ld dynamic loader, and any documentation of your operating system related to shared libraries. Some platforms require specific configuration and command line options when linking a shared library, or when linking a program using a shared library (+s option on HP for example).

Linux™ command-line example:

```
gcc -c -I $FGLDIR/include -fPIC myext.c
```

```
gcc -c -I $FGLDIR/include -fPIC cinterf.c
gcc -shared -o myext.so myext.o cinterf.o -L$FGLDIR/lib -lfgl
```

Windows™ command-line example using Visual C 8.0 and higher (with SxS manifest for the DLL!):

```
cl /DBUILDDLL /I%FGLDIR%/include /c myext.c
cl /DBUILDDLL /I%FGLDIR%/include /c cintref.c
link /dll /manifest /out:myext.dll myext.obj cinterf.obj %FGLDIR%\lib
\libfgl.lib
mt -manifest myext.dll.manifest -outputresource:myext.dll
```

If you build your DLL with a version of Microsoft™ Visual C++ that is different from the version used to build FGLRUN.EXE, the DLL must get private dependencies other than the process default. For example, when the C-Extension DLL needs the Visual C 9.0 runtime library MSVCR90.DLL, while the FGLRUN.EXE was build with VC 10 and needs MSVCR100.DLL. Private dependencies is specified with the resource id `ISOLATIONAWARE_MANIFEST_RESOURCE_ID`, by adding the **;2** modifier at the end of the `-outputresource` option, after the filename:

```
mt -manifest myext.dll.manifest -outputresource:myext.dll;2
```

To simplify compilation and linking of a C-Extension library, it is also possible to use the fglmkext command line tool:

```
fglmkext -o myext.so module_a.c module_b.c
```

> **Note:** The fglmkext command line tool contains platform-specific C compiler and linker options required to build a C Extension library.

Creating Informix® ESQL/C Extensions

C-Extension libraries can be created from ESQL/C sources, as long as you have an Informix® ESQL/C compiler which is compatible with your Genero runtime system.

In order to create a C-Extension from ESQL/C sources, you must:

1. Define the list of user functions in the C interface file, by including the `fglExt.h` header file.
2. Compile the C interface file with your C compiler.
3. Modify your ESQL/C source modules by including the `fglExt.h` header file.
4. Compile the ESQL/C modules with the `esql` compiler, with the position-independent code option.
5. Create the shared library with the compiled C interface file and ESQL/C modules by linking with the `libfgl` runtime system library, and with the ESQL/C libraries (`esql -libs`), to resolve the ESQL/C symbols.

Include the `fglExt.h` header file in the following way:

```
#include "f2c/fglExt.h"
```

You can compile .ec extensions with the native Informix® esql compiler. This section describes how to use the Informix® esql compiler.

The following example shows how to compile and link an extension library with Informix® esql compiler:

Linux™ command-line example:

```
esql -c -I$FGLDIR/include myext.ec
gcc -c -I$FGLDIR/include -fPIC cinterf.c
gcc -shared -o myext.so myext.o cinterf.o -L$FGLDIR/lib -lfgl \
    -L$INFORMIXDIR/lib -L$INFORMIXDIR/lib/esql `esql -libs`
```

Windows™ command-line example (using Microsoft™ Visual C++):

```
esql -c myext.ec -I%FGLDIR%/include
cl /DBUILDDLL /I%FGLDIR%/include /c cintref.c
esql -target:dll -o myext.dll myext.obj cinterf.obj %FGLDIR%\lib\libfgl.lib
```

When using Informix® esql, you link the extension library with Informix® client libraries. These libraries will be shared by the extension module and the Informix® database driver loaded by the Genero runtime system. Since both the extension functions and the runtime database driver use the same functions to execute SQL queries, you can share the current SQL connection opened in the Genero program to execute SQL queries in the extension functions. However, mixing connection management instructions (`DATABASE`, `CONNECT TO`) as well as database creation can produce unexpected results. For example you cannot do a `CREATE DATABASE` in your ESQL/C extension, and expect that the main program can use this database to execute SQL statements.

The C interface file

To make your C functions visible to the runtime system, you must define all the functions in the *C interface file*.

The *C interface file* is a C source file that defines the `usrFunctions` array. This array defines C functions that can be called from programs.

The last record of the `usrFunctions` array must be a line with all the elements set to NULL/0, to define the end of the list.

Each element of the `usrFunctions` array must be filled following members:

1. The first member is the name of the function, provided as a `(const char *)` character string.
2. The second member is the C function symbol, provided as an `(int (*function) (int))` C function pointer.
3. The third member is the number of parameters passed to the function through the runtime stack, provided as an `(int)`.
4. The fourth member is the number of values returned by the function, provided as an `(int)`; use -1 to specify a variable number of arguments.

You typically do a forward declaration of your C functions, before the `usrFunctions` array initializer:

```
#include "f2c/fglExt.h"

int c_init(int);
int c_set_trace(int);
int c_get_message(int);

UsrFunction usrFunctions[]={
  { "init",         c_init,         0, 0 },
  { "set_trace",    c_set_trace,    1, 0 },
  { "get_message",  c_get_message,  1, 1 },
  { NULL,           NULL,           0, 0 }
};
```

Note that the `UsrFunction` structure contains an additional member, dedicated for internal use. If you experience compiler warnings because of un-initialized structure members, simply complete the C function definitions with a fifth zero value:

```
/* Avoids C compiler warnings because of un-initialized structure members */

UsrFunction usrFunctions[]={
  { "init",         c_init,         0, 0, 0 },
            /* member for internal use ---^ */
  ...
```

Linking programs using C-Extensions

When creating a 42r program or 42x library, the linker needs to resolve all function names, including C-Extension functions.

If extension modules are not specified explicitly in the source files with the `IMPORT` directive, you must give the extension modules with the `-e` option in the command line:

```
fgllink -e myext1,myext2,myext3 -o myprog.42r moduleA.42m moduleB.42m ...
```

The `-e` option of fgllink does not write C-Extension references into the .42r file. If you use the `-e` argument with the fgllink command, you must also use the `-e` argument with the fglrun command, in order to load the libraries at runtime.

The `-e` option is not needed when using the default `userextension` module, or if C-Extensions are specified with the `IMPORT` directive.

Loading C-Extensions at runtime

The runtime system can load several C-Extensions libraries, allowing you to properly split your libraries by defining each group of functions in separate C interface files.

> **Note:** When running iOS platforms, the C-Extensions are linked statically to the GMI application.

Directories are searched for the C-Extensions libraries according to the FGLLDPATH environment variable rules. See the environment variable definition for more details.

If the C-Extension library depends on other shared libraries, make sure that the library loader of the operating system can find theses shared objects: You may need to set the LD_LIBRARY_PATH environment variable on UNIX™ or the PATH environment variable on Windows™ to point to the directory where these other libraries are located.

There are three ways to bind a C-Extension with the runtime system:

1. Using the `IMPORT` instruction in sources.
2. Using the default C-Extension name.
3. Using the `-e` option of fglrun.

Using the IMPORT instruction

The `IMPORT` instruction allows you to declare an external module in a .4gl source file. It must appear at the beginning of the source file.

The name of the module specified after the `IMPORT` keyword is converted to lowercase by the compiler. Therefore it is recommended to use lowercase file names only.

The compiler and the runtime system automatically know which C-Extensions must be loaded, based on the `IMPORT` instruction:

```
IMPORT mylib1
MAIN
  CALL myfunc1("Hello World")  -- C function defined in mylib1
END MAIN
```

When the `IMPORT` instruction is used, no other action has to be taken at runtime. The module name is stored in the 42m p-code and is automatically loaded when needed.

Using the default C-Extension name

All modules using a function from a C-Extension should now use the `IMPORT` instruction, however this could be a major change to existing sources.

To simplify migration of existing C-Extensions, the runtime system loads by default a module with the name `userextension`. Create this shared library with your existing C-Extensions, and the runtime system will load it automatically if it is in the directories specified by FGLLDPATH.

Using the -e fglrun option

In some cases you need several C-Extension libraries, which are used by different group of programs, so you can't use the default `userextension` solution. However, you don't want to review all your sources in order to use the `IMPORT` instruction.

You can specify the C-Extensions to be loaded by using the `-e` option of fglrun. The `-e` option takes a comma-separated list of module names, and can be specified multiple times in the command line. The next example loads five extension modules:

```
fglrun -e myext1,myext2,myext3 -e myext4,myext5 myprog.42r
```

By using the `-e` option, the runtime system loads the modules specified in the command line instead of loading the default `userextension` module.

Runtime stack functions

To pass values between a C function and a program, the C function and the runtime system use the runtime stack.

Stack function basics

The parameters passed to the C function must be popped from the stack at the beginning of the C function, and the return values expected by the Genero BDL call must be pushed on the stack before leaving the C function.

The `int` parameter of the C function defines the number of input parameters passed on the stack, and the function must return an `int` value defining the number of values returned on the stack.

> **Note:** If you don't pop / push the specified number of parameters / return values, you corrupt the stack and get a fatal error.

Pop parameters from the stack

The runtime system library includes a set of functions to retrieve the values passed as parameters on the stack. This table shows the library functions provided to pop values from the stack into C buffers:

Table 342: Library functions provided to pop values from the stack into C buffers

Function	Data type	Details
`void popdate(int4 *dst);`	`DATE`	4-byte integer value corresponding to days since12/31/1899.
`void popint(mint *dst);`	`INTEGER`	System dependent integer value (int)
`void popshort(int2 *dst);`	`SMALLINT`	2-byte integer value
`void poplong(int4 *dst);`	`INTEGER`	4-byte integer value
`void popflo(float *dst);`	`SMALLFLOAT`	4-byte floating point value
`void popdub(double *dst);`	`FLOAT`	8-byte floating point value

Function	Data type	Details
`void popdec(dec_t *dst);`	`DECIMAL`	See structure definition in `$FGLDIR/include/f2c headers`
`void popquote(char *dst, int size);`	`CHAR(n)`	The `size` parameter defines the size of the char buffer (with the '\0'). The trailing blanks are kept.
`void popvchar(char *dst, int size);`	`VARCHAR(n)`	The `size` parameter defines the size of the char buffer (with the '\0'). The trailing blanks are kept.
`void popstring(char *dst, int size);`	`VARCHAR(n)`	The `size` parameter defines the size of the char buffer (with the '\0'). This function trims all the trailing spaces, even the last one. There is no way to distinguish from NULL if the string has only spaces.
`void popdtime(dtime_t *dst, int size);`	`DATETIME`	See structure definition in `$FGLDIR/include/f2c headers` `size = TU_DTENCODE(start, end)`
`void popinv(intrvl_t *dst, int size);`	`INTERVAL`	See structure definition in `$FGLDIR/include/f2c headers` `size = TU_IENCODE(len, start, end)`
`void poplocator(loc_t **dst);`	`BYTE`, `TEXT`	See structure definition in `$FGLDIR/include/f2c headers` **Important:** this function pops the pointer of a `loc_t` object!

When using a pop function, the value is copied from the stack to the local C variable and the value is removed from the stack.

In a Genero program, strings (`CHAR`, `VARCHAR`) are not terminated by `'\0'`. Therefore, the C variable must have one additional character to store the `'\0'`. For example, the equivalent of a `VARCHAR(100)` in Genero BDL programs is a `char x[101]` in C.

Stack introspection

A set of C API functions are provided to query information on the parameters passed on the stack to a C function. Query for the parameter type and the actual size of a character string value, to adapt the buffer receiving the parameter.

Table 343: Library functions to introspect the runtime stack

Function	Description
`const char *fglcapi_peekStackType(void)`	Returns the type name of the topmost value on the stack as a string. For example, if the value on the stack is a CHAR(100), the function returns the string "CHAR(100)". **Note:** If the current value on the stack is a string literal ("foo") then the type name is "STRING" not "CHAR(3)".
`int fglcapi_peekStackBufferSize(void)`	Returns the proposed size of a C char buffer, when getting character strings from the stack with a pop* function. String pop functions such as popquote() and popvchar() require a C char buffer to be allocated. To allocate the buffer dynamically, use the fglcapi_peekStackBufferSize() function to get the actual size of the string parameter passed on the stack. Allocating char buffers with the proposed size avoids truncating values returned from the stack. **Important:** The size returned by this function depends on the encoding (LC_CTYPE) and the character length semantics. For example, assuming the value passed on the stack is a CHAR(100), the function returns: • 101 (100 + 1) when using byte semantics. • 301 (3 * 100 + 1) when using UTF-8 and character length semantics. See Length semantics settings on page 314 for more details about these concepts.

Stack introspection example:

```
int my_function(int n)
{
    int sz;
    char *buf;
    sz = fglcapi_peekStackBufferSize();
    buf = malloc(sz);
    popstring(buf, sz);
    // ...
```

```
    free(buf);
    return 0;
}
```

Push returns on the stack

To return a value from the C function, you must use one of the functions provided in the runtime system library.

Table 344: Functions provided in the runtime system library to return a value from a C function

Function	Data type	Details
`void pushdate(int4 val);`	`DATE`	4-byte integer value corresponding to days since12/31/1899.
`void pushdec(const dec_t *val, const unsigned decp);`	`DECIMAL`	See structure definition in `$FGLDIR/include/f2c headers`
`void pushint(mint val);`	`INTEGER`	System dependent integer value (int)
`void pushlong(int4 val);`	`INTEGER`	4-byte integer value
`void pushshort(int2 val);`	`SMALLINT`	2-byte integer value
`void pushflo(float *val);`	`SMALLFLOAT`	4-byte floating point value. **Important:** This function takes a pointer!
`void pushdub(double *val);`	`FLOAT`	8-byte floating point value. **Important:** This function takes a pointer!
`void pushquote(const char *val, int len);`	`CHAR(n)`	len = strlen(val) (without '\0')
`void pushvchar(const char *val, int len);`	`VARCHAR(n)`	len = strlen(val) (without '\0')
`void pushdtime(const dtime_t *val);`	`DATETIME`	See structure definition in `$FGLDIR/include/f2c headers`
`void pushinv(const intrvl_t *val);`	`INTERVAL`	See structure definition in `$FGLDIR/include/f2c headers`

When using a push function, the value of the C variable is copied at the top of the stack; therefore the scope and lifespan of the C variable does not matter.

To simplify migration of IBM I4GL legacy C extensions using `ret*()` style functions, Genero supports the following synonyms:

Table 345: Return value functions synonyms

Function	Equivalent
`void retdate(int4 val)`	`pushdate`
`void retdec(const dec_t *val)`	`pushdec`
`void retmoney(const dec_t *val)`	`pushdec`
`void retint(int val)`	`pushint`
`void retlong(int4 val)`	`pushlong`
`void retshort(int2 val)`	`pushshort`
`void retflo(float *val)`	`pushflo`
`void retdub(double *val)`	`pushdub`
`void retquote(const char *val)`	`pushquote`
`void retstring(const char *val)`	`pushquote`
`void retvchar(const char *val)`	`pushvchar`
`void retdtime(const dtime_t *val)`	`pushdtime`
`void retinv(const intrvl_t *val)`	`pushinv`

Note: Pay attention to the `retdec()`, `retmoney()`, `retquote()` and `retvchar()` functions. These do not have the same signature as the equivalent `push*()` functions.

C-Extension data types and structures

C types are used to write C-Extensions.

The following C types are used to write C-Extensions.

Table 346: C types used to write C-Extensions

Type name	Description
`int4`	signed integer with a size of 4 bytes
`uint4`	unsigned integer with a size of 4 bytes
`int2`	signed integer with a size of 2 bytes
`uint2`	unsigned integer with a size of 2 bytes
`int1`	signed integer with a size of 1 byte
`uint1`	unsigned integer with a size of 1 byte
`mint`	signed machine-dependent C int
`muint`	unsigned machine-dependent C int
`mlong`	signed machine-dependent C long
`mulong`	unsigned machine-dependent C long
`dec_t`	`DECIMAL` data type structure
`dtime_t`	`DATETIME` data type structure
`intrvl_t`	`INTERVAL` data type structure

Type name	Description
loc_t	TEXT / BYTE locator structure

Basic data types

Basic data types such as bigint, int4 and int2 are provided to define variables that must hold BIGINT (bigint), SMALLINT (int2), INTEGER (int4) and DATE (int4) values. Standard char array can be used to hold CHAR and VARCHAR data.

DATE

No specific typedef exists for the DATE type; you can use the int4 type to store a DATE value.

DECIMAL/MONEY

The dec_t structure is provided to hold DECIMAL and MONEY values.

The internals of dec_t structure can be ignored during C-Extension programming, because decimal API functions are provided to manipulate any aspects of a decimal.

DATETIME

The dtime_t structure holds a DATETIME value.

Before manipulating a dtime_t, you must initialize its qualifier qt_qual, by using the TU_DTENCODE macro:

```
dtime_t dt;
dt.dt_qual = TU_DTENCODE(TU_YEAR, TU_SECOND);
dtcvasc( "2004-02-12 12:34:56", &dt );
```

INTERVAL

The intrvl_t structure holds an INTERVAL value.

Before manipulating a intrvl_t, you must initialize its qualifier in_qual, by using the TU_IENCODE macro:

```
intrvl_t in;
in.in_qual = TU_IENCODE(5, TU_YEAR, TU_MONTH);
incvasc( "65234-02", &in );
```

TEXT/BYTE Locator

The loc_t structure is used to declare host variables for a TEXT/BYTE values (simple large objects). Because the potential size of the data can be quite large, this is a locator structure that contains information about the size and location of the TEXT/BYTE data, rather than containing the actual data.

Table 347: Fields of the loc_t structure

Field name	Data type	Description
loc_indicator	int4	Null indicator; a value of -1 indicates a null TEXT/BYTE value. Your program can set the field to indicate the insertion of a null value. Database client libraries

Field name	Data type	Description
		set the value for selects and fetches.
loc_type	int4	data type - SQLTEXT (for TEXT values) or SQLBYTES (for BYTE values).
loc_size	int4	Size of the TEXT/BYTE value in bytes; your program sets the size of the large object for insertions. Database client libraries set the size for selects and fetches.
loc_loctype	int2	Location - LOCMEMORY (in memory) or LOCFNAME (in a named file). Set loc_loctype after you declare the locator variable and before this declared variable receives the large object value.
loc_buffer	char *	If loc_loctype is LOCMEMORY, this is the location of the TEXT/BYTE value; your program must allocate space for the buffer and store its address here.
loc_bufsize	int4	If loc_loctype is LOCMEMORY, this is the size of the buffer loc_buffer; If you set loc_bufsize to -1, database client libraries will allocate the memory buffer for selects and fetches. Otherwise, it is assumed that your program will handle memory allocation and de-allocation.
loc_fname	char *	If loc_loc_type is LOCFNAME, this is the address of the pathname string that contains the file.

Example

```
loc_t *pb1
double ratio;
char *source = NULL, *psource = NULL;
int size;

if (pb1->loc_loctype == LOCMEMORY) {
    psource = pb1->loc_buffer;
    size = pb1->loc_size;
} else if (pb1->loc_loctype == LOCFNAME) {
    int fd;
    struct stat st;
    fd = open(pb1->loc_fname, O_RDONLY);
    fstat(fd, &st);
```

```
    size = st.st_size;
    psource = source = (char *) malloc(size);
    read(fd, source, size);
    close(fd);
}
```

Calling C functions from programs

C-Extensions functions can be called from the program in the same way that you call a BDL function.

The C functions that can be called from programs must use the following signature:

```
int function-name( int )
```

Here *function-name* must be written in lowercase letters. The fglcomp compiler converts all BDL functions names (following a CALL keyword) to lowercase.

The C function must be declared in the `usrFunctions` array in the C interface file.

> **Important:** Parameters and return values must be pushed/popped on the runtime stack, by using the stack functions. Parameters passed to the C function must be popped in the reverse order of the BDL call list: `CALL c_fct( A, B, C ) => pop C, B, A`. However, values returned from the C function must be pushed in the same order as in the BDL returning clause: `push A, B, C => CALL c_fct() RETURNING A, B, C`.

In the next code example, the C-Extension module `mycext.c` defines the `c_fct()` function:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "f2c/fglExt.h"

int c_fct( int n );

UsrFunction usrFunctions[]={
  {"c_fct",c_fct,2,2},
  {0,0,0,0}
};

int c_fct( int n )
{
   int rc;
   float price;
   char name[31];
   if (n != 2) exit(1);
   popflo(&price);
   popvchar(name, sizeof(name));
   printf(">> [%s] price:%f\n", name, price);
   pushint( strlen(name) );
   price = price * 2;
   pushflo( &price );
   return 0;
}
```

The C-Extension library is imported by the BDL module with `IMPORT`:

```
IMPORT mycext

MAIN
    DEFINE len INT, price2 FLOAT
    CALL c_fct("Hand gloves", 120.50)
```

```
          RETURNING len, price2
    DISPLAY "len = ", len
    DISPLAY "price2 = ", price2
END MAIN
```

Compilation and execution example on a Linux system:

```
$ gcc -I $FGLDIR/include -shared -fPIC -o mycext.so mycext.c

$ fglcomp myprog.4gl

$ fglrun myprog.42m
>> [Hand gloves] price:120.500000
len =          11
price2 =                   241.0
```

Calling program functions from C

It is possible to call an BDL function from a C-Extension function.

To call an BDL function from a C-Extension function, use the `fgl_call` macro:

```
fgl_call ( function-name, nb-params );
```

In this call, *function-name* is the name of the program function to call, and *nb-params* is the number of parameters pushed on the stack for the program function. The *function-name* must be written in lowercase letters; The fglcomp compiler converts all program functions names to lowercase.

The `fgl_call()` macro is converted to a function that returns the number of values returned on the stack.

> **Important:** Parameters and return values must be pushed/popped on the runtime stack, by using the stack functions. Parameters passed to the BDL function must be pushed in the same order as the BDL parameter list: `push A, B, C => FUNCTION fct( A, B, C )`. However, values returned from the BDL function must be popped in the reverse order of the BDL return clause: `RETURN A, B, C => pop C, B, A`.

The `myprog.4gl` BDL module defining the `MAIN` block and the `display_item()` function to be called from the C extension:

```
IMPORT mycext

MAIN
    CALL c_fct()
END MAIN

FUNCTION display_item(name, size)
   DEFINE name VARCHAR(30), size INTEGER
   DISPLAY name, size
   RETURN length(name), (size / 100)
END FUNCTION
```

The `mycext.c` C extension module calling the BDL function:

```
#include <stdlib.h>
#include <stdio.h>

#include "f2c/fglExt.h"

int c_fct( int n );

UsrFunction usrFunctions[]={
  {"c_fct",c_fct,0,0},
```

```
  {0,0,0,0}
};

int c_fct( int n )
{
   int rc, len;
   float size2;
   if (n != 0) exit(1);
   pushquote("Hand gloves", 11);
   pushint(54);
   rc = fgl_call( display_item, 2 );
   if (rc != 2) exit(1);
   popflo(&size2);
   popint(&len);
   printf(">> %d %f\n", len, size2);
   return 0;
}
```

Compilation and execution example on a Linux system:

```
$ gcc -I $FGLDIR/include -shared -fPIC -o mycext.so mycext.c

$ fglcomp myprog.4gl

$ fglrun myprog.42m
Hand gloves         54
>> 11 0.540000
```

Sharing global variables

While not recommended, you can share global variables declared in your program with a C module.

In order to share the global variables declared in your program, you must:

1. Generate the .c and .h interface files by using `fglcomp -G` with the module defining the global variables:

   ```
   GLOBALS
   DEFINE g_name CHAR(100)
   END GLOBALS
   ```

   ```
   fglcomp -G myglobals.4gl
   ```

   This will produce two files named `myglobals.h` and `myglobals.c`.
2. In the C module, include the generated header file and use the global variables directly:

   ```
   #include <string.h>
   #include "f2c/fglExt.h"
   #include "myglobals.h"

   int myfunc1(int c)
   {
      strcpy(g_name, "new name");
      return 0;
   }
   ```
3. When creating the C-Extension library, compile and link with the `myglobals.c` generated file.

   Tip: Using global variables is not recommended. It makes your code difficult to maintain. If you need persistent variables, use module variables and write set/get functions that you can interface with.

Simple C-Extension example

This example shows how to create a C-Extension library on Linux™ using gcc.

The command line options to compile and link shared libraries can change depending on the operating system and compiler/linker used.

The "split.c" file

```
#include <string.h>
#include "f2c/fglExt.h"

int fgl_split( int in_num );
int fgl_split( int in_num )
{
       char c1[101];
       char c2[101];
       char z[201];
       char *ptr_in;
       char *ptr_out;
       popvchar(z, 200); /* Getting input parameter */
       strcpy(c1, "");
       strcpy(c2, "");
       ptr_out = c1;
       ptr_in = z;
       while (*ptr_in != ' ' && *ptr_in != '\0')
       {
              *ptr_out = *ptr_in;
              ptr_out++;
              ptr_in++;
       }
       *ptr_out=0;
       ptr_in++;
       ptr_out = c2;
       while (*ptr_in != '\0')
       {
              *ptr_out = *ptr_in;
              ptr_out++;
              ptr_in++;
       }
       *ptr_out=0;
       pushvchar(c1, 100); /* Returning the first output parameter */
       pushvchar(c2, 100); /* Returning the second output parameter */
       return 2; /* Returning the number of output parameters (MANDATORY) */
}
```

The "splitext.c" C interface file

```
#include "f2c/fglExt.h"

int fgl_split(int);

UsrFunction usrFunctions[]={
  { "fgl_split", fgl_split, 1, 2 },
  { 0,0,0,0 }
};
```

Compile the C Module and the interface file

```
gcc -c -I $FGLDIR/include -fPIC split.c
```

```
gcc -c -I $FGLDIR/include -fPIC splitext.c
```

Create the shared library

```
gcc -shared -o libsplit.so split.o splitext.o -L$FGLDIR/lib -lfgl
```

The program "split.4gl"

```
IMPORT libsplit
MAIN
      DEFINE str1, str2 VARCHAR(100)
      CALL fgl_split("Hello World") RETURNING str1, str2
      DISPLAY "1: ", str1
      DISPLAY "2: ", str2
END MAIN
```

Compile the .4gl module

```
fglcomp split.4gl
```

Run the program without the -e option

```
fglrun split
```

Implementing C-Extensions for GMI

This section describes how to program C-Extensions for the GMI VM.

C-Extensions for GMI

With C-Extensions for GMI, you can address specific needs on iOS platforms, that are not available by default in the Genero language. For example, implement functions to interface with mobile specific hardware like sensors, card readers, scanners, bluetooth, etc.

The runtime system virtual machine build in the GMI for iOS platforms can be extended with the C-Extension technology. The basics to implement C-Extensions are the same for iOS as for Unix/Windows platforms, but there are some differences, explained in this section.

The main difference is that user libraries cannot be loaded dynamically on iOS and thus require a re-link of the GMI binary with the user-defined C-Extension library.

Writing C-Extension sources for GMI

C-Extension source files can be organized in several `.c` or `.m` files, but the final library name must be `userextension`.

For a first test, we recommend that you group all your C-Extension functions in a single sources file called `userextension.m`.

In the Objective C source file, you should add the following lines, to include typical iOS header files:

```
#include <Foundation/Foundation.h>
#include <UIKit/UIKit.h>
```

The Genero runtime system header file must be included as well:

```
#include "f2c/fglExt.h"
```

The C-Extension functions must be registered as usual, in a `UsrFunction` array, defining the number of input and output parameters:

```
UsrFunction usrFunctions[]={
  {"get_user_info",get_user_info,1,1},
  ...
  {NULL,NULL,0,0}
};
```

Using iOS C-Extensions in your program

The application code needs to be compiled on the development platform before it is deployed on the iOS device or simulator, by using the C-Extension library build for the development platform.

In your Genero program, import the C-Extension module with `IMPORT` userextension. You can also omit this `IMPORT` instruction, because the runtime system tries to find and load the `userextension` library by default. Note also that C-Extension function have a global scope, so you can omit to prefix the function name with the lib/module name:

```
IMPORT userextension
MAIN
   DEFINE info STRING
   LET info = get_user_info()
   ...
END MAIN
```

Compiler behavior regarding `IMPORT userextension` usage:

- With `IMPORT userextension`: The compiler can check references to functions defined in the extension. The programmer can qualify a function-name as `userextension.`*`function-name`*. But in this case, the `userextension.so` shared library must exist on the development platform.
- Without `IMPORT userextension`: The compiler can not check references to those functions. The compiler does not load the `userextension` module implicitly. C-Extension function names can not be qualified. In this case, the `userextension.so` library is not required for compilation, but it will be needed if the final program is linked, or if you want to execute/test the application in client/server development mode.

Compiling and linking with C-Extensions on the development platform

On the development machine, if you link 42r programs, or if you want the compiler to check for missing symbols (with the `-r` option), the `userextension` library must exist in the development environment.

> **Note:** At runtime, on the development machine, the extension library will be loaded at first extension function call. But when the application is deployed on the iOS device, the extension library will be part of the GMI/VM binary (because it is statically linked).

To create the `userextension` library for the development environment, you must build an Objective-C shared library.

If the C-Extension contains iOS API calls, it will not be possible to compile the extension library as is on the development machine: Write conditional pre-processor macros to hide the iOS specific code, and simulate the function behavior for the development platform:

```
#ifndef EMULATE_IOS
#include <Foundation/Foundation.h>
#include <UIKit/UIKit.h>
#endif
...
int get_user_info(int pc)
{
    char prop[101];
```

```
    char value[101];
    int z = (int) sizeof(prop);
    assert(pc==1);
    popvchar(prop, z);
#ifndef EMULATE_IOS
    ... here goes the iOS specific code ...
#else
    value[0] = '\0';
#endif
    pushvchar(value, (int) strlen(value));
    return 1;
}
```

Command line example to create a shared library with the XCode environment (note that we define the NOT_IOS_IMPL constant to compile the code without iOS specific API calls):

```
$ cc -shared -o userextension.dylib userextension.c \
      -D EMULATE_IOS -I $FGLDIR/include -L $FGLDIR/lib -lfgl
```

Building the iOS app with C extensions

Regular mobile iOS apps are created with the gmibuildtool on page 2592 command-line tool. However, if you want to build an iOS app using C extensions, you must setup a `Makefile` calling the `FGLDIR/lib/Makefile-gmi` generic makefile.

For more details, see Building iOS apps with Genero on page 2586

User-defined front calls

Front-ends can be extended with custom functions to access specific features.

It is possible to implement custom front-end functions to interface with platform-specific features, and use the feature from a Genero program through a front call. For example, you can implement a front-end function module interfacing with a bar code reader, to return bar codes to the Genero program.

This section describes how to implement your own front calls by front-end type. Because each front-end type uses different technologies, you must use native platform APIs to implement front calls.

Implement front call modules for GDC

Custom front call modules for the desktop front-end are implemented by using the API for GDC front calls in C language.

GDC custom front call basics

In order to extend the GDC with your own front calls, you must be familiar with C++ programming, and have a C++ compiler installed on your development platform.

GDC front call modules must be implemented as a Dynamic Linked Library (.DLL) on Windows™ platforms, as a shared library (.so) on Linux™, or as a Dynamic Library (.dyLib) under Mac Os X. This shared library must be deployed on each platform where the GDC front-end executes.

The GDC is able to automatically load the front call module and find the function, based on the module name and function name used in the Genero BDL front call (`ui.Interface.frontCall`).

The API for GDC front calls is based on the `frontEndInterface` front call interface structure, that is used to interface with the GDC core, in order to pass/return values to/from a front call.

Follow these steps to implement a custom front call module for the GDC:

1. Create a C source to implement your front call functions.

2. In the front call functions body:
   a. Check the number of parameters passed with the `getParamCount()` function.
   b. Pop parameter values with one of the `pop*()` functions.
   c. Perform the function task.
   d. Push the result values with one of the `push*()` functions.
   e. Return 0 on success, -1 otherwise.
3. Compile and link the shared library.
4. Deploy the shared library to the platform where GDC executes.

The front call interface structure

Information required to execute the front call is transmitted to the extension module through the front call interface structure. This structure contains a list of function pointers to:

- manage the stack (push or pop for each handled data type)
- get information about the function (number of in and out parameters)
- get information about the front-end (front call environment variables)

The following defines the front call interface structure:

```
struct frontEndInterface
  {
      short (* getParamCount) ();
      short (* getReturnCount) ();
      void (* popInteger) (long &, short &);
      void (* pushInteger) (const long, short);
      void (* popString) (char *, short &, short &);
      void (* pushString) (const char *, short, short);
      void (* getFrontEndEnv) (const char *, char *, short &);
      void (* popWString) (wchar_t *, short &, short &);
      void (* pushWString) (const wchar_t*, short, short);
  };
```

Important: The front call interface structure is defined for the C++ language.

Prototype of a front call function implementation

The prototype of each front call function must be:

```
int function_name ( const struct frontEndInterface &fci );
```

1. *function_name* is the name of your function.
2. `fci` is the front call interface structure.

The `fci` structure will be filled by the GDC and passed to the custom function. You can then use this structure to pop/push values from/to the stack, and get environment information from the core GDC.

The function must return 0 on success, -1 otherwise.

Front call environment variables

The front call function can query the GDC for front call environment variables, to get information about the context.

The following front call environment variables are supported:

Table 348: Supported front call environment variables for the GDC

Environment Variable	Description
`frontEndPath`	The path where the GDC front-end is installed.

Module initialization and finalization

The font-call module can define initialization and finalization functions. GDC will automatically call these functions as follows:

- `void initialize();`

  This function is called when the front call module library is loaded. If needed, perform variable initialization and resource allocation in this function.
- `void finalize();`

  This function is called when the GDC front-end stops. If needed, perform resource release in this function.

The API for custom front call implementation

Table 349: Front call interface functions

Function	Description
`short getParamCount();`	This function returns the number of parameters given to the function called.
`short getReturnCount();`	This function returns the number of returning values of the function called.
`void ( * getFrontEndEnv ) (const char * name, char * value, short & length );`	This function is used to get context information from the front-end. • `name` is the name of the front call environment variable. • `value` is the char buffer to hold the value of the variable. • `length` is the actual length of the value.
`void popInteger( long & value, short & isNull );`	This function is used to get an integer from the stack. • `value` is the reference to where the popped integer will be set. • `isNull` indicates whether the parameter is null.
`void pushInteger( const long value, short isNull );`	This function is used to push an integer on the stack. • `value` is the value of the integer. • `isNull` indicates whether the value is null.
`void popString( char * value, short & length, short & isNull );`	This function is used to get a string from the stack. • `value` is the pointer where the popped string will be set. • `length` is the length of the string.

Function	Description
	• `isNull` indicates whether the parameter is null.

`void pushString(`
`  const char * value,`
`  short length,`
`  short isNull );`
	This function is used to push a string on the stack.
	• `value` is the value of the string.
	• `length` the length of the string. A length of -1 indicates that the length is detected based on the content of the string.
	• `isNull` indicates whether the parameter is null.

`void ( * popWString )`
`(wchar_t *value,`
`short & length,`
`short & isNull);`
	This function is used to get a WideChar string from the stack.
	• `value` is the pointer where the popped string will be set.
	• `length` is the length of the string.
	• `isNull` indicates whether the parameter is null.

`void ( * pushWString )`
`(wchar_t *value,`
`short length,`
`short isNull);`
	This function is used to push a WideChar string on the stack.
	• `value` is the value of the string.
	• `length` the length of the string. A length of -1 indicates that the length is detected based on the content of the string.
	• `isNull` indicates whether the parameter is null.

Calling the custom front call from BDL

In the Genero program, use the `ui.Interface.frontCall()` API to call the front-end function. This method takes the front call module name as the first parameter and the front call function name as second parameter. The front call module name is defined by the name of the dynamic library (*module_name*.DLL, *module_name*.so or *module_name*.dylib).

For example, if you implement a front call module with the name "mymodule.so", the Genero program code must use the name "`mymodule`" as front call module name:

```
CALL ui.Interface.frontCall("mymodule", "myfunction", ["John DOE"], [msg])
```

Deploying the custom front call module

The shared library implementing the custom front call functions must be deployed on the platform where the GDC executes: Copy your custom front call modules in the bin directory of the GDC installation directory (i.e. `%GDCDIR%\bin`). This is also true when the GDC is deployed as ActiveX over the GAS.

Example

This example implements a simple front call function that computes the sum of two integer numbers. It takes two parameters and returns two values.

`mymodule.h`:

```
struct frontEndInterface
  {
      short (* getParamCount) ();
      short (* getReturnCount) ();
      void (* popInteger) (long &, short &);
```

```
      void (* pushInteger) (const long, short);
      void (* popString) (char *, short &, short &);
      void (* pushString) (const char *, short, short);
      void (* getFrontEndEnv) (const char *, char *, short &);
      void (* popWString) (wchar_t *, short &, short &);
      void (* pushWString) (const wchar_t*, short, short);
  };

#ifdef WIN32
#define EXPORT extern "C" __declspec(dllexport)
#else
#define EXPORT extern "C"
#endif

EXPORT void initialize();
EXPORT void finalize();
EXPORT int mysum(const frontEndInterface &fx);
```

`mymodule.cpp`:

```
#include "mymodule.h"
#include <stdio.h>
#include <string.h>

void initialize() {
}

void finalize() {
}

int mysum(const struct frontEndInterface &fci) {
  long param1, param2;
  short isNull1, isNull2;
  long sum;
  char msg[255];

  if (fci.getParamCount() != 2 || fci.getReturnCount() != 2) {
      return -1;
  }

  fci.popInteger(param2, isNull2);
  fci.popInteger(param1, isNull1);

  sum = param1 + param2;

  if (!isNull1 && !isNull2) {
    sum = param1 + param2;
    sprintf(msg, "%d + %d = %d", param1, param2, sum);
  } else {
    sum = 0;
    sprintf(msg, "Parameters are NULL");
  }

  fci.pushInteger(sum, 0);
  fci.pushString(msg, strlen(msg), 0);

  return 0;
}
```

To invoke the sum front-end function, use the `ui.Interface.frontCall()` method in your Genero program:

```
MAIN
```

```
  DEFINE res INT, msg STRING
  MENU
    ON ACTION frontcall ATTRIBUTES(TEXT="Call custom front call")
      CALL ui.Interface.frontCall("mymodule", "mysum",
                                  [100,250], [res,msg])
      DISPLAY "Result: ", res, "\n", msg
    ON ACTION quit
      EXIT MENU
  END MENU
END MAIN
```

Implement front call modules for GMA

Custom front call modules for the Android™ front-end are implemented by using the API for GMA front calls in Java™.

GMA custom front call basics

In order to extend the GMA with your own front calls, you must be familiar with Java programming concepts, and if you want to interface with Android apps, understand concepts such as Android Activity and Intent.

The API for GMA front calls is based on the following Java interfaces:

- `com.fourjs.gma.extension.v1.IFunctionCallController`
- `com.fourjs.gma.extension.v1.IFunctionCall`

The front call function controller (`IFunctionCallController`) is implemented by the GMA, it is used to notify function call results, raise runtime exceptions and invoke activities.

The front call function body (`IFunctionCall`) implements the actual custom front call code.

The steps to implement an `IFunctionCall` class are:

1. Create a Java source file with the name of the front call function, for example: "`getPhoneId.java`", that will implementing the `IFunctionCall` interface.
2. Define the Java package name identifying the front call module, for example: "`package com.mycompany.utilities;`".
3. Define a private `IFunctionCallController` object reference to handle the function controller.
4. Implement the `setFunctionCallController()` method for the function controller registration.
5. Implement the `invoke()` method to perform the actual front call task. In this method, use the controller's `returnValues()` method to return values from the front call. If needed, you can raise runtime errors with controller's `raiseError()` method. It is also possible to start an Android Activity with the `startActivity*` controller methods.
6. If an activity is started with controller's `startActivityForResult` method, implement the `onActivityResult()` method in the function body class, to handle the end of the activity, and call controller's `returnValues()` method to return values from the front call.
7. If needed, implement the `onSaveInstanceState()` and the `onRestoreInstanceState()` methods, to respectivly save and restore information when Android has to suspend the application.

> **Note:** In any case, the `IFunctionCall` class must either call the controller's `returnValues()` or `raiseError()` methods to give the control back to the Genero program.

The `com.fourjs.gma.extension.v1.IFunctionCall` interface

Table 350: Methods of the com.fourjs.gma.extension.v1.IFunctionCall interface

Method	Description
`void setFunctionCallController(IFunctionCallController` *`controller`*`)`	This method binds the front call function controller object to the function body object. The *controller* parameter is the `IFunctionCallController` object to bind with the front call function body object.
`abstract void invoke(Object[]` *`args`*`) throws IllegalArgumentException`	This method performs the front call. Is will be called when the front call is executed from the Genero program. The *args* parameter is a variable list of parameters passed to the front call. This corresponds to the third argument of ui.Interface.frontCall on page 395
`void onSaveInstanceState(Bundle` *`state`*`)`	Saves the state of an ongoing function call when Android needs to suspend the application. The *state* parameter is the bundle to save the state to.
`void onRestoreInstanceState(Bundle` *`state`*`)`	Restores the state of an ongoing function call, when Android needs to restore the application. The *state* parameter is the bundle to restore the state from.
`void onActivityResult(int` *`resultCode`*`, Intent` *`data`*`)`	Callback invoked when an activity started through `IFunctionCallController.startActivityForResult` finishes. The *resultCode* parameter is the integer result code returned by the child activity through its `setResult()` method. The *data* parameter is an Intent object, which can return result data to the caller (various data can be attached to Intent "extras").

The `com.fourjs.gma.extension.v1.IFunctionCallController` interface

Table 351: Methods of the com.fourjs.gma.extension.v1.IFunctionCallController interface

Method	Description
`void returnValues(IFunctionCall` *`functionCall`*`, Object...`*`values`*`)`	Notifies the controller that the front call function call has finished successfully. To be called typically at the end of the `IFunctionCall.invoke()` method. The *functionCall* parameter is the current `IFunctionCall` object invoked.

Method	Description
	The *values* parameter defines the variable list of front call function return values. This corresponds to the fourth parameter of ui.Interface.frontCall on page 395.
`void raiseError(IFunctionCall functionCall, String message)`	Notifies the controller of an error in the front call function call. This leads to a BDL runtime exception.To be called if needed within the `IFunctionCall.invoke()` method. The *functionCall* parameter is the current `IFunctionCall` object invoked. The *message* parameter holds the error message to be returned to the Genero program in the second part of the error -6333 message (see front call error handling in ui.Interface.frontCall on page 395).
`void startActivity(IFunctionCall functionCall, Intent intent)`	Starts a new activity. The function call won't be notified of the end of the activity. The Genero program will run in parallel of this activity. The behavior is similar to a `RUN WITHOUT WAITING`. The *functionCall* parameter is the current `IFunctionCall` object invoked. The *intent* parameter describes the activity to start.
`void startActivityForResult(IFunctionCall functionCall, Intent intent)`	Starts a new activity. The function call won't be notified of the end of the activity. The Genero program will remain blocked as long as the started activity isn't finished. The behavior is similar to a `RUN`. The method `IFunctionCall.onActivityResult` will be called once the activity finishes. The *functionCall* parameter is the current `IFunctionCall` object invoked. The *intent* parameter describes the activity to start.
`Activity getCurrentActivity()`	Returns the current `Activity` object. Provided in case if you need to pass the current activity to an Android API requiring this object. **Important:** Don't use the returned activity to start other activities (don't call `Activity.startActivity` or `Activity.startActivityForResult`), use the helpers of the current interface instead.

Calling the custom front call from BDL

In the Genero program, use the `ui.Interface.frontCall()` API to call the front-end function. This method takes the front call module name as first parameter and the front call function name as

second parameter. The front call module name is defined by the Java package name of the custom class implementing the `IFunctionCall` interface, and the front call function name is defined by the name of the class.

For example, if you implement the following front call function:

```
package com.mycompany.utilities;
...
public class GetPhoneId implements IFunctionCall {
...
```

The Genero program code must pass the Java package name "`com.mycompany.utilities`" as front call module name and the class name "`GetPhoneId`" as front call function name:

```
CALL ui.Interface.frontCall("com.mycompany.utilities", "GetPhoneId", ["John
 DOE"], [msg])
```

Deploying the custom front call

The compiled Java classes implementing the front calls must be included in the mobile application Android package (.apk), which is created in the Genero Studio deployment procedure. The same GMA package building rules apply for front calls and for simple Java extensions. See Packaging custom Java extensions for GMA on page 1590 for more details.

Example

The next example implements a HelloWorld call as a front call module.

`HelloWorld.java`:

```
package com.mycompany.testmodule;

import android.content.Intent;
import android.os.Bundle;

import com.fourjs.gma.extension.v1.IFunctionCall;
import com.fourjs.gma.extension.v1.IFunctionCallController;

public class HelloWorld implements IFunctionCall {

    private IFunctionCallController mController;

    @Override
    public void setFunctionCallController(IFunctionCallController
 controller) {
        mController = controller;
    }

    @Override
    public void invoke(Object[] args) throws IllegalArgumentException {
        if (args.length != 1) {
            throw new IllegalArgumentException("HelloWorld takes one
 argument");
        }

        mController.returnValues(this, "Hello " + args[0].toString());
    }

    @Override
    public void onSaveInstanceState(Bundle state) {
    }
```

```
    @Override
    public void onRestoreInstanceState(Bundle state) {
    }

    @Override
    public void onActivityResult(int returnCode, Intent data) {
    }
}
```

In order to invoke the HelloWorld front-end function, use the `ui.Interface.frontCall()` API in the Genero program:

```
MAIN
  DEFINE msg STRING
  MENU
    ON ACTION frontcall ATTRIBUTES(TEXT="Call custom front call")
      CALL ui.Interface.frontCall("com.mycompany.testmodule", "HelloWorld",
 ["John DOE"], [msg])
    ON ACTION quit
      EXIT MENU
  END MENU
END MAIN
```

Implement front call modules for GMI

Custom front call modules for the iOS front-end are implemented by using the API for GMI front-calls in Objective-C.

GMI custom front call basics

In order to extend the GMI with your own front calls, you must be familiar with Objective-C programming, and if you want to interface with iOS Apps, have a knowledge of the iOS API.

> **Important:** Before starting with GMI front call implementation, you need to get the GMI package and unzip the archive into the FGLDIR directory, as described in the prerequisites sections of Building iOS apps with Genero on page 2586.

The API for GMI front calls is based on the `FrontCall` class and the `FrontCallHelper` and `FunctionCall` protocols. You can find these in the file `frontcall.h` in the in the `FGLDIR/include/gmi` diretory.

To implement custom front calls, write a class which extends `FrontCall` and implement the "`moduleName`" and "`execute:retCount:params`" methods as well as the "`initWithFunctionModuleHelper:`" initializer.

To register your front calls with GMI, implement a function "`NSArray* frontCalls()`" in your extension project, which has to return an array of Objective-C strings with the names of the `FrontCall` classes you implemented.

Follow these steps to implement a custom front call module for the GMI:

1. Import the `frontcall.h` header file in your source.
2. Define an interface (`MyFrontCall`) which extends `FrontCall`.
3. Create the class (`MyFrontCall`) which implements this interface:
   a. Implement the `- (instancetype) initWithFunctionModuleHelper:(id)aHelper` initializer, calling `[superinitWithFunctionModuleHelper:aHelper]` to pass the `FrontCallHelper` to the base implementation.
   b. Implement the `- (NSString*) moduleName` method, returning the name of the front call module.

   c. Implement the `- (void)execute:(NSString)name retCount:(int)retCount params:(NSArray)params` method, defining the body of your front calls. See below for details about the `execute` method.
4. Implement the function `NSArray* frontCalls()` and return the class object of your class as the first element in the array (see below for code example).

API to implement custom front calls in GMI

To get parameters passed from the Genero program to the front call, and return values from the front call to the Genero program, use the following macros and methods of the `FrontCall` class:

Table 352: GMI custom front call API

Macro / Method	Description
`(void) FC_REQUIRED_PARAMS(`*`count`*`)`	Checks that the number of parameters passed by the Genero program equals *count*. This macro will raise an error in the Genero program if not enough parameters were passed.
`(NSString *) FC_PARAM(`*`index`*`)`	Get the string parameter passed to the front call, at the given position. If the parameters are of a different type, use the `doubleValue`, `floatValue` and `integerValue` methods on `NSString` or a `NSScanner`, to convert the parameter to the expected type.
`(int) FC_PARAM_INT(`*`index`*`)`	Get the int parameter passed to the front call, at the given position.
`(void) intResult:(int) intValue`	Ends the front call by returning one integer to Genero.
`(void) doubleResult:(double) doubleValue`	Ends the front call by returning one double to Genero.
`(void) stringResult:(NSString * ):stringValue*`	Ends the front call by returning one string to Genero.
`(void) startResult`	Initiate setting multiple result values. Must be followed by `add*` function calls and ended with `endResult`.
`(void) addIntResult:(int)intValue`	Add an integer to the list of results returned.

Macro / Method	Description
	To be used after a `startResult` call.
`(void) addDoubleResult:(double) doubleValue`	Add a double to the list of results returned. To be used after a `startResult` call.
`(void) addStringResult:(NSString * ) stringValue*`	Add a string to the list of results returned. To be used after a `startResult` call.
`(void) endResult`	Finalize the setting of multiple result values an return the results to the Genero program, with front call error code zero (i.e. success).
`(void) ok`	Ends the front call without returning any value to Genero, indicating that the front call execution was successful.
`(void) error(FCErrorCode):error`	Ends the front call with a specific front call error code defined in `FCErrorCode enum` in `frontcall.h`, to indicate that front call execution failed, typically because of invalid parameters or invalid function name.
`(void) errorWithMessage(NSString * )message*`	Ends the front call with front call return code -4 (maps to BDL error -6333), and a user-defined error message, that can be read with `ERR_GET()` in the Genero program.
`(void) willSetResultLater`	To be called at the end of the `execute` function, if result values are intended to be set after the `execute` function did return. If the `willSetResultLater` function is used, the current front call will not end until one of the result functions is called. For example, if your front call opens a message box, the `execute` function will return before one of the message box buttons are selected. Once a button is pressed, the front call result value is set.

Calling the custom front call from BDL

In the Genero program, use the `ui.Interface.frontCall()` API to call the front-end function. This method takes the front call module name as first parameter and the front call function name as second parameter.

The front call module name is defined by the string value returned from the `-(NSString * ) moduleName*` method of your front call implementation, and the front-call function name is passed to the `execute` method you implemented as first parameter (name).

For example, if you implement the following class:

```
#import <gmi/frontcall.h>
...
```

```
@interface MyFrontCall : FrontCall
   ...
@end

@class MyFrontCall

-(instancetype) initWithFunctionModuleHelper:(id)aHelper
{
   if (self = [superinitWithFunctionModuleHelper:aHelper]) {
      ...
   }
   return self;
}

-(NSString*) moduleName{
   return @"MyModule";
}

-(void)execute:(NSString)name
              retCount:(int)retCount
              params:(NSArray)params
{
   [super execute:name retCount:retCount params:params];
   if ([[name lowercaseString]isEqualToString:@"myfrontcall"]) {
      ...
```

The Genero program code must pass the module name "`MyModule`" as front call module name and the class name "`MyFrontCall`" as front call function name:

```
CALL ui.Interface.frontCall("MyModule", "MyFrontCall", ["John DOE"],[msg])
```

Custom front call implementation details (execute method)

First of all, call the execute method of the parent `FrontCall` class, right at the top of the `execute` method:

```
   [super execute:name retCount:retCount params:params];
```

The `execute` method must check the name of the front call function passed as parameter, to perform the expected code. This is the function name passed to the `ui.Interface.frontCall()` call in the Genero program:

```
   if([[name lowercaseString] isEqualToString:@"myfunction"]) {
```

Implement the body of the front call function in the `if()` block as follows:

Add an `assert()` line, to make sure that the number of return values match:

```
      assert(retCount == 2);
```

In order to get the parameters passed from the Genero program, use the FC_* macros in the body of your front call function.

First, check that the number of parameters passed is correct, with the `FC_REQUIRED_PARAMS(`*`count`*`)` macro:

```
      FC_REQUIRED_PARAMS(3);
      ...
```

Retrieve the parameters passed to the front call with the `FC_PARAM(index)` or `FC_PARAM_INT(index)` macros, which return a `NSString*` and an `int` respectively. If needed, use the `doubleValue`, `floatValue` and `integerValue` methods on `NSString` or a `NSScanner`, to convert the parameter to the expected type:

```
      NSString * info = FC_PARAM(0);
      int v1 = [FC_PARAM(1) integerValue];
      double v2 = [FC_PARAM(2) doubleValue];
```

Implement the actual code of the front call.

To return values to Genero, use one of the helper methods such as `intResult:value`, if a single value must be returned to the Genero program. If more than one value must be returned, build a return set with the `startResult`, `add*Result` and `endResult` methods:

```
    [self startResult];
    [self addIntResult:isIpad];
    [self addIntResult:canLocate];
    [self endResult];
```

If the front call displays a UI (e.g. an `UIAlertController` or displays a customer `UIViewController`), call the `willSetResultLater` method of the `FrontCall` class, to avoid that the control flow is returned to the Genero program upon exit of the `execute` method:

```
    [self willSetResultLater];
```

Additionally, if you call the `willSetResultLater` method, you need to call one of the result methods like `stringResult` at a later time.

Deploying the custom front call

The complied Objective-C classes and the `NSString frontCalls()` function must be included in the iOS app build process.

The same app building rules apply for custome front calls as for C extensions.

See Building iOS apps with Genero on page 2586 for more details.

Example

In this example, the `ExtensionFrontCall` class implements two front calls: "isipad" and "logindialog".

We start by defining the interface for the custom front call module:

```
@interface ExtensionFrontCall : FrontCall<UIAlertViewDelegate>
@end
```

The `ExtensionFrontCall` class extends `FrontCall`, and implements the `UIAlertViewDelegate` protocol which is used by the "logindialog" front call.

Next, we start the implementation of the interface:

```
@implementation ExtensionFrontCall

-(instancetype) initWithFunctionModuleHelper:(id)aHelper
{
   if (self = [superinitWithFunctionModuleHelper:aHelper]) {
   }
   return self;
}
```

```
-(NSString*) moduleName{
   return @"ExtensionFrontCall";
}

-(void)execute:(NSString)name retCount:(int)retCount params:(NSArray)params
 {
   [super execute:name retCount:retCount params:params];
   ...
}

...
@end
```

We use the standard initializer which will be called by GMI on startup and define "`ExtensionFrontCall`" as module name by returning it from the `moduleName` method.

We also start the implementation of the `execute` method by calling the super method.

Now we can implement the `frontCalls` function to notify GMI about the front call module we are adding:

```
NSArray* frontCalls()
{
   return @[ [ExtensionFrontCall class] ];
}
```

This function is added above the interface implementation in the example, but could be defined in any file, as long as it is included in the project. If more than one module has to be defined, add the class names of the other modules to the returned array (e.g. `return @[ [ExtensionFrontCall class] , [AnotherFrontCall class] ];`)

The `isipad` front call example

This front call simply returns the information on which device GMI is running. If it is an iPad the int 1 will be returned to the Genero program:

```
if ([[name lowercaseString] isEqualToString:@“isipad”]) {
    assert(retCount == 1);
    BOOL isIpad = UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad;
    [self startResult];
    [self addIntResult:isIpad];
    [self endResult];
}
```

After checking that only one return parameter was defined in Genero, the code identifies the platform with the `UI_USER_INTERFACE_IDIOM()` API and stores the result in the `isIpad` variable.

The next three lines return the result value to Genero, by starting a result block with `startResult`, adding an int to the return set with `addIntResult`, and finally calling `endResult` to send the result to the Genero progra,.

We could also have used the single line: `[self intResult:isIpad];` to achieve the same behavior, since we only return one result value.

The Genero program will call the `isIPad` front call as follows:

```
DEFINE res INTEGER
CALL ui.Interface.frontCall( "ExtensionFrontCall", "isipad", [ ], [res] )
```

The `logindialog` front call example

This front call will display a login dialog to the user. It expects two parameters (the title and the message for the login dialog), and will return the login name and the password entered by the end user:

```
if([[name lowercaseString] isEqualToString:@“logindialog”]) {
    assert(retCount == 2);
    FC_REQUIRED_PARAMS(2);
    NSString *title = FC_PARAM(0);
    NSString *message = FC_PARAM(1);
    UIAlertView *alert = [[UIAlertView alloc]
      initWithTitle:title
      message:message
      delegate:self
      cancelButtonTitle:NSLocalizedString(@"Cancel",@"Cancel")
      otherButtonTitles:NSLocalizedString(@"OK",@"OK"),nil];
    alert.alertViewStyle = UIAlertViewStyleLoginAndPasswordInput;
    [alert show];
    [self willSetResultLater];
}
```

We first check that two result values were set in Genero and that two parameters were supplied to the front call.

Then we use the `FC_PARAM` macro to fetch the parameters and assign them to `NSStrings`.

Then we allocate and initialize an `UIAlertView` with the given message and title and set the `alertViewStyle` to "`UIAlertViewStyleLoginAndPasswordInput`", so that one plain text field and one password field will be displayed on the alert.

In the `initWithTitle` call we also set "`self`" as the delegate of the alert so that we receive callbacks after user input (we had added the `UIAlertViewDelegate` protocol to our `ExtensionFrontCall` interface definition).

Finally, we call `willSetResultLater`, to keep the control flow in iOS. If we don’t call this function, GMI concludes the front call was not handled by the `execute` function (as none of the `xxxResult` functions was called inside), and the front call will fail with a “Frontcall not found” error message.

The `ExtensionFrontCall` class implements the `alertView:didDismissWithButtonIndex:` method from the `UIAlertViewDelegate` protocol:

```
pragma mark UIAlertViewDelegate(void)

alertView:(UIAlertView *)alertViewdidDismissWithButtonIndex:
(NSInteger)buttonIndex {
   [self startResult];
   if (buttonIndex != alertView.cancelButtonIndex) {
      [self addStringResult:[alertViewtextFieldAtIndex:0].text];
      [self addStringResult:[alertViewtextFieldAtIndex:1].text];
   } else {
      [self addStringResult:nil];
      [self addStringResult:nil];
   }
   [self endResult];
}
```

This method will be called after the user has tapped on one of the buttons and the view has been dismissed. Inside this method, we first call `startResult` to enable adding more than one return value.

If the tapped button was not the Cancel button, we add the values of the login and password fields as strings to the results and then call `endResult` to return the control flow to the Genero program.

The Genero program will call the login dialog front call as follows:

```
DEFINE ul, up STRING
CALL ui.Interface.frontCall( "ExtensionFrontCall", "logindialog",
                             ["MyApp","User login"],
                             [un, up] )
IF up IS NULL THEN
   ERROR "Login canceled"
   EXIT PROGRAM
END IF
```

> **Note:** The file `userextension.m` of the GMI Extension project contains a complete example on how to write custom front calls.

Implement front call modules for GWC - HTML5 theme

Custom front call modules for the GWC-HTML5 theme front-end are implemented by using JavaScript™.

GWC-HTML5 theme custom front call basics

When using the GWC-HTML5 theme, front-end calls are JavaScript™ functions executed locally on the workstation where the browser is running.

> **Note:** Executing front calls in the context of a web browser is limited to the OS functions a web browser can do. For example, it will not be possible to delete a file on the computer where the browser executes.

To implement custom front calls for GWC-HTML5 theme, you must edit the `csf.js` JavaScript file located in `$FGLASDIR/tpl/SetHtml5`. Genero built-in front calls and custom front calls are implemented in the `csf.js` file.

> **Important:** Custom front call module and function names must be registered in lowercase for the GWC-HTML5 theme front-end.

Follow these steps to implement a custom front call module for the GWC-HTML5 theme:

1. Edit the `$FGLASDIR/tpl/SetHtml5/csf.js` file.
2. Add a JavaScript object using the name of the front call module to the `gwc.frontCallModuleList` object:

   ```
   gwc.frontCallModuleList.mymodule = { ... }
   ```

3. Add your front call functions as JavaScript methods to the newly-created module object (with potential parameters):

   ```
   gwc.frontCallModuleList.mymodule = {
        myfunction : function ( param1, ... ) {
            ...
        }
   }
   ```

   The parameters of the JavaScript method must match the parameter list of the `ui.Interface.frontCall("mymodule", "myfunction", [`*`param-list`*`], [`*`return-list`*`])`.
4. If the front call must return values to the Genero program, add a `return` instruction in the JavaScript method:

   ```
       return ( [value1, ... ]);
   ```

The number of returned values must match the number of variables used in the return list of the Genero front call `ui.Interface.frontCall("mymodule", "myfunction", [`*`param-list`*`], [`*`return-list`*`])`.

If the front call does not return any value to the Genero program, the JavaScript method must return an empty list:

```
    return [];
```

Note:

- If the SetHtml5 directory contains compressed `.js` files, do not forget to compress the modified `csf.js` file, or remove the compressed version of the file (the GAS will use the non-compressed version).
- Keep in mind that the JavaScript modules can be cached in your browser. You may need to refresh the cache when doing modifications in the `csf.js` file.
- Make sure to save your custom front call definitions added to `csf.js` before installing a new version of the GAS; the existing `csf.js` will be overwritten by the new installation.
- Front call module and function names are case sensitive.

Example

Add the following lines in the `csf.js` file:

```
gwc.frontCallModuleList.mymodule = {
  myfunction1: function(param) {
    alert("param: " + param);
    return [];
  }
  myfunction2: function(param1,param2) {
    alert("param1: " + param1 + "\nparam2: " + param2);
    return [55,"aaa"];
  }
};
```

The above JavaScript code implements a front call module list with functions that can be called from the Genero programs as follows:

```
DEFINE r INTEGER, s STRING
CALL ui.Interface.frontCall("mymodule", "myfunction1", ["abc"], [] )
CALL ui.Interface.frontCall("mymodule", "myfunction2", [123,"abc"], [r,s] )
```

Implement front call modules for GWC - JavaScript

Custom front call modules for the GWC-JS front-end are implemented by using JavaScript™.

GWC-JS custom front call basics

In order to extend the GWC-JS with your own front calls, you must be familiar with JavaScript programming concepts.

> **Important:** Custom front call module and function names must be registered in lowercase for the GWC-JS front-end.

With GWC-JS, front-end calls are JavaScript functions executed locally on the workstation where the browser is running.

> **Note:** Executing front calls in the context of a web browser is limited to the OS functions a web browser can do. For example, it will not be possible to delete a file on the computer where the browser executes.

Customizing the GWC-JS front-end

In order to integrate your custom front calls in the GWC-JS front end, you need to setup the GWC-JS customization environment.

Basically, you will have to:

1. Setup GWC-JS customization (install Node.js).
2. Extract the GWC-JS front-end archive into a *project-dir* directory,
3. Copy your custom front calls JavaScript modules in the *`project-dir`*`/customization`,
4. Rebuild the GWC-JS front-end with the `grunt` utility.
5. Configure the GAS to use the customized GWC-JS front-end.

For more details, see GWS-JS customization chapter in the GAS documentation.

Structure of a custom front call JavaScript module

One JavaScript module will define a front call module implementing several front call fonctions.

The `.js` file must be copied into the *`project-dir`*`/customization` directory.

A custom front call JavaScript module must have the following structure:

```
"use strict";

modulum('FrontCallService.modules.module-name', ['FrontCallService'],
  /**
   * @param {gbc} context
   * @param {classes} cls
   */
  function(context, cls) {
    context.FrontCallService.modules.module-name = {

      function-name: function (param1, ...) {

           ... user code ...

      {
        return [ values ... ]
      |
          this.setReturnValues([ values ... ]);
      }

      },

      [...]   /* More functions can be defined for this module */

    };
  }
);
```

Where:

1. *module-name* is the name of the front call module, and corresponds to the first parameter of `ui.Interface.frontCall()`.
2. *function-name* is the name of the front call function, and corresponds to the second parameter of `ui.Interface.frontCall()`.
3. *param1*, *param2* ... are the input values provided as third parameter of `ui.Interface.frontCall()`.
4. *values* is a JavaScript array containaing the values to be returned in the last parameter of of `ui.Interface.frontCall()`.

GWC-JS custom front call API

The following JavaScript functions are provided to implement your custom front-calls:

Table 353: GWC-JS custom front call API

Method	Description
`this.parametersError( [ message ] )`	This function can be invoked when an invalid number of parameters is passed to the front call, in order to raise on exception in the BDL program. The *message* parameter holds the error message to be returned to the Genero program in the second part of the error -6333 message (see front call error handling in ui.Interface.frontCall on page 395).
`this.runtimeError( [ message ] )`	This function can be used to raise an exception in the BDL program, when the front call needs to warn the program that an error occured. The *message* parameter holds the error message to be returned to the Genero program in the second part of the error -6333 message (see front call error handling in ui.Interface.frontCall on page 395).
`this.setReturnValues( values )`	This function sets the values to be returned to the BDL program in the case of an asynchroneous front call. See Asynchroneous custom front calls on page 1634 for more details

Synchroneous custom front calls

Synchroneous front calls can directly return the front call values with a classic JavaScript `return` instruction, by specifying a JavaScript array.

The next code example returns a single value:

```
    return ["Hello " + name + " !"];
```

Following code example returns three values:

```
    return ["first", "second", "third"];
```

Asynchroneous custom front calls

JavaScript custom front calls sometimes require asynchroneous programming. In such case, the custom front call API provides the `setReturnValues()` function to register values that must be returned to the BDL program.

For example, to return value after a delay of 5 seconds:

```
        window.setTimeout(function () {
          this.setReturnValues(["After 5s, Hello " + name + " !"]);
        }.bind(this), 5000);
```

Example

The next JavaScript code example implements a synchroneous and an asynchroneous custom front call function:

```
"use strict";

modulum('FrontCallService.modules.mymodule',
 ['FrontCallService'],
  /**
   * @param {gbc} context
   * @param {classes} cls
   */
  function(context, cls) {
    context.FrontCallService.modules.mymodule = {

      add_hello_sync: function (name) {
        if (name === undefined) {
          this.parametersError();
          return;
        }
        if (name.length === 0) {
          this.runtimeError("name shouldn't be empty");
          return;
        }

        return ["Hello, " + name + " !"];
      },

      add_hello_async: function (name) {
        if (name === undefined) {
          this.parametersError();
          return;
        }
        if (name.length === 0) {
          this.runtimeError("name shouldn't be empty");
          return;
        }

        window.setTimeout(function () {
          this.setReturnValues(["After 5s, Hello, " + name +
 " !"]);
        }.bind(this), 5000);
      }
    };
  }
);
```

From the Genero BDL program:

```
DEFINE res INTEGER
CALL ui.Interface.frontcall("mymodule","add_hello_sync",
 ["world"] , [res])
CALL ui.Interface.frontcall("mymodule","add_hello_async",
 ["world"] , [res])
```

Web Components

Implement specialized form elements with Web Components.

For more details, see Web components on page 1416.

Library reference

Reference for classes and functions provided as built-in or extension packages.

- Built-in functions on page 1637
- Utility functions on page 1667
- Built-in packages on page 1687
- Extension packages on page 1947
- Built-in front calls on page 1881
- File extensions on page 2296
- Genero BDL errors on page 2297

Built-in functions

A *built-in function* is a predefined function that is part of the runtime system, or provided as a library function automatically loaded when a program starts. The built-in functions are part of the language.

Note that some operators such as `FIELD_TOUCHED(`*`field-spec`*`)` look like functions, but these are core language operators that are different in terms of semantics and order of precedence.

- Built-in functions on page 1637
- List of desupported built-in functions on page 1666
- The key code table on page 1666

Built-in functions

Table 354: Built-in functions

Function	Description

`arg_val(`
`  `*`position`* `INTEGER )`
`  RETURNING `*`result`* `STRING`
	Returns a command line argument by position.

`arr_count()` `  RETURNING `*`result`* `INTEGER`	Returns the number of rows entered during an `INPUT ARRAY` statement.
`arr_curr()` `  RETURNING `*`result`* `INTEGER`	Returns the current row in a `DISPLAY ARRAY` or `INPUT ARRAY`.

`downshift(`
`  `*`source`* `STRING )`
`  RETURNING `*`result`* `STRING`
	Converts a string to lowercase.

`err_get(` `  `*`errnum`* `INTEGER )`	Returns the text corresponding to an error number.

Function	Description
`  RETURNING result STRING`	
`err_print(` `  errnum INTEGER )`	Prints in the error line the text corresponding to an error number.
`err_quit(` `  errnum INTEGER )`	Prints in the error line the text corresponding to an error number and terminates the program.
`errorlog(` `  text STRING )`	Copies the string passed as parameter into the error log file.
`fgl_buffertouched()` `  RETURNING result INTEGER`	Returns TRUE if the input buffer was modified in the current field.
`fgl_db_driver_type()` `  RETURNING drvtype CHAR(3)`	Returns the 3-letter identifier/code of the current database driver.

`fgl_decimal_truncate(`
`  value DECIMAL,`
`  decimals INTEGER )`
`  RETURNING result DECIMAL`
	Returns a decimal truncated to the precision passed as parameter.

`fgl_decimal_sqrt(`
`  value DECIMAL )`
`  RETURNING result DECIMAL`
	Computes the square root of the decimal passed as parameter.

`fgl_decimal_exp(`
`  value DECIMAL )`
`  RETURNING result DECIMAL`
	Returns the value of Euler's constant (e) raised to the power of the decimal passed as parameter.

`fgl_decimal_logn(`
`  value DECIMAL )`
`  RETURNING result DECIMAL`
	Returns the natural logarithm of the decimal passed as parameter.

`fgl_decimal_power(`
`  base DECIMAL,`
`  exp DECIMAL )`
`  RETURNING result DECIMAL`
	Raises decimal to the power of the real exponent.

`fgl_dialog_getbuffer()` `  RETURNING result STRING`	Returns the text of the input buffer of the current field.
`fgl_dialog_getbufferlength()`	Returns the number of rows to feed a paged DISPLAY ARRAY.

Function	Description
RETURNING *result* INTEGER	
fgl_dialog_getbufferstart() RETURNING *result* INTEGER	Returns the row offset of the page to feed a paged display array.
fgl_dialog_getcursor() RETURNING *index* INTEGER	Returns the position of the edit cursor in the current field.
fgl_dialog_getfieldname() RETURNING *result* STRING	Returns the name of the current input field.
fgl_dialog_getkeylabel(*keyname* STRING) RETURNING *result* STRING	Returns the label associated to a key for the current interactive instruction.
fgl_dialog_getselectionend() RETURNING *position* INTEGER	Returns the position of the last selected character in the current field.
fgl_dialog_infield(*field-name* STRING) RETURNING *result* INTEGER	This function checks for the current input field.
fgl_dialog_setbuffer(*value* STRING)	Sets the input buffer of the current field.
fgl_dialog_setcurrline(*line* INTEGER, *row* INTEGER)	This function moves to a specific row in a record list.
fgl_dialog_setcursor(*position* INTEGER)	This function sets the position of the edit cursor in the current field.
fgl_dialog_setfieldorder(*active* INTEGER)	This function enables or disables field order constraint.
fgl_dialog_setkeylabel(*keyname* STRING, *label* STRING)	Sets the label associated to a key for the current interactive instruction.
fgl_dialog_setselection(*cursor* INTEGER, *end* INTEGER)	Selects the text in the current field.
fgl_drawbox(	Draws a rectangle in the current window.

Function	Description
height INTEGER, *width* INTEGER, *line* INTEGER, *column* INTEGER, *color* INTEGER)	
fgl_drawline(*column* INTEGER, *line* INTEGER, *width* INTEGER, *type* CHAR(1), *color* INTEGER)	Draws a line in the current window (TUI and traditional mode).
fgl_eventloop() RETURNING *status* BOOLEAN	Waits for a user interaction event.
fgl_dialog_getcursor() RETURNING *index* INTEGER	Returns the position of the edit cursor in the current field.
fgl_getenv(*variable* STRING) RETURNING *result* STRING	Returns the value of the environment variable.
fgl_getfile(*src* STRING, *dst* STRING)	Retrieves a file from the front-end context to the virtual machine context.
fgl_gethelp(*help-id* INTEGER) RETURNING *result* STRING	Returns the help text according to its identifier by reading the current help file.
fgl_getkey() RETURNING *keynum* INTEGER	Waits for a keystroke and returns the key number.
fgl_getkeylabel(*keyname* STRING) RETURNING *result* STRING	Returns the default label associated to a key.
fgl_getpid() RETURNING *result* INTEGER	Returns the system process identifier.
fgl_getresource(*name* STRING) RETURNING *result* STRING	Returns the value of an FGLPROFILE entry.
fgl_getversion()	Returns the product version number of Genero.

Function	Description
RETURNING *result* STRING	
fgl_getwin_height() RETURNING *result* INTEGER	Returns the number of rows of the current window.
fgl_getwin_width() RETURNING *result* INTEGER	Returns the width of the current window as a number of columns.
fgl_getwin_x() RETURNING *result* INTEGER	Returns the horizontal position of the current window.
fgl_getwin_y() RETURNING *result* INTEGER	Returns the vertical position of the current window.
fgl_keyval(*string* STRING) RETURNING *result* INTEGER	Returns the key code of a logical or physical key.
fgl_lastkey() RETURNING *result* INTEGER	Returns the key code corresponding to the logical key that the user most recently typed in the form.
fgl_putfile(*src* STRING, *dst* STRING)	Transfers a file from the virtual machine context to the front end context.
fgl_report_print_binary_file(*filename* STRING)	Prints a file containing binary data during a report.
fgl_report_set_document_handler(*handler* om.SaxDocumentHandler)	Redirects the next report to an XML document handler.
fgl_scr_size(*screen-array* STRING) RETURNING *result* INTEGER	Returns the size of the specified screen array in the current form.
fgl_set_arr_curr(*row* INTEGER)	Moves to a specific row in a record list.
fgl_setenv(*variable* STRING, *value* STRING)	Sets the value of an environment variable.
fgl_setkeylabel(*keyname* STRING,	Sets the default label associated to a key.

Function	Description
label STRING)	
fgl_setsize(*height* INTEGER, *width* INTEGER)	Sets the size of the main application window.
fgl_settitle(*label* STRING)	Sets the title of the current application window.
fgl_system(*command* STRING)	Runs a command on the application server.
fgl_width(*expression* STRING) RETURNING *result* INTEGER	Returns the number of columns needed to represent the printed version of the expression.
fgl_window_getoption(*attribute* STRING) RETURNING *result* STRING	Returns attributes of the current window.
length(*expression* STRING) RETURNING *result* INTEGER	Returns the number of the character string passed as parameter.
num_args() RETURNING *result* INTEGER	Returns the number of program arguments.
scr_line() RETURNING *result* INTEGER	Returns the index of the current row in the screen array.
set_count(*nbrows* INTEGER)	Defines the number of rows containing explicit data in a static array used by the next dialog.
showhelp(*helpnum* INTEGER)	Displays a runtime help text.
startlog(*filename* STRING)	Initializes error logging and opens the error log file passed as the parameter.
upshift(*source* STRING) RETURNING *result* STRING	Converts a string to uppercase.

arg_val()
Returns a command line argument by position.

Syntax

```
arg_val(
   position INTEGER )
  RETURNING result STRING
```

1. *position* is an integer defining the argument position.
2. *result* is a string containing the program argument.

Usage

This function provides a mechanism for passing values to the program through the command line that invokes the program. You can design a program to expect or allow arguments after the name of the program in the command line.

The *position* parameter defines the argument to be returned. 0 returns the name of the program, 1 returns the first argument.

Like all built-in functions, `arg_val()` can be invoked from any program block. You can use it to pass values to `MAIN`, which cannot have formal arguments, but you are not restricted to calling `arg_val()` from the `MAIN` statement.

Use the `arg_val()` function to retrieve individual arguments during program execution. Use the `num_args()` function to determine how many arguments follow the program name on the command line.

If *position* is greater than 0, `arg_val(`*`position`*`)` returns the command-line argument used at a given position. The value of *position* must be between 0 and the value returned by `num_args()`, the number of command-line arguments. The expression `arg_val(0)` returns the name of the application program.

If the argument position is negative or greater than `num_args()`, the method returns `NULL`.

arr_count()
Returns the number of rows entered during an `INPUT ARRAY` statement.

Syntax

```
arr_count()
  RETURNING result INTEGER
```

1. *result* is the current number of records that exist in the array.

Usage

Use `arr_count()` to determine the number of program records that are currently stored in a static program array used by the `INPUT ARRAY` instruction.

This function is typically called inside or after `INPUT ARRAY` or `DISPLAY ARRAY` statement.

`arr_count()` returns a positive integer, corresponding to the index of the furthest record within the static program array that the user accessed. Not all the rows counted by `arr_count()` necessarily contain data (for example, if the user presses the Down key more times than there are rows of data.

This function is not required when using dynamic arrays. In such case, the total number of rows in defined by the `array.getLength()` method after the dialog, or by the `ui.Dialog.getArrayLength()` method during the dialog execution.

arr_curr()
Returns the current row in a `DISPLAY ARRAY` or `INPUT ARRAY`.

Syntax

```
arr_curr()
  RETURNING result INTEGER
```

Usage

The `arr_curr()` function returns an integer value that identifies the current row of a list of rows in an `INPUT ARRAY` or `DISPLAY ARRAY` instruction. The first row is numbered 1.

Note that `arr_curr()` and `scr_line()` can return different values if the program array is larger than the screen array.

Consider using the `ui.Dialog.getCurrentRow()` method instead of `arr_curr()` when executing several list-handled instruction in parallel inside a `DIALOG` block.

downshift()
Converts a string to lowercase.

Syntax

```
downshift(
   source STRING )
  RETURNING result STRING
```

1. *source* is the character string to convert to lowercase letters.

Usage

The `downshift()` function returns a string value in which all uppercase characters in its argument are converted to lowercase.

The character conversion depends on locale settings (the LC_CTYPE environment variable). Non-alphabetic or lowercase characters are not altered.

scr_line()
Returns the index of the current row in the screen array.

Syntax

```
scr_line()
  RETURNING result INTEGER
```

Usage

The `scr_line()` function returns the index of the current row in the screen array. It is typically used inside a `DISPLAY ARRAY` or `INPUT ARRAY` statement.

> **Important:** When using new graphical objects such as TABLE containers, this function can return an invalid screen array line number, because the current row may not be visible if the user scrolls in the list with scrollbars.

Do not confuse `scr_line()` with `arr_curr()`, the first returns the index of the current row in the form screen array, and the second returns the index of the current row in the program variable.

num_args()
Returns the number of program arguments.

Syntax

```
num_args()
  RETURNING result INTEGER
```

Usage

Returns the number of arguments passed to the program.

The function returns 0 if no arguments are passed to the program.

err_get()
Returns the text corresponding to an error number.

Syntax

```
err_get(
   errnum INTEGER )
  RETURNING result STRING
```

1. *errnum* is a runtime error or an Informix® SQL error.

Usage

The `err_get()` function returns the error message corresponding to the number passed as parameter.

IBM® Informix® SQL error numbers can only be supported if the program is connected to an Informix database. Do not use this function in the context of SQL execution, when using different type of database servers.

err_print()
Prints in the error line the text corresponding to an error number.

Syntax

```
err_print(
   errnum INTEGER )
```

1. *errnum* is a runtime error or an Informix® SQL error.

Usage

The `err_print()` function displays to the screen the error message corresponding to the number passed as parameter. The message will be displayed in the error line defined by the program.

IBM® Informix® SQL error numbers can only be supported if the program is connected to an Informix database. Do not use this function when programming an application that must run with different type of database servers.

err_quit()
Prints in the error line the text corresponding to an error number and terminates the program.

Syntax

```
err_quit(
```

```
   errnum INTEGER )
```

1. *errnum* is a runtime error or an Informix® SQL error.

Usage

The `err_quit()` function prints the error message corresponding to the number passed as parameter. The message will be displayed in standard error stream and the program will terminate.

IBM® Informix® SQL error numbers can only be supported if the program is connected to an Informix database. Do not use this function when programming an application that must run with different type of database servers.

errorlog()
Copies the string passed as parameter into the error log file.

Syntax

```
errorlog(
   text STRING )
```

1. *text* is the character string to be inserted in the error log file.

Usage

The `errorlog()` function writes the passed string in the current error log file. The error log file is defined by a call to the `startlog()` function.

Use this function to identify errors in programs and to customize error handling. The error log functions can also be used to trace the way a program is used to improve it, record work habits or help to detect attempts to breach security.

fgl_buffertouched()
Returns TRUE if the input buffer was modified in the current field.

Syntax

```
fgl_buffertouched()
  RETURNING result INTEGER
```

Usage

The function returns TRUE if the input buffer has been modified after the current field was selected (i.e. got the focus).

Call this function in AFTER FIELD, AFTER INPUT, AFTER CONSTRUCT, ON KEY, ON ACTION blocks.

This function is not equivalent to FIELD_TOUCHED(): The modification status of `fgl_buffertouched()` is reset when entering a new field, while `FIELD_TOUCHED()` returns TRUE when a field was modified during the interactive instruction.

fgl_db_driver_type()
Returns the 3-letter identifier/code of the current database driver.

Syntax

```
fgl_db_driver_type()
  RETURNING drvtype CHAR(3)
```

Usage

This function can be called after connecting to a database server with the `CONNECT` or `DATABASE` instructions, in order to identify the type of the target database with the driver type.

Returned value is the 3-letter driver code, in lower case, such as "`ifx`", "`ora`", "`db2`", etc.

See the drivers table for more details about the list of database driver types.

The function returns `NULL` if there is no current database driver (i.e. if database connection is not yet established).

fgl_decimal_truncate()
Returns a decimal truncated to the precision passed as parameter.

Syntax

```
fgl_decimal_truncate(
   value DECIMAL,
   decimals INTEGER )
  RETURNING result DECIMAL
```

1. *value* is the decimal to be converted.
2. *decimals* defines the number of digits after the decimal point.

Usage

This function truncates the decimal to the number of decimal digits specified.

The value is not rounded, it is just truncated. For example, when truncating 12.345 to 2 decimal digits, the result will be 12.34, not 12.35.

fgl_decimal_sqrt()
Computes the square root of the decimal passed as parameter.

Syntax

```
fgl_decimal_sqrt(
   value DECIMAL )
  RETURNING result DECIMAL
```

1. *value* is the decimal to be computed.

fgl_decimal_exp()
Returns the value of Euler's constant (e) raised to the power of the decimal passed as parameter.

Syntax

```
fgl_decimal_exp(
   value DECIMAL )
  RETURNING result DECIMAL
```

1. *value* is the decimal to be computed.

fgl_decimal_logn()
Returns the natural logarithm of the decimal passed as parameter.

Syntax

```
fgl_decimal_logn(
```

```
    value DECIMAL )
  RETURNING result DECIMAL
```

1. *value* is the decimal to be computed.

fgl_decimal_power()
Raises decimal to the power of the real exponent.

Syntax

```
fgl_decimal_power(
      base DECIMAL,
      exp DECIMAL )
    RETURNING result DECIMAL
```

1. *base* is the decimal to be raise to the power of *exp*.
2. *exp* is the exponent.

Usage

Unlike the `**` operator, the `fgl_decimal_power()` function supports real numbers for the exponent.

fgl_dialog_getbuffer()
Returns the text of the input buffer of the current field.

Syntax

```
fgl_dialog_getbuffer()
  RETURNING result STRING
```

Usage

The `fgl_dialog_getbuffer()` function returns the content of the input buffer of the current field. It must be used in `INPUT`, `INPUT ARRAY` and `CONSTRUCT` blocks.

The function is especially useful in a `CONSTRUCT` instruction, because there is no variable associated to fields in this case.

Consider using the `ui.Dialog.getFieldBuffer()` method instead.

fgl_dialog_setbuffer()
Sets the input buffer of the current field.

Syntax

```
fgl_dialog_setbuffer(
   value STRING )
```

1. *value* is the text to set in the current input buffer.

Usage

In the default buffered input mode, this function modifies the input buffer of the current field; the corresponding input variable is not assigned. It makes no sense to call this function in `BEFORE FIELD` blocks of `INPUT` and `INPUT ARRAY`. However, if the statement is using the `UNBUFFERED` mode, the function will set both the field buffer and the program variable. If the string set by the function does not represent a valid value that can be stored by the program variable, the buffer and the variable will be set to `NULL`.

The `fgl_dialog_setbuffer()` function must be used in `INPUT`, `INPUT ARRAY` and `CONSTRUCT` blocks.

This function sets the modification flag for both `FIELD_TOUCHED()` and `fgl_buffertouched()` functions. There is a slight difference between both functions: The modification flag for `fgl_buffertouched()` is reset to `FALSE` when entering the field.

The function is especially useful in a `CONSTRUCT` instruction, because there is no variable associated to fields in this case.

fgl_dialog_getfieldname()
Returns the name of the current input field.

Syntax

```
fgl_dialog_getfieldname()
  RETURNING result STRING
```

Usage

This function returns the name of the current input field during a dialog execution. It must be use in `INPUT`, `INPUT ARRAY` or `CONSTRUCT` blocks.

Only the column part of the field name is returned (screen record name is omitted).

The `fgl_dialog_getfieldname()` is similar to the `INFIELD()` operator and `fgl_dialog_infield()` function.

fgl_dialog_infield()
This function checks for the current input field.

Syntax

```
fgl_dialog_infield(
   field-name STRING )
  RETURNING result INTEGER
```

1. *field-name* is the name if the form field.

Usage

The `fgl_dialog_infield()` function returns `TRUE` if the field name passed as the parameter is the current input field.

The function must be called in `INPUT`, `INPUT ARRAY` or `CONSTRUCT` blocks.

This function is the equivalent of the `INFIELD()` operator, except that the function takes a string expression as parameter, while the `INFIELD()` operator expects a hard-coded form field name.

fgl_dialog_setcursor()
This function sets the position of the edit cursor in the current field.

Syntax

```
fgl_dialog_setcursor(
   position INTEGER )
```

1. *position* is the edit cursor position in the text, using byte length semantics.

Usage

The `fgl_dialog_setcursor()` moves the edit cursor to the specified position in the current field. The function must be called in interactive instructions control blocks, when staying in the current field.

This function has only an effect when staying in the current field, it should not be called in an `AFTER FIELD` or `AFTER ROW` for example.

Note that you can use `FGL_DIALOG_SETSELECTION()` to select a piece of text in a field.

> **Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

fgl_dialog_setfieldorder()
This function enables or disables field order constraint.

Syntax

```
fgl_dialog_setfieldorder(
   active INTEGER )
```

1. When *active* is `TRUE`, the field order is constrained. When *active* is `FALSE`, the field order is not constrained.

Usage

Typical applications control user input with `BEFORE FIELD` and `AFTER FIELD` blocks. In many cases the field order and the sequential execution of `AFTER FIELD` blocks is important in order to validate the data entered by the user. But with graphical front ends you can use the mouse to move to a field. By default the runtime system executes all `BEFORE FIELD` and `AFTER FIELD` blocks of the fields used by the interactive instruction, from the origin field to the target field selected by mouse click. If needed, you can force the runtime system to ignore all intermediate field triggers, by calling this function with a `FALSE` attribute.

This function must be called outside interactive dialog blocks, typically at the beginning of the program.

Consider using the `Dialog.fieldOrder` parameter when all programs are affected. The FGLPROFILE profile entry is the default when the `fgl_dialog_setfieldorder()` function is not used.

Consider using `OPTIONS FIELD ORDER FORM` for new developments with graphical rendering.

fgl_dialog_setcurrline()
This function moves to a specific row in a record list.

Syntax

```
fgl_dialog_setcurrline(
  line INTEGER,
  row INTEGER )
```

1. *line* is the line number in the form screen array.
2. *row* is the row number in the program array variable.

Usage

Moves to the row / screen line specified. See fgl_set_arr_curr() for more details.

To be called during a `DISPLAY ARRAY` or `INPUT ARRAY` instruction, inside `BEFORE DISPLAY` / `BEFORE INPUT` or `ON ACTION` / `ON KEY` blocks only.

The *line* parameter is ignored in GUI mode.

fgl_dialog_getbufferstart()
Returns the row offset of the page to feed a paged display array.

Syntax

```
fgl_dialog_getbufferstart()
  RETURNING result INTEGER
```

Usage

The `FGL_DIALOG_GETBUFFERSTART()` function returns the record list offset to be used to fill a page of a `DISPLAY ARRAY` running in paged mode.

This function must be called in the context of the `ON FILL BUFFER` trigger. The returned value is undefined if the function is used outside this trigger.

fgl_dialog_getbufferlength()
Returns the number of rows to feed a paged DISPLAY ARRAY.

Syntax

```
fgl_dialog_getbufferlength()
  RETURNING result INTEGER
```

Usage

The `fgl_dialog_getbufferlength()` function returns the number of rows to be provided by the program to fill a page of a `DISPLAY ARRAY` running in paged mode.

This function must be called in the context of the `ON FILL BUFFER` trigger. The returned value is undefined if the function is used outside this trigger.

fgl_dialog_getcursor() / fgl_getcursor()
Returns the position of the edit cursor in the current field.

Syntax

```
fgl_dialog_getcursor()
  RETURNING index INTEGER
```

1. *index* is the edit cursor position in the text, using byte length semantics.

Usage

The `fgl_dialog_getcursor()` function can be used in conjunction with `fgl_dialog_getselectionend()` to get the position of the edit cursor and the piece of text that is selected in the current field.

> **Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

fgl_dialog_getkeylabel()
Returns the label associated to a key for the current interactive instruction.

Syntax

```
fgl_dialog_getkeylabel(
   keyname STRING )
  RETURNING result STRING
```

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.

Usage

The `fgl_dialog_getkeylabel()` function returns the label defined for the function or control key passed as parameter, for the current interactive instruction.

This function returns the key labels defined for the current dialog. There are different levels of key label definitions.

This function is provided for backward compatibility, use action defaults to define action view texts.

fgl_dialog_getselectionend()
Returns the position of the last selected character in the current field.

Syntax

```
fgl_dialog_getselectionend()
  RETURNING position INTEGER
```

1. *position* is the position of the last selected character in the current field text, using in byte length semantics.

Usage

The `fgl_dialog_getselectionend()` function returns the edit cursor position of the last selected character in the text of the current field.

> **Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

The function returns zero if the complete text is selected.

The edit cursor position returned by `fgl_dialog_getcursor()` will be lower as the position returned by `fgl_dialog_getselectionend()` if the text has been selected backwards.

fgl_dialog_setkeylabel()
Sets the label associated to a key for the current interactive instruction.

Syntax

```
fgl_dialog_setkeylabel(
   keyname STRING,
   label STRING )
```

1. *keyname* is the logical name of a key such as `F11` or `DELETE`,`INSERT`, `CANCEL`.
2. *label* is the text associated to the key.

Usage

The `fgl_dialog_setkeylabel()` associates a text to a function or control key. for the current dialog. Default action views (i.e. buttons that appears in the control frame of a window) will get the label displayed instead of the function or control key name.

This function defines the key labels for the current dialog. There are different levels of key label definitions.

> **Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

fgl_dialog_setselection()
Selects the text in the current field.

Syntax

```
fgl_dialog_setselection(
   cursor INTEGER,
   end INTEGER )
```

1. *cursor* defines the edit cursor position, using byte length semantics.
2. *end* defines the selection end position, using byte length semantics.

Usage

A call to `fgl_dialog_setselection(cursor, end)` sets the text selection in the current form field. The *cursor* parameter defines the character position of the edit cursor (equivalent to `fgl_dialog_getcursor()` position), while *end* defines the character position of the end of the text selection (equivalent to `fgl_dialog_getselectionend()` position).

> **Important:** When using byte length semantics, the positions are expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

cursor can be lower, greater or equal to *end*.

This function has only an effect when staying in the current field, it should not be called in an `AFTER FIELD` or `AFTER ROW` for example.

fgl_drawbox()
Draws a rectangle in the current window.

Syntax

```
fgl_drawbox(
   height INTEGER,
   width INTEGER,
   line INTEGER,
   column INTEGER,
   color INTEGER )
```

1. *height* is the height of the rectangle.
2. *width* is the width of the rectangle.
3. *line* is the horizontal coordinate of the upper side of the rectangle.
4. *column* is the vertical coordinate of the left side of the rectangle.
5. *color* is the color number (ignored).

Usage

The `fgl_drawbox()` function draws a rectangle based on the character terminal coordinates in the current open window.

Dimensions and coordinates are specified in grid cells unit (i.e. characters).

This function is provided for backward compatibility. A call to this function will be ignored if the current window is not SCREEN based. The function is supported to draw rectangles in text mode applications.

fgl_drawline()
Draws a line in the current window (TUI and traditional mode).

Syntax

```
fgl_drawline(
   column INTEGER,
   line INTEGER,
   width INTEGER,
   type CHAR(1),
   color INTEGER)
```

1. *line* is the horizontal coordinate of the upper side of the rectangle.
2. *column* is the vertical coordinate of the left side of the rectangle.
3. *width* is the width of the line.
4. *type* (ignored).
5. *color* is the color number (ignored).

Usage

The `fgl_drawline()` function draws a line based on the character terminal coordinates in the current open window.

Dimensions and coordinates are specified in grid cells unit (i.e. characters).

This function is provided for backward compatibility. A call to this function will be ignored if the current window is not SCREEN based. The function is supported to draw lines in text mode applications.

fgl_eventloop()
Waits for a user interaction event.

Syntax

```
fgl_eventloop()
   RETURNING status BOOLEAN
```

1. *status* is boolean indicating if the user event loop must continue.

Usage

The `fgl_eventloop()` function is used to implement the parallel dialog main event loop, in conjuction with `START DIALOG` / `TERMINATE DIALOG` instructions, used to register and end parallel dialogs.

The function waits until a user interaction event occurs and returns `TRUE` or `FALSE` to indicate if the event loop must continue or stop. It is typically used in a `WHILE` / `END WHILE` loop:

```
WHILE fgl_eventLoop()
END WHILE
```

Parallel dialogs are registrer with the `START DIALOG` instruction. As long as at least one parallel dialog is registered, the fgl_evenLoop() function returns `TRUE`. When the last parallel dialog is ended with a `TERMINATE DIALOG` instruction, the fgl_evenLoop() function returns `FALSE` and the even loop is stopped.

fgl_getenv()
Returns the value of the environment variable.

Syntax

```
fgl_getenv(
   variable STRING )
  RETURNING result STRING
```

1. *variable* is the name of the environment variable.

Usage

The argument of `fgl_getenv()` must be the name of an environment variable.

If the requested value exists in the current user environment, the function returns the value of that variable. If the specified environment variable is not defined, the function returns a `NULL` value. If the environment variable is defined but does not have a value assigned to it, the function returns blank spaces.

fgl_gethelp()
Returns the help text according to its identifier by reading the current help file.

Syntax

```
fgl_gethelp(
   help-id INTEGER )
  RETURNING result STRING
```

1. *help-id* is the help text identifier.

Usage

The `fgl_gethelp()` function returns the text corresponding to the help message number passed as parameter.

The text is read from the current help file. The current help file is defined by the `OPTIONS HELP FILE` instruction.

fgl_getpid()
Returns the system process identifier.

Syntax

```
fgl_getpid()
  RETURNING result INTEGER
```

Usage

The `fgl_getpid()` function returns the current process identifier. The process identifier is provided by the operating system.

fgl_getfile()
Retrieves a file from the front-end context to the virtual machine context.

Syntax

```
fgl_getfile(
   src STRING,
   dst STRING )
```

1. *src* is the path of the file to retrieve from the front-end context.
2. *dst* is the path of the file to write in the virtual machine context.

Usage

The `fgl_getfile()` function uploads a file from the front-end workstation disk to the application server disk where fglrun is executed.

> **Important:** Using this function can result in a security hole if you allow the end user to specify the file paths without control. There is not limitation on the file content or file paths: If the user executing the application on the server side is allowed to write critical server files, the program could transfer files from the client workstation and overwrite critical server files. On the other hand, critical files can be read from the client workstation and copied on the application server. It is in the hands of the programmer to implement file path and/or file content restrictions in the programs using `fgl_getfile()`.

When the front-end is located on a mobile device (GMA or GMI), the `fgl_getfile()` function can take an opaque file path as first argument, to identify a local device resource returned from a front call such as choosePhoto on page 1927, takeVideo on page 1939. This allows you to retrieve the media file into the virtual machine context, for persistent storage, and to share it with applications running on other devices. This `fgl_getfile()` feature can be used with a standalone app running on the device, or a client/server app executing on a server and displaying on the device. For more details, see Runtime images on page 787.

fgl_getkey()
Waits for a keystroke and returns the key number.

Syntax

```
fgl_getkey()
  RETURNING keynum INTEGER
```

1. *keynum* is the integer key code of the pressed key.

Usage

`fgl_getkey()` waits for a keystroke and returns the key code corresponding to the pressed physical key.

This function should only be used in text mode.

Unlike `fgl_lastkey()`, which can return a value indicating the logical effect of whatever key the user pressed,`fgl_getkey()` returns an integer representing the key code of the physical key that the user pressed. The `fgl_getkey()` function recognizes the same codes for keys that the `fgl_keyval()` function returns. Unlike `fgl_keyval()`, which can only return keystrokes that are entered during dialogs, `fgl_getkey()` can be called outside a dialog context.

fgl_getkeylabel()
Returns the default label associated to a key.

Syntax

```
fgl_getkeylabel(
   keyname STRING )
  RETURNING result STRING
```

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.

Usage

The `fgl_getkeylabel()` function returns the default label defined for the function or control key passed as parameter.

This function returns the default key labels defined for the all dialogs. There are different levels of key label definitions.

This function is provided for backward compatibility, use action defaults to define action view texts.

fgl_getresource()
Returns the value of an FGLPROFILE entry.

Syntax

```
fgl_getresource(
   name STRING )
  RETURNING result STRING
```

1. *name* is the FGLPROFILE entry name to be read.

Usage

The `fgl_getresource()` function reads the FGLPROFILE file(s) and returns the value defined for the entry passed as parameter.

If the entry does not exist in the configuration file, the function returns NULL.

Note that FGLPROFILE entry names are not case sensitive.

fgl_getversion()
Returns the product version number of Genero.

Syntax

```
fgl_getversion()
  RETURNING result STRING
```

Usage

The `fgl_getversion()` function returns the product version number the Genero Business Development Language runtime system.

> **Important:** This function is provided for debugging only; do not write business code dependent on the build number. The format of the returned value is subject of change in future versions.

fgl_getwin_height()
Returns the number of rows of the current window.

Syntax

```
fgl_getwin_height()
  RETURNING result INTEGER
```

Usage

The `fgl_getwin_height()` function returns the height of the current window, in character units.

This function is provided for text mode applications, in GUI mode, windows are re-sizeable and thus their height is variable.

fgl_getwin_width()
Returns the width of the current window as a number of columns.

Syntax

```
fgl_getwin_width()
  RETURNING result INTEGER
```

Usage

The `fgl_getwin_width()` function returns the width of the current window, in character units.

This function is provided for text mode applications, in GUI mode, windows are re-sizeable and thus their width is variable.

fgl_getwin_x()
Returns the horizontal position of the current window.

Syntax

```
fgl_getwin_x()
  RETURNING result INTEGER
```

Usage

The `fgl_getwin_x()` function returns the horizontal coordinate of the top/left corner of the current window.

This function is provided for text mode applications, in GUI mode, windows are movable and thus their position is variable.

fgl_getwin_y()
Returns the vertical position of the current window.

Syntax

```
fgl_getwin_y()
  RETURNING result INTEGER
```

Usage

The `fgl_getwin_y()` function returns the vertical coordinate of the top/left corner of the current window.

This function is provided for text mode applications, in GUI mode, windows are movable and thus their position is variable.

fgl_keyval()
Returns the key code of a logical or physical key.

Syntax

```
fgl_keyval(
   string STRING )
  RETURNING result INTEGER
```

1. *string* can be a single character, a digit, a printable symbol like @, #, $ or a special keyword such as ACCEPT.

Usage

`fgl_keyval()` can be used in form-related statements to examine a value returned by the `fgl_lastkey()` and `fgl_getkey()` functions.

Key names recognized by `fgl_keyval()` are: `ACCEPT`, `HELP`, `NEXT`, `RETURN`, `DELETE`, `INSERT`, `NEXTPAGE`, `RIGHT`, `DOWN`, `INTERRUPT`, `PREVIOUS`, `TAB`, `ESC`, `ESCAPE`, `LEFT`, `PREVPAGE`, `UP`, `F1` through `F64`, `CONTROL-`*`character`* (where *character* can be any letter except A, D, H, I, J, L, M, R, or X).

The function returns NULL if the parameter does not correspond to a valid key.

If you specify a single character, `fgl_keyval()` considers the case. In all other instances, the function ignores the case of its argument, which can be uppercase or lowercase letters.

To determine whether the user has performed an action, such as inserting a row, specify the logical name of the action (such as INSERT) rather than the name of the physical key (such as F1). For example, the logical name of the Accept action is ACCEPT, while the default physical key is ESCAPE. To test if the key most recently pressed by the user corresponds to the Accept action, specify `fgl_keyval("ACCEPT")` rather than `fgl_keyval("ESCAPE")` or `fgl_keyval("ESC")`. Otherwise, if a key other than ESCAPE is set as the Accept key and the user presses that key,`FGL_LASTKEY()` does not return a code equal to `fgl_keyval("ESCAPE")`.

This function is provided for backward compatibility especially for TUI mode applications. `fgl_keyval()` is well supported in text mode, but this function can only be emulated in GUI mode, because the front-ends communicate with the runtime system with other events as keystrokes.

fgl_lastkey()
Returns the key code corresponding to the logical key that the user most recently typed in the form.

Syntax

```
fgl_lastkey()
  RETURNING result INTEGER
```

Usage

The `fgl_lastkey()` function returns a numeric code corresponding to the user's last keystroke before the function was called. For example, if the last key that the user pressed was a lowercase *s*, the function returns the code 115 (i.e. the ASCII character set code).

The value of `fgl_lastkey()` is undefined in a `MENU` statement.

The function returns `NULL` if no key has been pressed.

It is not required to know the specific key codes returned by `fgl_lastkey()`: The `FGL_KEYVAL()` function can be used to compare the key code of the last key pressed. The `FGL_KEYVAL()` function lets you compare the last key pressed with a logical of physical key. For example, you do not need to know the physical key defined to validate a dialog, you can use the logical name "accept" instead. For a complete list of key codes and logical key names, see the Key code table.

Pay attention to the fact that this function is provided for backward compatibility. The abstract user interface protocol is based on logical events, not only key events. For example, in GUI mode, when selecting a new row with the mouse in a table, there is no key press as when moving in a static screen array in TUI mode. However, the runtime system tries to emulate as much as possible keystrokes from non-keystroke events.

fgl_putfile()
Transfers a file from the virtual machine context to the front end context.

Syntax

```
fgl_putfile(
   src STRING,
   dst STRING)
```

1. *src* is the path to the file to transmit from the virtual machine context.
2. *dst* is the path to the file to write in the front end context.

Usage

The `fgl_putfile()` function downloads a file from the application server disk where fglrun is executed to the front-end workstation disk.

> **Important:** Using this function can result in a security hole if you allow the end user to specify the file paths without control. There is not limitation on the file content or file paths: If the user executing the application on the server side is allowed to read critical server files, the program could transfer these files on the client workstation. On the other hand, critical files can be written on the client workstation. It is in the hands of the programmer to implement file path and/or file content restrictions in the programs using `fgl_putfile()`.

fgl_report_print_binary_file()
Prints a file containing binary data during a report.

Syntax

```
fgl_report_print_binary_file(
   filename STRING )
```

1. *filename* is the name of the binary file.

Usage

This function prints a file containing binary data during a report.

This function is provided for backward compatibility and must only be using inside a `REPORT` routine.

fgl_report_set_document_handler()
Redirects the next report to an XML document handler.

Syntax

```
fgl_report_set_document_handler(
```

```
   handler om.SaxDocumentHandler )
```

1. *handler* is the document handler variable.

Usage

This function attaches the specified XML document handler to the next executed report, it must be called before the execution of a report.

This function is provided for backward compatibility, you should use the `TO XML HANDLER` of `START REPORT` instead.

fgl_setkeylabel()
Sets the default label associated to a key.

Syntax

```
fgl_setkeylabel(
   keyname STRING,
   label STRING )
```

1. *keyname* is the logical name of a key such as `F11` or `DELETE`, `INSERT`, `CANCEL`.
2. *label* is the text associated to the key.

Usage

`fgl_setkeylabel()` associates a text to a function or control key. Default action views (i.e. buttons that appears in the control frame of a window) will get the label displayed instead of the function or control key name.

This function defines the default key labels for all dialogs. There are different levels of key label definitions.

> **Note:** This feature is supported for backward compatibility. Consider using action attributes to define accelerator keys and decorate actions.

fgl_scr_size()
Returns the size of the specified screen array in the current form.

Syntax

```
fgl_scr_size(
   screen-array STRING )
  RETURNING result INTEGER
```

1. *screen-array* is the name of a screen-array in the current displayed form.

Usage

The `fgl_scr_size()` function takes the name of a screen array as parameter identifying an array in the currently opened form and returns an integer that corresponds to the number of screen records in that screen array.

This function is typically used with traditional text mode forms having screen arrays with a constant size, to display data in screen array rows with the `DISPLAY TO` instruction.

For modern GUI applications, consider using the UNBUFFERED mode in dialogs, to get automatic form field synchronization with program variables.

Error -1108 will be raised if the passed screen-array does not exits in the current form, and error -1114 is returned if no form is currently displayed.

fgl_setsize()
Sets the size of the main application window.

Syntax

```
fgl_setsize(
   height INTEGER,
   width INTEGER )
```

1. *height* is the number of lines of the window.
2. *width* is the number of columns of the window.

Usage

This function defines the size of the main window when using the traditional GUI mode.

fgl_settitle()
Sets the title of the current application window.

Syntax

```
fgl_settitle(
   label STRING )
```

1. *label* is the text of the title.

Usage

The `fgl_settitle()` function defines the title of the current window, as well as the default title for new created windows.

This function is provided for backward compatibility, the title of a window can be defined with the TEXT attribute of a `LAYOUT` section.

fgl_setenv()
Sets the value of an environment variable.

Syntax

```
fgl_setenv(
   variable STRING,
   value STRING )
```

1. *variable* is the name of the environment variable.
2. *value* is the value to be set.

Usage

The `fgl_setenv()` function sets or modifies the value of an environment variable.

There is a little difference between Windows™ and UNIX™ platforms when passing a NULL as the *value* parameter: On Windows platforms, the environment variable is removed, while on UNIX, the environment variable gets an empty value (i.e. it is not removed from the environment).

> **Important:** You may experience unexpected results if you change environment variables that are already used by the current program - for example, when you are connected to INFORMIX and you change the INFORMIXDIR environment variable.

fgl_set_arr_curr()
Moves to a specific row in a record list.

Syntax

```
fgl_set_arr_curr(
   row INTEGER )
```

1. *row* is the row number is the program array variable.

Usage

This function is typically used to control navigation in a `DISPLAY ARRAY` or `INPUT ARRAY`, within an `ON ACTION` or `ON KEY` block. The function can also be used inside `BEFORE DISPLAY` or `BEFORE INPUT` blocks, to jump to a specific row when the dialog starts. You should not use this function in an other context.

Control blocks like `BEFORE ROW` and field/row validation in `INPUT ARRAY` are performed, as if the user moved to another row, except when the function is called in `BEFORE DISPLAY` or `BEFORE INPUT`.

When a new row is reached by using with this function, the first editable field gets the focus.

An alternative to the `fgl_set_arr_curr()` function is the ui.Dialog.setCurrentRow() method; However, the dialog class method will be used in a different programming pattern, as it does not trigger the control blocks like the built-in function.

fgl_system()
Runs a command on the application server.

Syntax

```
fgl_system(
   command STRING )
```

1. *command* is the command line to be executed on the server.

Usage

The `fgl_system()` function suspends the execution of the program and executes the command passed as parameter on the application server where fglrun is executed. The command is executed in a new shell and the program is suspended until the command terminates.

When running the program in TUI mode, the terminal is switched to line mode before executing the command passed to the `fgl_system()` function.

This function is provided for backward compatibility. In older versions, the function could raise a terminal emulator on the front-end to show the command output on the workstation. This feature is no longer supported.

fgl_width()
Returns the number of columns needed to represent the printed version of the expression.

Syntax

```
fgl_width(
   expression STRING )
  RETURNING result INTEGER
```

1. *expression* is any valid string expression.

Usage

The `fgl_width()` function returns the number of columns that will be used if you display *expression* on a text terminal.

If the parameter is `NULL`, the function returns zero.

The number of columns used by a character depends on the glyph (i.e. the graphical symbol used to draw the character on the screen). For example, an ASCII character like A uses one column, while one Chinese ideogram uses 2 columns (i.e. on a text terminal, the size of one Chinese ideogram takes the same size as AB).

Trailing blanks are counted in the length of the string.

fgl_window_getoption()
Returns attributes of the current window.

Syntax

```
fgl_window_getoption(
   attribute STRING )
  RETURNING result STRING
```

1. *attribute* is the name of a window attribute.

Usage

The `fgl_window_getoption()` function returns the value of the window attribute passed as parameter.

Possible parameters are: `name`, `x`, `y`, `width`, `height`, `formline`, `messageline`.

This function is provided for backward compatibility, do not use this function in modern GUI applications.

length()
Returns the number of the character string passed as parameter.

Syntax

```
length(
   expression STRING )
  RETURNING result INTEGER
```

1. *expression* is any valid character string expression supported by the language.

Usage

The `length()` function counts the length of a character string.

If the parameter is `NULL`, the function returns zero.

> **Important:** When using byte length semantics, the length is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

set_count()
Defines the number of rows containing explicit data in a static array used by the next dialog.

Syntax

```
set_count(
   nbrows INTEGER )
```

1. *nbrows* defines the number of explicit rows in the static array.

Usage

When using a static array in an `INPUT ARRAY` (with `WITHOUT DEFAULTS` clause) or a `DISPLAY ARRAY` statement, you must specify the number of rows in the array which contain explicit data. In typical applications, these array elements contain the values fetched from a `SELECT` statement.

`set_count()` must be called before a `DISPLAY ARRAY` or `INPUT ARRAY` statement.

The number of rows can also specified with the `COUNT` attribute of `INPUT ARRAY` and `DISPLAY ARRAY` statements.

When using a dynamic array, the number of rows is implicitly defined by the array.

showhelp()
Displays a runtime help text.

Syntax

```
showhelp(
   helpnum INTEGER )
```

1. *helpnum* is the help message number in the current help file.

Usage

The `showhelp()` function displays a runtime help text, corresponding to its specified argument, from the current help file defined by the `OPTIONS HELP FILE` instruction.

In GUI mode, the help text will be displayed in a new popup window. In TUI mode, the help text is displayed in a the whole screen.

startlog()
Initializes error logging and opens the error log file passed as the parameter.

Syntax

```
startlog(
   filename STRING )
```

1. *filename* is the name of the error log file.

Usage

Call `startlog()` in the `MAIN` program block to open or create an error log file. After `startlog()` has been invoked, a record of every subsequent error that occurs during the program execution is written in the error log file.

The format of the error records appended to the error log file after each subsequent error is as follows:

```
Date: 03/06/99 Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX column.
SYSTEM error number -100
ISAM error: duplicate value for a record with unique key.
```

To report specific application errors, use the `errorlog()` function to make an entry in the error log file.

If the argument of `startlog()` is not the name of an existing file, `startlog()` creates a new one. If the file already exists, `startlog()` opens it and positions the file pointer so that subsequent error messages can be appended to this file.

Example

```
CALL startlog("/var/myapp/logs/error-" || fgl_getpid() ||
 ".log")
...
CALL errorlog("The current user is not allowed to perform order
 validation")
```

upshift()
Converts a string to uppercase.

Syntax

```
upshift(
   source STRING )
  RETURNING result STRING
```

1. *source* is the character string to convert to uppercase letters.

Usage

The `upshift()` function returns a string value in which all lowercase characters in its argument are converted to uppercase.

The character conversion depends on locale settings (the LC_CTYPE environment variable). Non-alphabetic or uppercase characters are not altered.

List of desupported built-in functions

Table 355: Desupported built-in functions

Function	Description
`FGL_FORMFIELD_GETOPTION()`	Returns attributes of a specified form field.
`FGL_GETUITYPE()`	Returns the type of the front end.
`FGL_WINDOW_OPEN()`	Opens a new window with coordinates and size.
`FGL_WINDOW_OPENWITHFORM()`	Opens a new window with coordinates and form.
`FGL_WINDOW_CLEAR()`	Clears the window having the name that is passed as a parameter.
`FGL_WINDOW_CLOSE()`	Closes the window having the name that is passed as a parameter.
`FGL_WINDOW_CURRENT()`	Makes current the window having the name that is passed as a parameter.

The key code table

This table lists internal key codes. Avoid hard-coding these numbers in your sources; otherwise the source will not be compatible with future versions the language.

Always use the FGL_KEYVAL( ) function instead.

Table 356: Internal key codes

Value	Key name	Description
1 to 26	Control-*x*	Control key, where *x* is the any letter from A to Z. The key code corresponding to Control-A is 1, Control-B is 2, etc.
others < 256	ASCII chars	Other codes correspond to the ASCII characters set.
2000	up	The up-arrow logical key.
2001	down	The down-arrow logical key.
2002	left	The left-arrow logical key.
2003	right	The right-arrow logical key.
2005	nextpage	The next-page logical key.
2006	prevpage	The previous-page logical key.
2008	help	The help logical key.
2011	interrupt	The interrupt logical key.
2020	home	The home logical key.
2021	end	The end logical key.
2016	accept	The accept logical key.
2017	backspace	The backspace logical key.
3000 to 3255	F*x*	Function key, where *x* is the number of the function key. The key code corresponding to a function key F*x* is 3000+*x*-1, for example, 3011 corresponds to F12.

Utility functions

A utility function is a function provided in a separate library; it is not built in the runtime system.

To use a utility function, declare the module where the function is defined with the IMPORT FGL instruction:

```
IMPORT FGL fgldialog
...
   CALL fgl_winmessage( ... )
```

For backward compatibility, utility function are also grouped in a 42x library named libfgl4js.42x, which can be linked to your programs.

The 42x library file, 42m modules and 42f forms are located in FGLDIR/lib. The sources of the utility functions and form files are provided in the FGLDIR/src directory.

- Common dialog utility functions (IMPORT FGL fgldialog) on page 1668

- Database utility functions (IMPORT FGL fgldbutl) on page 1672
- Front-end dialog utility functions (IMPORT FGL fglwinexec) on page 1676

Common dialog utility functions (IMPORT FGL fgldialog)

Table 357: Common dialog utility functions (fgldialog.4gl)

Function	Description
`fgl_winbutton( title STRING, text STRING, default STRING, buttons STRING, icon STRING, danger SMALLINT ) RETURNING result STRING`	Displays an interactive message box containing multiple choices, in a popup window.
`fgl_winmessage( title STRING, text STRING, icon STRING )`	Displays an interactive message box containing text and OK button.
`fgl_winprompt( x INTEGER, y INTEGER, text STRING, default STRING, length INTEGER, type INTEGER ) RETURNING value STRING`	Displays a dialog box containing a field that accepts a value.
`fgl_winquestion( title STRING, text STRING, default STRING, buttons STRING, icon STRING, danger SMALLINT ) RETURNING value STRING`	Displays an interactive message box with configurable Ok/Yes/No/Cancel/Ignore/Abort/Retry buttons.
`fgl_winwait( text STRING )`	Displays an interactive message box and waits for user validation.

fgl_winbutton()
Displays an interactive message box containing multiple choices, in a popup window.

Syntax

```
fgl_winbutton(
   title STRING,
   text STRING,
   default STRING,
   buttons STRING,
```

```
   icon STRING,
   danger SMALLINT )
  RETURNING result STRING
```

1. title defines the title of the message window.
2. text specifies the string displayed in message window.
3. default indicates the default button to be pre-selected.
4. buttons defines a set of button labels separated by "|".
5. icon is the name of the icon to be displayed.
6. danger (for X11 only), number of the warnings item. Otherwise, this parameter is ignored.

Usage

Use the `fgl_winbutton()` function to open a message box and let the end user select an option in a set of buttons. The function returns the label of the button which has been selected by the user.

Use '\n' in *text* to separate lines (this does not work in TUI mode).

Supported names for the *icon* parameter are: `information`, `exclamation`, `question`, `stop`.

You can define up to 7 buttons that each have 10 characters.

If two buttons start with the same letter, the user will not be able to select one of them in the TUI mode.

The "&" before a letter for a button is displayed in TUI mode, or underlines the next letter in graphical front-ends.

This function is provided for backward compatibility, use a menu with "dialog" style instead.

Example

```
IMPORT FGL fgldialog
MAIN
  DEFINE answer STRING
  LET answer = fgl_winbutton( "Media selection", "What is your
 favorite media?",
      "Lynx", "Floppy Disk|CD-ROM|DVD-ROM|Other", "question", 0)
  DISPLAY "Selected media is: " || answer
END MAIN
```

fgl_winmessage()

Displays an interactive message box containing text and OK button.

Syntax

```
fgl_winmessage(
   title STRING,
   text STRING,
   icon STRING )
```

1. *title* defines message box title.
2. *text* is the text displayed in the message box. Use '\n' to separate lines.
3. *icon* is the name of the icon to be displayed.

Usage

The `fgl_winmessage()` function displays a message box to the end user.

Important: With front-ends implementing this function with the system dialog box API creating a modal window, the end user will have to close the modal window first, before continuing within the window of another program. Consider using a menu with "dialog" style instead, to not block other programs.

Supported names for the *icon* parameter are: `information`, `exclamation`, `question`, `stop`. Note that on some front-ends such as iOS devices, the native message popup window does not display an image.

On front-ends using a system dialog box API, the OK buttons will be automatically localized according to the operating system language settings. On other front-ends, the option buttons will be decorated according to action default settings.

Example

```
IMPORT FGL fgldialog
MAIN
  CALL fgl_winmessage( "Title", "This is a critical message.",
 "stop")
END MAIN
```

fgl_winprompt()
Displays a dialog box containing a field that accepts a value.

Syntax

```
fgl_winprompt(
   x INTEGER,
   y INTEGER,
   text STRING,
   default STRING,
   length INTEGER,
   type INTEGER )
  RETURNING value STRING
```

1. *x* is the column position in characters.
2. *y* is the line position in characters.
3. *text* is the message shown in the box.
4. *default* is the default value.
5. *length* is the maximum length of the input value.
6. *type* is the data type of the return value.
7. *value* is the value entered by the user.

Usage

The `fgl_winprompt()` function allows the end user to enter a value.

This function is provided for backward compatibility, you can also use your own input dialog with a customized form to get a value from the user. Or use the standard `PROMPT` instruction.

Possible values for the type parameter are: `0=CHAR, 1=SMALLINT, 2=INTEGER, 7=DATE, 255=invisible`

Avoid passing `NULL` values.

Example

```
IMPORT FGL fgldialog
MAIN
```

```
  DEFINE answer DATE
  LET answer = fgl_winprompt( 10, 10, "Today", DATE, 10, 7 )
  DISPLAY "Today is " || answer
END MAIN
```

fgl_winquestion()
Displays an interactive message box with configurable Ok/Yes/No/Cancel/Ignore/Abort/Retry buttons.

Syntax

```
fgl_winquestion(
   title STRING,
   text STRING,
   default STRING,
   buttons STRING,
   icon STRING,
   danger SMALLINT )
  RETURNING value STRING
```

1. *title* is the message box title.
2. *text* is the message displayed in the message box. Use '\n' to separate lines (does not work on ASCII client).
3. *default* defines the default button that is preselected.
4. *buttons* defines the options. Must be a pipe-separated list of 2 or three options : ok, yes, no, cancel, abort, retry, ignore.
5. *icon* is the name of the icon to be displayed.
6. *danger* is supported for backward compatibility. This parameter is ignored.

Usage

The `fgl_winquestion()` function shows a question message box to the end user and waits for an answer.

> **Important:** With front-ends implementing this function with the system dialog box API creating a modal window, the end user will have to close the modal window first, before continuing within the window of another program. Consider using a menu with "dialog" style instead, to not block other programs.

The function returns the label of the option which has been selected by the user.

Supported names for the *icon* parameter are: `information`, `exclamation`, `question`, `stop`. Note that on some front-ends such as iOS devices, the native message popup window does not display an image.

The *buttons* parameter defines the list of options that the user can select. Possible values are: `ok`, `yes`, `no`, `cancel`, `abort`, `retry`, `ignore`. You must specify a pipe-separated list of options, with a maximum of 3 options. For example: `"ok"`, `"yes|no"`, `"yes|no|cancel"`, `"abort|retry|ignore"`.

> **Important:** To display the popup window of this API, desktop and mobile front-ends use the platform specific message box API, with a predefined set of buttons. Some non-standard option combinations may not be supported, such as `"ok|yes|abort"`. Further, the order of the buttons depends also from platform standards. For example, with `"abort|retry|ignore"`, the buttons can appear in the following order: `[Retry] [Ignore] [Abort]`.

On front-ends using a system dialog box API, the option buttons will be automatically localized according to the operating system language settings. On other front-ends, the option buttons will be decorated according to action default settings.

Example

```
IMPORT FGL fgldialog
MAIN
  DEFINE answer STRING
  LET answer = "yes"
  WHILE answer = "yes"
    LET answer = fgl_winquestion(
        "Procedure", "Would you like to continue ? ",
        "cancel", "yes|no|cancel", "question", 0)
  END WHILE
  IF answer = "cancel" THEN
     DISPLAY "Canceled."
  END IF
END MAIN
```

fgl_winwait()
Displays an interactive message box and waits for user validation.

Syntax

```
fgl_winwait(
   text STRING )
```

1. *text* is the message displayed in the message box. Use '\n' to separate lines (not working on ASCII client).

Usage

The `fgl_winwait()` function displays a message to the end user and waits until the user presses the OK button.

> **Important:** With front-ends implementing this function with the system dialog box API creating a modal window, the end user will have to close the modal window first, before continuing within the window of another program. Consider using a menu with "dialog" style instead, to not block other programs.

Database utility functions (IMPORT FGL fgldbutl)

Table 358: Database utility functions (fgldbutl.4gl)

Function	Description
`db_get_database_type()` `  RETURNING result STRING`	Returns the database type for the current connection.

`db_get_sequence(`
`   id STRING )`
`  RETURNING result BIGINT`
	Generates a new sequence for a given identifier.

`db_start_transaction()` `  RETURNING result INTEGER`	Starts a nested transaction call.
`db_finish_transaction(`	Terminates a nested transaction call.

Function	Description
commit INTEGER) RETURNING *result* INTEGER	
db_is_transaction_started() RETURNING *result* INTEGER	Indicates whether a nested transaction call is started.

db_get_database_type()
Returns the database type for the current connection.

Syntax

```
db_get_database_type()
  RETURNING result STRING
```

Usage

After connecting to the database, you can get the type of the database server with this function.

Important: This function is deprecated, use the `fgl_dbdriver_type()` function instead.

Table 359: Codes returned by db_get_database_type() per database type

Code	Description
ASE	Sybase ASE
DB2	IBM® DB2®
IFX	IBM® Informix®
MYS	Oracle MySQL
MSV	Microsoft™ SQL Server
ORA	Oracle Database
PGS	PostgreSQL

db_get_sequence()
Generates a new sequence for a given identifier.

Syntax

```
db_get_sequence(
   id STRING )
  RETURNING result BIGINT
```

1. *id* is the identifier of the sequence.
2. *result* is the new generated sequence.

Usage

This function generates a new sequence from a register table created in the current database.

Important:

1. Needs a database table called SEQREG.

2. The function must be called inside a transaction block.

The table must be created as follows:

```
CREATE TABLE seqreg (
  sr_name VARCHAR(30) NOT NULL,
  sr_last BIGINT NOT NULL,
  PRIMARY KEY (sr_name)
)
```

Each time you call this function, the sequence is incremented in the database table and returned by the function.

It is mandatory to use this function inside a transaction block, in order to generate unique sequences.

Example

```
IMPORT FGL fgldbutl
MAIN
  DEFINE ns BIGINT, s INTEGER
  DATABASE mydb
  BEGIN WORK
  LET ns = db_get_sequence("mytable")
  INSERT INTO mytable VALUES ( ns, 'a new sequence' )
  COMMIT WORK
END MAIN
```

db_start_transaction()
Starts a nested transaction call.

Syntax

```
db_start_transaction()
  RETURNING result INTEGER
```

1. *result* is the SQL execution status or the transaction start. Zero indicates success.

Usage

On most database engines, you can only have a unique transaction, that is started with `BEGIN WORK` and ended with `COMMIT WORK` or `ROLLBACK WORK`. But in a some cases, you may need to do complex nested function calls, executing several SQL instruction that must all be grouped in a single transaction. The nested transaction utility functions help you to implement this.

With this nested transaction technique, you encapsulate transaction start and end withing the utility function. Custom functions doing SQL operations can then be reused in different parts of your application: If the caller does not start the transaction, the called function will automatically start and end the transaction.

The `db_start_transaction()` function encapsulates the `BEGIN WORK` instruction to start a transaction, in order to implement nested transactions.

> **Note:** These transaction encapsulation functions are provided for special cases, where the function call graph is complex. In general, you should simply use the standard `BEGIN WORK` / `COMMIT WORK` / `ROLLBACK WORK` instructions to implement transaction blocks.

These transaction management functions execute a real transaction instruction at the boundaries of the subsequent start/finish calls.

Example

```
IMPORT FGL fgldbutl

MAIN
  DEFINE s INTEGER
  DATABASE mydb
  LET s = db_start_transaction() -- real BEGIN WORK
  IF s != 0 THEN DISPLAY "error 1" END IF
  WHENEVER ERROR CONTINUE
  UPDATE customer SET cust_name = 'Undef'
  WHENEVER ERROR STOP
  LET s = SQLCA.SQLCODE
  IF s != 0 THEN
     DISPLAY "error 2"
  ELSE
     LET s = do_update()
     IF s != 0 THEN DISPLAY "error 3" END IF
  END IF
  LET s = db_finish_transaction(s==0) -- real COMMIT or ROLLBACK
 WORK
  IF s != 0 THEN DISPLAY "error 4" END IF
END MAIN

FUNCTION do_update()
  DEFINE s INTEGER
  LET s = db_start_transaction() -- no SQL command (nested)
  IF s != 0 THEN
     DISPLAY "error 1.1"
  ELSE
     WHENEVER ERROR CONTINUE
     UPDATE customer SET cust_status = 'X'
     WHENEVER ERROR STOP
     LET s = SQLCA.SQLCODE
     IF s != 0 THEN
        DISPLAY "error 1.2"
     END IF
  END IF
  LET s = db_finish_transaction(s==0) -- no SQL command (nested)
  IF s != 0 THEN DISPLAY "error 1.3" END IF
  RETURN s
END FUNCTION
```

db_finish_transaction()
Terminates a nested transaction call.

Syntax

```
db_finish_transaction(
   commit INTEGER )
  RETURNING result INTEGER
```

1. *commit* is a boolean that indicates whether the transaction must be committed.
2. *result* is the SQL execution status or the commit or rollback. Zero indicates success.

Usage

This function encapsulates the `COMMIT WORK` or `ROLLBACK WORK` instructions to end a transaction.

When the number of calls to `DB_START_TRANSACTION()` matches, this function executes a `COMMIT WORK` if the passed parameter is `TRUE`; if the passed parameter is `FALSE`, it executes a `ROLLBACK WORK`.

If the number of start/finish calls does not match, the function does nothing.

db_is_transaction_started()
Indicates whether a nested transaction call is started.

Syntax

```
db_is_transaction_started()
  RETURNING result INTEGER
```

- *result* is a boolean value that indicates if a nested transaction call sequence was started.

Usage

The function returns `TRUE` if a transaction was started with `db_start_transaction()`, and was not yet finished with a call to the `db_finish_transaction()` function.

Front-end dialog utility functions (IMPORT FGL fglwinexec)

Table 360: Front-end-side dialog functions (fglwinexec.4gl) (deprecated: use ui.Interface.frontCall() instead)

Function	Description
`winopendir( dirname STRING, caption STRING ) RETURNING result STRING`	Opens a dialog window to get a directory path on the front-end workstation.
`winopenfile( dirname STRING, typename STRING, extlist STRING, caption STRING ) RETURNING result STRING`	Opens a dialog window to get a file to be read on the front-end workstation.
`winsavefile( dirname STRING, typename STRING, extlist STRING, caption STRING ) RETURNING result STRING`	Opens a dialog window to get a file path to save data on the front-end workstation.
`winshellexec( filename STRING ) RETURNING result INTEGER`	Opens a document on the workstation where the Windows™ front end runs. Microsoft™ Windows™ only!
`winexecwait(`	Executes a program on the workstation where the Windows™ front-end runs and waits for termination.

Function	Description
command STRING) RETURNING *result* INTEGER	Microsoft™ Windows™ only!
winexec(*command* STRING) RETURNING *result* INTEGER	Executes a program on the workstation where the Windows™ front end runs and returns immediately. Microsoft™ Windows™ only!

winopendir()
Opens a dialog window to get a directory path on the front-end workstation.

Syntax

```
winopendir(
   dirname STRING,
   caption STRING )
  RETURNING result STRING
```

1. *dirname* is the default path to be displayed in the dialog window.
2. *caption* is the label to be displayed.

Usage

This function opens a dialog window to let the user select a directory path on the front end workstation file system.

The function returns the directory path on success.

The function returns NULL if a problem has occurred or if the user canceled the dialog.

> **Important:** This function is provided for backward compatibility and should be avoided to run your programs with different sort of front-ends. It must be called after the front-end connection was established.

winopenfile()
Opens a dialog window to get a file to be read on the front-end workstation.

Syntax

```
winopenfile(
   dirname STRING,
   typename STRING,
   extlist STRING,
   caption STRING )
  RETURNING result STRING
```

1. *dirname* is the default path to be displayed in the dialog window.
2. *typename* is the name of the file type to be displayed.
3. *extlist* is a blank-separated list of file extensions defining the file type.
4. *caption* is the label to be displayed.

Usage

This function opens a dialog window to let the user select a file path on the front end workstation file system, in order to open the file.

The function returns the file path on success.

The function returns NULL if a problem has occurred or if the user canceled the dialog.

> **Important:** This function is provided for backward compatibility and should be avoided to run your programs with different sort of front-ends. It must be called after the front-end connection was established.

winsavefile()
Opens a dialog window to get a file path to save data on the front-end workstation.

Syntax

```
winsavefile(
   dirname STRING,
   typename STRING,
   extlist STRING,
   caption STRING )
  RETURNING result STRING
```

1. *dirname* is the default path to be displayed in the dialog window.
2. *typename* is the name of the file type to be saved.
3. *extlist* is a blank separated list of file extensions defining the file type.
4. *caption* is the label to be saved.

Usage

This function opens a dialog window to let the user select a file path on the front end workstation file system, in order to save the file.

The function returns the file path on success.

The function returns NULL if a problem has occurred or if the user canceled the dialog.

> **Important:** This function is provided for backward compatibility and should be avoided to run your programs with different sort of front-ends. It must be called after the front-end connection was established.

winexec() MS Windows™ FE Only!
Executes a program on the workstation where the Windows™ front end runs and returns immediately.

Syntax

```
winexec(
   command STRING )
  RETURNING result INTEGER
```

1. *command* is the command to be executed on the front end.

Usage

The function executes the program on the Windows™ front end and returns the control to the program without waiting.

> **Important:** This function is provided for backward compatibility and should be avoided to run your programs with different sort of front-ends. It must be called after the front-end connection was established.

winexecwait() MS Windows™ FE Only!
Executes a program on the workstation where the Windows™ front-end runs and waits for termination.

Syntax

```
winexecwait(
   command STRING )
  RETURNING result INTEGER
```

1. *command* is the command to be executed on the front end.

Usage

The function executes the program on the Windows™ front end and waits for its termination.

> **Important:** This function is provided for backward compatibility and should be avoided to run your programs with different sort of front-ends. It must be called after the front-end connection was established.

winshellexec() MS Windows™ FE Only!
Opens a document on the workstation where the Windows™ front end runs.

Syntax

```
winshellexec(
   filename STRING )
  RETURNING result INTEGER
```

1. *filename* is the file to be opened on the front end.

Usage

The function opens a document on the Windows™ front end without waiting.

> **Important:** This function is provided for backward compatibility and should be avoided to run your programs with different sort of front-ends. It must be called after the front-end connection was established.

vCard utility functions (IMPORT FGL VCard)

Table 361: vCard utility functions (VCard.4gl)

Function	Description

`PUBLIC TYPE VCAddress RECORD`
`  PostOfficeBox,`
`  ExtendedAddress, --  apartment or suite number`
`  Street,`
`  City,`
`  State,`
`  ZIP,`
`  Country STRING`
`  -- , CountryCode STRING -- X-ABADR:de`
	The VCAddress structured type to hold vCard address data.

Function	Description
`  END RECORD`	

`PUBLIC TYPE VCName RECORD`
`    FirstName,`
`    LastName,`
`    MiddleName,`
`    Prefix,`
`    Suffix STRING`
`    --, FormattedName STRING`
`  END RECORD`
	The VCName structured type to hold vCard data related to the person name.

`PUBLIC TYPE VCPerson RECORD`
`    FirstName STRING, -- N[1]`
`    LastName STRING, -- N[2]`
`    MiddleName STRING, -- N[3]`
`    Prefix STRING, -- N[4]`
`    Suffix STRING, -- N[5]`
`    formattedName STRING, -- FN`
`    nickname STRING, -- NICKNAME`
`    jobTitle STRING, -- TITLE`
`    organization STRING, --`
` ORG.value[1]`
`    department STRING, --`
` ORG.value[2]`
`    birthday STRING, -- BDAY`
`    note STRING, -- NOTE`
`    address DYNAMIC ARRAY OF RECORD`
`        type STRING,`
`        PostOfficeBox, -- ADR[1]`
`        ExtendedAddress, -- ADR[2]`
`        Street, -- ADR[3]`
`        City, -- ADR[4]`
`        State, -- ADR[5]`
`        ZIP, -- ADR[6]`
`        Country STRING   -- ADR[7]`
`    END RECORD,`
`    phone DYNAMIC ARRAY OF RECORD`
`        type STRING,`
`        number STRING -- TEL`
`    END RECORD,`
`    email DYNAMIC ARRAY OF RECORD`
`        type STRING,`
`        value STRING -- EMAIL`
`  END RECORD`
	The VCPerson structured type to hold vCard data.

`format_person(`
`   person VCPerson )`
`  RETURNING result STRING`
	Converts a VCPerson record to a vCard string representation vCard.

`scan_address(`
`   source STRING,`
`   type STRING )`
	Extracts an address from a string representing a vCard.

Function	Description
RETURNING *address* VCAddress	
scan_email(*source* STRING, *type* STRING) RETURNING *email* STRING	Extracts an email from a string representing a vCard.
scan_person(*source* STRING) RETURNING *person* VCPerson	Extracts person's data from a string representing a vCard.
scan_phone(*source* STRING, *type* STRING) RETURNING *phone* STRING	Extracts a phone number from a string representing a vCard.

VCAddress type
The VCAddress structured type to hold vCard address data.

Syntax

```
PUBLIC TYPE VCAddress RECORD
    PostOfficeBox,
    ExtendedAddress, --  apartment or suite number
    Street,
    City,
    State,
    ZIP,
    Country STRING
    -- , CountryCode STRING -- X-ABADR:de
  END RECORD
```

Usage

This type defines a record structure to hold vCard address information. It is used for values returned by the scan_address() function.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE a VCard.VCAddress
  LET a.Street = "Sunset Bld"
END MAIN
```

VCName type
The VCName structured type to hold vCard data related to the person name.

Syntax

```
PUBLIC TYPE VCName RECORD
    FirstName,
```

```
    LastName,
    MiddleName,
    Prefix,
    Suffix STRING
    --, FormattedName STRING
  END RECORD
```

Usage

This type defines a record structure to hold vCard information related to the person name. It is used for values returned by the scan_name() function.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE n VCard.VCName
  LET n.FirstName = "Hans"
  LET n.LastName = "Mustermann"
END MAIN
```

VCPerson type
The VCPerson structured type to hold vCard data.

Syntax

```
PUBLIC TYPE VCPerson RECORD
    FirstName STRING, -- N[1]
    LastName STRING, -- N[2]
    MiddleName STRING, -- N[3]
    Prefix STRING, -- N[4]
    Suffix STRING, -- N[5]
    formattedName STRING, -- FN
    nickname STRING, -- NICKNAME
    jobTitle STRING, -- TITLE
    organization STRING, -- ORG.value[1]
    department STRING, -- ORG.value[2]
    birthday STRING, -- BDAY
    note STRING, -- NOTE
    address DYNAMIC ARRAY OF RECORD
        type STRING,
        PostOfficeBox, -- ADR[1]
        ExtendedAddress, -- ADR[2]
        Street, -- ADR[3]
        City, -- ADR[4]
        State, -- ADR[5]
        ZIP, -- ADR[6]
        Country STRING   -- ADR[7]
    END RECORD,
    phone DYNAMIC ARRAY OF RECORD
        type STRING,
        number STRING -- TEL
    END RECORD,
    email DYNAMIC ARRAY OF RECORD
        type STRING,
        value STRING -- EMAIL
  END RECORD
```

Usage

This type defines a record structure to hold vCard information. It is used by VCard functions such as format_person().

Example

```
IMPORT FGL VCard
MAIN
  DEFINE p VCard.VCPerson
  LET p.FirstName = "Hans"
  LET p.LastName = "Mustermann"
END MAIN
```

format_person()
Converts a VCPerson record to a vCard string representation vCard.

Syntax

```
format_person(
   person VCPerson )
  RETURNING result STRING
```

1. *person* is a VCPerson record.
2. *result* is the resulting vCard string (version 3.0).

Usage

This function converts a record defined with the VCPerson type, to a string representing a vCard.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE p VCard.VCPerson
  LET p.FirstName = "Hans"
  LET p.LastName = "Mustermann"
  LET p.email[1].VALUE = "hans@nomail.com"
  LET p.phone[1].TYPE = "HOME"
  LET p.phone[1].number = "+49 123 4567 8901"
  LET p.phone[2].TYPE = "WORK"
  LET p.phone[2].number = "+49 123 9876 5431"
  DISPLAY VCard.format_person(p.*)
END MAIN
```

Output:

```
BEGIN:VCARD
VERSION:3.0
N:Hans;Mustermann;;;
FN:Hans Mustermann
TEL;TYPE=HOME:+49 123 4567 8901
TEL;TYPE=WORK:+49 123 9876 5431
EMAIL:hans@nomail.com
END:VCARD
```

scan_address()
Extracts an address from a string representing a vCard.

Syntax

```
scan_address(
   source STRING,
   type STRING )
  RETURNING address VCAddress
```

1. *source* is the vCard string (version 3.0).
2. *type* is the type of address (HOME, WORK, pref).
3. *address* is the address found, returned as VCAddress structure.

Usage

This function parses the vCard string passed as parameter to find address data according to a type, and returns address information in a record defined with the `VCAddress` type.

The function looks for lines starting with the "ADR" keyword.

The second parameter (*type*) defines is the value of the "TYPE" attribute in an "ADR" line. Values can for example be "HOME", "WORK", "pref". If this parameter is `NULL`, the address with TYPE=pref will be returned. If no preferred address exists, the first address will be returned.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE a VCard.VCAddress,
         f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_address(f,"WORK") RETURNING a.*
  DISPLAY a.*
END MAIN
```

scan_email()
Extracts an email from a string representing a vCard.

Syntax

```
scan_email(
   source STRING,
   type STRING )
  RETURNING email STRING
```

1. *source* is the vCard string (version 3.0).
2. *type* is the type of email (HOME, WORK, pref).
3. *email* is the email found.

Usage

This function parses the vCard string passed as parameter to find "EMAIL" data according to a type, and returns the email address as a string.

The function looks for lines starting with the "EMAIL" keyword.

The second parameter (*type*) defines is the value of the "TYPE" attribute in an "EMAIL" line. Values can for example be "HOME", "WORK", "pref". If this parameter is NULL, the email with TYPE=pref will be returned. If no preferred email exists, the first email will be returned.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE m STRING,
         f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_email(f,NULL) RETURNING m
  DISPLAY m
END MAIN
```

scan_name()
Extracts name information from a string representing a vCard.

Syntax

```
scan_name(
   source STRING)
  RETURNING name VCName
```

1. *source* is the vCard string (version 3.0).
2. *name* is the name found, returned as VCName structure.

Usage

This function parses the vCard string passed as parameter to find person name data, and returns name information in a record defined with the VCName type.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE n VCard.VCName,
         f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_name(f) RETURNING n.*
  DISPLAY n.*
END MAIN
```

scan_person()
Extracts person's data from a string representing a vCard.

Syntax

```
scan_person(
   source STRING )
  RETURNING person VCPerson
```

1. *source* is the vCard string (version 3.0).
2. *person* is the resulting VCPerson structure.

Usage

This function parses the vCard string passed as parameter, extracts all information, and returns a record defined with the VCPerson type.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE p VCard.VCPerson,
         f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_person(f) RETURNING p.*
  DISPLAY p.*
END MAIN
```

scan_phone()
Extracts a phone number from a string representing a vCard.

Syntax

```
scan_phone(
   source STRING,
   type STRING )
  RETURNING phone STRING
```

1. *source* is the vCard string (version 3.0).
2. *type* is the type of phone number (HOME, WORK, TEXT, VOICE, FAX, CELL, VIDEO, PAGER, TEXTPHONE, pref).
3. *phone* is the phone number found.

Usage

This function parses the vCard string passed as parameter to find phone data according to a type, and returns the phone number in a string.

The function looks for lines starting with the "TEL" keyword.

The second parameter (*type*) defines is the value of the "TYPE" attribute in an "TELs" line. Values can for example be "HOME", "WORK", "TEXT", "VOICE", "FAX", "CELL", "VIDEO", "PAGER", "TEXTPHONE", "pref". If this parameter is `NULL`, the phone number with TYPE=pref will be returned. If no preferred phone number exists, the first phone number will be returned.

Example

```
IMPORT FGL VCard
MAIN
  DEFINE n STRING,
         f TEXT
  LOCATE f IN FILE arg_val(1)
  CALL VCard.scan_phone(f,NULL) RETURNING n
  DISPLAY n
END MAIN
```

Built-in packages

These topics cover the built-in classes provided by the Genero Business Development Language.

- The BYTE data type as class on page 1687
- The STRING data type as class on page 1689
- The TEXT data type as class on page 1695
- DYNAMIC ARRAY as class on page 1697
- The Java Array type as class on page 1701
- The Application class on page 1703
- The Channel class on page 1707
- The StringBuffer class on page 1738
- The StringTokenizer class on page 1749
- The TypeInfo class on page 1752
- The MessageServer class on page 1754
- The Interface class on page 1755
- The Window class on page 1769
- The Form class on page 1774
- The Dialog class on page 1784
- The ComboBox class on page 1820
- The DragDrop class on page 1827
- The DomDocument class on page 1833
- The DomNode class on page 1839
- The NodeList class on page 1858
- The SaxAttributes class on page 1860
- The SaxDocumentHandler class on page 1865
- The XmlReader class on page 1871
- The XmlWriter class on page 1876

BDL data types package

These topics cover the built-in classes of BDL data types

- The BYTE data type as class on page 1687
- The STRING data type as class on page 1689
- The TEXT data type as class on page 1695
- DYNAMIC ARRAY as class on page 1697
- The Java Array type as class on page 1701

The BYTE data type as class

The BYTE built-in data type provides a set of utility methods to manipulate BYTE data.

BYTE methods can be invoked with the variable, for example:

```
DEFINE b BYTE
CALL t.writeFile("mydata")
```

BYTE data type methods

Table 362: Object methods

Name	Description
`readFile( filename STRING )`	Reads a file into a BYTE locator.
`writeFile( filename STRING )`	Writes the containt of a BYTE to a file.

BYTE.readFile
Reads a file into a BYTE locator.

Syntax

```
readFile( filename STRING )
```

1. *filename* is the path the file to be loaded.

Usage

This method reads a content from the specified file into the BYTE locator.

The bytes are loaded as is, without any conversion.

If the file is not found or if it cannot be read, the error -8087 is raised.

Example

```
MAIN
  DEFINE b BYTE
  LOCATE b IN MEMORY
  CALL b.readFile("mydata")
END MAIN
```

BYTE.writeFile
Writes the containt of a BYTE to a file.

Syntax

```
writeFile( filename STRING )
```

1. *filename* is the file to be written to.

Usage

This method writes the containt of the current BYTE locator to the specified file.

The bytes are written as is, without any conversion.

If the file cannot be written, the error -8087 is raised.

Example

```
MAIN
  DEFINE b BYTE
  LOCATE b IN MEMORY
```

```
  SELECT col_byte INTO b FROM ...
  CALL b.writeFile("mydata")
END MAIN
```

The STRING data type as class

The `STRING` built-in data type provides a set of utility methods to manipulate character strings.

`STRING` methods can be invoked with the variable, for example:

```
DEFINE s STRING
IF s.equalsIgnoreCase("pink") THEN
   ...
END IF
```

STRING data type methods

Table 363: Object methods

Name	Description
`append( string STRING ) RETURNING result STRING`	Concatenates a string.
`equals( string STRING ) RETURNING result BOOLEAN`	Compares a string to the content of the variable.
`equalsIgnoreCase( string STRING ) RETURNING result BOOLEAN`	Makes a case-insensitive string comparison.
`getCharAt( position INTEGER ) RETURNING result CHAR(1)`	Returns the character at the specified position.
`getIndexOf( string STRING, start INTEGER ) RETURNING result INTEGER`	Returns the position of a sub-string.
`getLength( ) RETURNING result INTEGER`	Returns the length of the current string.
`subString( start INTEGER, end INTEGER ) RETURNING result STRING`	Returns a sub-string according to start and end positions.
`toLowerCase( ) RETURNING result STRING`	Returns the string converted to lower case.
`toUpperCase( )`	Returns the string converted to upper case.

Name	Description
`RETURNING result STRING`	
`trim( )` `RETURNING result STRING`	Removes leading a trailing blanks.
`trimLeft( )` `RETURNING result STRING`	Removes leading blanks.
`trimRight( )` `RETURNING result STRING`	Removes trailing blanks.

STRING.append
Concatenates a string.

Syntax

```
append( string STRING )
       RETURNING result STRING
```

1. *string* is the string to be concatenated.

Usage

This method concatenates a string to the current `STRING` variable and returns the resulting string.

The original `STRING` variable is not modified.

Appending a `NULL` will have no effect: the original string is returned.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.append("... more text")
END MAIN
```

STRING.equals
Compares a string to the content of the variable.

Syntax

```
equals( string STRING )
       RETURNING result BOOLEAN
```

1. *string* is the string to compare with.

Usage

This method compares a string to the current `STRING` variable and returns `TRUE` if both strings match.

If the original `STRING` variable or the string passed as parameter is `NULL`, the result with be `FALSE`.

Example

```
MAIN
  DEFINE s STRING
  LET s = "white"
  IF s.equals("white") THEN
     DISPLAY "Matches"
  END IF
END MAIN
```

STRING.equalsIgnoreCase
Makes a case-insensitive string comparison.

Syntax

```
equalsIgnoreCase( string STRING )
       RETURNING result BOOLEAN
```

1. *string* is the string to compare with.

Usage

This method compares a string to the current `STRING` variable by ignoring the character case, and returns `TRUE` if both strings match.

If the original `STRING` variable or the string passed as parameter is `NULL`, the result with be `FALSE`.

Example

```
MAIN
  DEFINE s STRING
  LET s = "white"
  IF s.equalsIgnoreCase("WHITE") THEN
     DISPLAY "Matches"
  END IF
END MAIN
```

STRING.getCharAt
Returns the character at the specified position.

Syntax

```
getCharAt( position INTEGER )
       RETURNING result CHAR(1)
```

1. *position* is the position of the character int the string.

Usage

This method extracts the character at the specified position from the `STRING` variable.

If the `STRING` variable is `NULL`, or if the *position* is out of the bounds of the string, the result will be `NULL`.

> **Important:** When using byte length semantics, the position is expressed in bytes, and when using char length semantincs, position is specified in characters. In byte length semantics, the method returns `NULL` if the position does not match a valid character-byte index in the current string.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.getCharAt(4)
END MAIN
```

STRING.getIndexOf
Returns the position of a sub-string.

Syntax

```
getIndexOf( string STRING, start INTEGER )
       RETURNING result INTEGER
```

1. *string* is the sub-string to be searched.
2. *start* is the starting position for the search.

Usage

This method scans a `STRING` variable to find the sub-string passed as parameter, and returns the position of the sub-string.

The method starts to search the sub-string at the starting position specified as second parameter.

The method returns zero if:

- The substring was not found.
- The variable is `NULL`.
- The sub-string is `NULL`.
- The start position is out of bounds.

Important: When using byte length semantics, the position is expressed in bytes, and when using char length semantincs, it is specified in characters.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.getIndexOf("text",1)
END MAIN
```

STRING.getLength
Returns the length of the current string.

Syntax

```
getLength( )
       RETURNING result INTEGER
```

Usage

This method counts the number of bytes or characters in a `STRING` variable.

Note: Unlike the `LENGTH( )` function, the getLength() method counts the trailing blanks.

If the STRING variable is NULL, the method returns zero.

> **Important:** When using byte length semantics, the length is expressed in bytes, and when using char length semantincs, it is expressed in characters.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.getLength()
END MAIN
```

STRING.subString
Returns a sub-string according to start and end positions.

Syntax

```
subString( start INTEGER, end INTEGER )
       RETURNING result STRING
```

1. *start* is the starting position of the sub-string.
2. *end* is the ending position of the sub-string.

Usage

This method returns a sub-string of the current STRING variable according to a start and end position in the original string.

If the STRING variable is NULL, or when the positions are out of bounds, the method returns NULL.

> **Important:** When using byte length semantics, the positions are expressed in bytes, and when using char length semantincs, positions are expressed in characters.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.subString(6,9)
END MAIN
```

STRING.toLowerCase
Returns the string converted to lower case.

Syntax

```
toLowerCase( )
       RETURNING result STRING
```

Usage

This method converts the current STRING variable to lower case and returns the resulting string.

If the original STRING variable is NULL, the result is NULL.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.toLowerCase()
END MAIN
```

STRING.toUpperCase
Returns the string converted to upper case.

Syntax

```
toUpperCase( )
       RETURNING result STRING
```

Usage

This method converts the current `STRING` variable to upper case and returns the resulting string.

If the original `STRING` variable is `NULL`, the result is `NULL`.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text"
  DISPLAY s.toUpperCase()
END MAIN
```

STRING.trim
Removes leading a trailing blanks.

Syntax

```
trim( )
       RETURNING result STRING
```

Usage

This method deletes the white space characters before the first character and after the last character of the current `STRING` variable and returns the resulting string.

If the original `STRING` variable is `NULL`, the result will be `NULL`.

Example

```
MAIN
  DEFINE s STRING
  LET s = "   Some text   "
  DISPLAY s.trim()
END MAIN
```

STRING.trimLeft
Removes leading blanks.

Syntax

```
trimLeft( )
       RETURNING result STRING
```

Usage

This method deletes the white space characters before the first character of the current `STRING` variable and returns the resulting string.

If the original `STRING` variable is `NULL`, the result will be `NULL`.

Example

```
MAIN
  DEFINE s STRING
  LET s = "    Some text"
  DISPLAY s.trimLeft()
END MAIN
```

STRING.trimRight
Removes trailing blanks.

Syntax

```
trimRight( )
       RETURNING result STRING
```

Usage

This method deletes the white space characters after the last character of the current `STRING` variable and returns the resulting string.

If the original `STRING` variable is `NULL`, the result will be `NULL`.

Example

```
MAIN
  DEFINE s STRING
  LET s = "Some text     "
  DISPLAY s.trimRight()
END MAIN
```

The TEXT data type as class
The `TEXT` built-in data type provides a set of utility methods to manipulate TEXT data.

`TEXT` methods can be invoked with the variable, for example:

```
DEFINE t TEXT
CALL t.writeFile("mydata")
```

TEXT data type methods

Table 364: Object methods

Name	Description
`getLength( )` `RETURNING result INTEGER`	Returns the length of a TEXT content.
`readFile( filename STRING )`	Reads a file into a TEXT locator.
`writeFile( filename STRING )`	Writes the containt of a TEXT to a file.

TEXT.getLength
Returns the length of a TEXT content.

Syntax

```
getLength( )
   RETURNING result INTEGER
```

Usage

This method returns the number of bytes of the TEXT data.

> **Important:** This method returns always a number of bytes, even when using character length semantics.

Example

```
MAIN
  DEFINE t TEXT
  LOCATE t IN MEMORY
  DISPLAY t.getLength()
END MAIN
```

TEXT.readFile
Reads a file into a TEXT locator.

Syntax

```
readFile( filename STRING )
```

1. *filename* is the path the file to be loaded.

Usage

This method reads a content from the specified file into the TEXT locator.

If the file is not found or if it cannot be read, the error -8087 is raised.

> **Important:** The character set used in the file must match the current application locale.

Example

```
MAIN
  DEFINE t TEXT
  LOCATE t IN MEMORY
  CALL t.readFile("mydata")
END MAIN
```

TEXT.writeFile
Writes the containt of a TEXT to a file.

Syntax

```
writeFile( filename STRING )
```

1. *filename* is the file to be written to.

Usage

This method writes the containt of the current TEXT locator to the specified file.

If the file cannot be written, the error -8087 is raised.

Important: The character set used in the file must match the current application locale.

Example

```
MAIN
  DEFINE t TEXT
  LOCATE t IN MEMORY
  SELECT col_text INTO t FROM ...
  CALL t.writeFile("mydata")
END MAIN
```

DYNAMIC ARRAY as class

The DYNAMIC ARRAY (or static ARRAY) type provides a set of utility methods to manipulate the array elements.

DYNAMIC ARRAY methods can be invoked with the variable, for example:

```
DEFINE a DYNAMIC ARRAY OF STRING
CALL a.appendElement()
DISPLAY a.getLength()
```

DYNAMIC ARRAY methods

Table 365: Object methods

Name	Description
`appendElement( )`	Adds a new element to the end of the array.
`clear( )`	Removes all elements of the array.

Name	Description
`RETURNING result INTEGER`	
`deleteElement( index INTEGER )`	Removes an element from the array according to its index.
`getLength( )` `RETURNING result INTEGER`	Returns the length of the array.
`insertElement( index INTEGER )`	Inserts a new element at the given index.
`sort( key STRING, reverse BOOLEAN )`	Sorts the rows in the array.

DYNAMIC ARRAY.appendElement
Adds a new element to the end of the array.

Syntax

```
appendElement( )
```

Usage

This method creates a new element at the end of the array.

The element is initialized to `NULL`.

Example

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  ... (array has already 10 elements)
  CALL a.appendElement()
  LET a[a.getLength()] = a.getLength()
  DISPLAY a.getLength() -- shows 11
  DISPLAY a[10]  -- shows 10
  DISPLAY a[11]  -- shows 11
END MAIN
```

Since element allocation occurs automatically for dynamic arrays, you can omit the call the `appendElement()` method and assign directly the new element:

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[100] = 87234 -- Array gets a length of 100 automatically
  LET a[101] = 98562 -- New element at position 101
END MAIN
```

DYNAMIC ARRAY.clear
Removes all elements of the array.

Syntax

```
clear( )
```

```
    RETURNING result INTEGER
```

Usage

This method clears the array, by removing all its elements.

Use the `clear()` method just before filling the array with a new set of elements, if the array is potentially not empty.

Example

```
FUNCTION fill_array(arr)
    DEFINE arr DYNAMIC ARRAY OF STRING
    DEFINE i INTEGER
    CALL arr.clear()
    FOR i=1 TO 10
       LET arr[i] = "Item #"||i
    END FOR
END FUNCTION
```

DYNAMIC ARRAY.deleteElement
Removes an element from the array according to its index.

Syntax

```
deleteElement( index INTEGER )
```

Usage

This method removes the array element at the specified index.

No error is raised if the index is out of bounds.

Example

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[10] = 9
  CALL a.deleteElement(5)
  DISPLAY a.getLength() -- shows 9
  DISPLAY a[9]  -- shows 9
END MAIN
```

DYNAMIC ARRAY.getLength
Returns the length of the array.

Syntax

```
getLength( )
   RETURNING result INTEGER
```

Usage

This method returns the number of elements in the array.

Example

```
DEFINE arr DYNAMIC ARRAY OF STRING,
       i INTEGER
FOR i=1 TO arr.getLength()
    DISPLAY arr[i]
END FOR
```

DYNAMIC ARRAY.insertElement
Inserts a new element at the given index.

Syntax

```
insertElement( index INTEGER )
```

Usage

This method inserts a new element in the array, before the specified index.

No error is raised if the index is out of bounds.

Example

```
MAIN
  DEFINE a DYNAMIC ARRAY OF INTEGER
  LET a[10] = 11
  CALL a.insertElement(10)
  LET a[10] = 10
  DISPLAY a.getLength() -- shows 11
  DISPLAY a[10]  -- shows 10
  DISPLAY a[11]  -- shows 11
END MAIN
```

DYNAMIC ARRAY.sort
Sorts the rows in the array.

Syntax

```
sort( key STRING, reverse BOOLEAN )
```

1. *key* is the name of a member of a structured array (`DYNAMIC ARRAY OF RECORD`), or `NULL` if the array is not structured.
2. *reverse* is `FALSE` for ascending order, `TRUE` for descending order.

Usage

This method sorts the array according to the name of the member passed as first parameter, for arrays defined with a structured type (`DYNAMIC ARRAY OF RECORD`). If the array is defined with a simple type, the first argument can be `NULL`.

The second parameter defines the sort order as ascending (FALSE) or descending (TRUE).

When doing subsequent calls to the sort() method using different record members of the array, the rows will be ordered by all of the record members specified for the cumulative sorts, with the most recent call defining the main sort field.

Another way to think of this is in terms of the `ORDER BY` clause of a SQL statement: If your dynamic array contained the variables A, B and C, and you included the following calls to the `sort()` method:

```
CALL a.sort("C",false)
CALL a.sort("B",false)
CALL a.sort("A",false)
```

This would be equivalent to writing an `ORDER BY` clause that states:

```
ORDER BY A, B, C
```

Note: Character string data is sorted according to the current application locale.

Example

```
MAIN
  DEFINE a DYNAMIC ARRAY OF RECORD
               key INTEGER,
               name VARCHAR(30)
         END RECORD
  LET a[1].key = 776236    LET a[1].name = "aaaaa"
  LET a[2].key = 273434    LET a[2].name = "cccccccc"
  LET a[3].key = 934092    LET a[3].name = "bbbbb"
  CALL a.sort("name",FALSE)
    -- Array is sorted by name (asc order)
  CALL a.sort("key",TRUE)
    -- Array is sorted by key (desc order), then by name (asc
 order)
    -- within each key;
    -- The current sort becomes the main sort field, the initial
 sort
    -- becomes the secondary sort field
END MAIN
```

The Java Array type as class

The Java Array type provides a set of utility methods array elements.

Java array methods can be invoked with a type reference or the array variable, for example:

```
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
    LET names[1] = "aaaaaaa"
    DISPLAY names.getLength()
END MAIN
```

Java Array type methods

Table 366: Class methods

Name	Description
java-array-type`.create(` *size* `INTEGER )` `RETURNING` *object java-array-type*	Creates a new Java array of the given type.

Table 367: Object methods

Name	Description
`getLength( )` `RETURNING result INTEGER`	Returns the length of the Java array.

java-array-type.create
Creates a new Java array of the given type.

Syntax

```
java-array-type.create( size INTEGER )
   RETURNING object java-array-type
```

1. *size* defines the actual number of elements of the array.

Usage

This class method creates a new instance of the Java array specified by the type used, with the size provided as parameter.

The type must be declared as a user defined type define with the `ARRAY [ ] OF` notation reserved for Java arrays.

Example

```
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
    LET names[1] = "aaaaaaa"
    DISPLAY names[1]
END MAIN
```

java-array.getLength
Returns the length of the Java array.

Syntax

```
getLength( )
   RETURNING result INTEGER
```

Usage

This method returns the number of elements in the Java array.

Example

```
IMPORT JAVA java.lang.String
MAIN
    TYPE string_array_type ARRAY[] OF java.lang.String
    DEFINE names string_array_type
    LET names = string_array_type.create(100)
    LET names[1] = "aaaaaaa"
    DISPLAY names.getLength()
```

```
END MAIN
```

The base package

These topics cover the built-in classes for the base class

- The Application class on page 1703
- The Channel class on page 1707
- The SqlHandle class on page 1725
- The StringBuffer class on page 1738
- The StringTokenizer class on page 1749
- The TypeInfo class on page 1752
- The MessageServer class on page 1754

The Application class

The `base.Application` class provides a set of utility functions related to the program environment.

Command line arguments, execution directory and FGLPROFILE resource entries are some of the elements you can query with this class.

This class is built-in and can be used directly in the source code.

This class does not have to be instantiated. It provides class methods for the current program.

base.Application methods

Table 368: Class methods

Name	Description
`base.Application.getArgument( index INTEGER ) RETURNING result STRING`	Returns the command line argument by position.
`base.Application.getArgumentCount() RETURNING result INTEGER`	Returns the total number of command line arguments.
`base.Application.getProgramDir() RETURNING result STRING`	Returns the directory path of the current program.
`base.Application.getProgramName() RETURNING result STRING`	Returns the name of the current program.
`base.Application.getFglDir() RETURNING result STRING`	Returns the path to the FGLDIR installation directory.
`base.Application.getResourceEntry( entry STRING ) RETURNING result STRING`	Returns the value of an FGLPROFILE entry.
`base.Application.getStackTrace()`	Returns the function call stack trace.

Name	Description
`RETURNING result STRING`	
`base.Application.isMobile()` `RETURNING result BOOLEAN`	Indicates if the application runs on a mobile device.

base.Application.getArgument
Returns the command line argument by position.

Syntax

```
base.Application.getArgument(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the index of the program argument.

Usage

The index is the program argument position. The first program argument is identified by the position 1. Argument number zero is the program name.

Returns `NULL` if there is no argument provided at the position.

Example

```
MAIN
  DEFINE i INTEGER
  FOR i=1 TO base.Application.getArgumentCount()
    DISPLAY base.Application.getArgument(i)
  END FOR
END MAIN
```

base.Application.getArgumentCount
Returns the total number of command line arguments.

Syntax

```
base.Application.getArgumentCount()
  RETURNING result INTEGER
```

Usage

Returns the total number of command line arguments, can be used to scan the argument values with `base.Application.getArgument()`.

base.Application.getFglDir
Returns the path to the FGLDIR installation directory.

Syntax

```
base.Application.getFglDir()
  RETURNING result STRING
```

Usage

The `getFglDir()` method returns the installation directory path defined by the FGLDIR environment variable. The directory path is system-dependent.

base.Application.getProgramDir
Returns the directory path of the current program.

Syntax

```
base.Application.getProgramDir()
  RETURNING result STRING
```

Usage

This method returns the directory path where the program file (42r) is located.

The directory path is system-dependent.

base.Application.getProgramName
Returns the name of the current program.

Syntax

```
base.Application.getProgramName()
  RETURNING result STRING
```

Usage

This method returns the name of the current program. This is the name of the 42m or 42r module passed to fglrun, without the file extension.

base.Application.getResourceEntry
Returns the value of an FGLPROFILE entry.

Syntax

```
base.Application.getResourceEntry(
   entry STRING )
  RETURNING result STRING
```

1. *entry* is the name of an FGLPROFILE entry.

Usage

This method returns the fglprofile value of the FGLPROFILE resource entry passed as parameter.

Example

```
MAIN
  DISPLAY
 base.Application.getResourceEntry("mycompany.params.logmode")
END MAIN
```

base.Application.getStackTrace
Returns the function call stack trace.

Syntax

```
base.Application.getStackTrace()
  RETURNING result STRING
```

Usage

Discover which functions have been called when a program raises an error. Use the `getStackTrace()` method to print the stack trace to a log file. This method returns a string containing a formatted list of the current function stack.

You typically use this function in a `WHENEVER ERROR CALL` handler.

```
MAIN
  WHENEVER ERROR CALL my_handler
  ...
END MAIN
...
FUNCTION my_handler()
  DISPLAY base.Application.getStackTrace()
END FUNCTION
```

Example of stack trace output:

```
#0 my_handler() at debug.4gl:173
#1 save_customer_data() at customer.4gl:1534
#2 edit_customer() at customer.4gl:542
#3 main at main.4gl:23
```

base.Application.isMobile
Indicates if the application runs on a mobile device.

Syntax

```
base.Application.isMobile()
  RETURNING result BOOLEAN
```

1. *result* is `TRUE` if the program runs on a mobile device.

Usage

This class method can be called to check if the program code is running on a smartphone or tablet device. The method will return `TRUE` if the program executes in standalone mode (i.e. the runtime system is on the mobile device).

Example

```
MAIN
  IF base.Application.isMobile() THEN
     MESSAGE "We are on a mobile device."
  END IF
END MAIN
```

The Channel class

The `base.Channel` class is a built-in class providing basic input/output functions.

base.Channel methods

Table 369: Class methods

Name	Description
`base.Channel.create()` `  RETURNING result base.Channel`	Create a new channel object.

Table 370: Object methods

Name	Description
`dataAvailable()` `  RETURNING result BOOLEAN`	Tests if some data can be read from the channel.
`close()`	Closes the channel.
`isEof()` `  RETURNING result BOOLEAN`	Detect the end of a file.

`openFile(`
`  path STRING,`
`  mode STRING )`
	Opening a file channel.

`openPipe(`
`  cmd STRING,`
`  mode STRING )`
	Opening a pipe channel to a sub-process.

`openClientSocket(`
`  host STRING,`
`  port INTEGER,`
`  mode STRING,`
`  timeout INTEGER )`
	Open a TCP client socket channel.

`openServerSocket(`
`  interface STRING, port INTEGER,`
`  mode STRING )`
	Open a TCP server socket channel.

`read(`
`  [ variable-list ] )`
`  RETURNING result INTEGER`
	Reads a list of data delimited by a separator from the channel.

`readLine()`	Read a complete line from the channel.

Name	Description
`RETURNING result STRING`	
`readOctets( bytes INTEGER) RETURNING result STRING`	Read a given number of bytes and return it as a character string.
`setDelimiter( delim STRING )`	Define the value delimiter for a channel.
`write( [ variable-list ] )`	Writes a list of data delimited by a separator to the channel.
`writeLine( line STRING )`	Write a complete line to the channel.
`writeNoNL( string STRING )`	Writes a string to the channel (without newline character).

base.Channel.create
Create a new channel object.

Syntax

```
base.Channel.create()
  RETURNING result base.Channel
```

Usage

Use the `base.Channel.create()` class method to create a channel object.

The new created object must be assigned to a program variable defined with the `base.Channel` type.

Example

```
DEFINE ch base.Channel
LET ch = base.Channel.create()
```

base.Channel.close
Closes the channel.

Syntax

```
close()
```

Usage

Call the `close()` method when you are done with the channel. The channel can be re-opened after it has been closed.

Note: A channel is automatically closed, when the channel object is destroyed.

Example

```
CALL ch.close()
```

base.Channel.dataAvailable
Tests if some data can be read from the channel.

Syntax

```
dataAvailable()
  RETURNING result BOOLEAN
```

Usage

The `dataAvailable()` method returns `TRUE` if some data can be read from the channel.

This method is only to be used in some rare cases. Use `dataAvailable()` if the protocol allows asynchronous messages from the peer. An example is an asynchronous error message from the peer, to stop sending more data.

`dataAvailable()` checks if at least one byte is available on the stream. A subsequent read will block, if the read operation can not be completed. This should not happen: the methods `read()` and `readLine()` and their counterparts `write()` and `writeLine()` read and write complete lines (a line is a sequence of characters terminated by the line separator).

The method opens up the possibility to read data asynchronously. One possible use for this method is to stop a data transfer to a remote site after receiving an error message from the remote site.

Example

The local site sends a huge amount of data to the remote site using `base.Channel.writeLine()`. An error may occur during the processing of data by the remote side. The remote site writes an error message causing the local site to stop the data transmission.

On the local site, the file is `parent.4gl`.

```
-- this file: parent.4gl
MAIN
  DEFINE i INT
  DEFINE c base.Channel

  LET c = base.Channel.create()
  CALL c.openPipe("fglrun child", "u")
  WHILE TRUE
    IF c.dataAvailable() THEN
      DISPLAY "message from child: ", c.readLine()
      EXIT WHILE
    END IF
    CALL c.writeLine("line " || i)
  END WHILE
END MAIN
```

On the remote site, the file is `child.4gl`.

```
-- this file: child.4gl
MAIN
```

```
  DEFINE c base.Channel
  DEFINE s STRING
  DEFINE n INT

  LET n = 0
  LET c = base.Channel.create()
  CALL c.openFile("", "u")
  WHILE NOT c.isEof()
    LET s = c.readLine()
    LET n = n + 1
    IF n == 3 THEN
      CALL c.writeLine("error: something happens")
      CALL readRemainingData(c)
      EXIT WHILE
    END IF
  END WHILE
END MAIN

FUNCTION readRemainingData(c)
  DEFINE c base.Channel
  DEFINE s STRING
  WHILE NOT c.isEof()
    LET s = c.readLine()
  END WHILE
END FUNCTION
```

base.Channel.isEof
Detect the end of a file.

Syntax

```
isEof()
  RETURNING result BOOLEAN
```

Usage

Use the `isEof()` method to detect the end of a file while reading from a channel.

The end of file is only detected after the last read. In other words, you first read, then check for the end of file and process if not end of file.

Example

```
DEFINE s STRING
WHILE TRUE
  LET s = ch.readLine()
  IF ch.isEof() THEN
     EXIT WHILE
  END IF
  DISPLAY s
END WHILE
```

base.Channel.openClientSocket
Open a TCP client socket channel.

Syntax

```
openClientSocket(
   host STRING,
```

```
   port INTEGER,
   mode STRING,
   timeout INTEGER )
```

1. *host* is the name of the host machine you want to connect to.
2. *port* is the port number of the service.
3. *mode* is the open mode. Can be `"r"`, `"w"` or `"u"` (combined with `"b"` if needed).
4. *timeout* is the timeout in seconds. -1 indicates no timeout (wait forever)

Usage

Use the `openClientSocket()` method to establish a TCP connection to a server.

Pay attention to character set used by the network protocol you want to use by opening a channel with this method: The protocol must be based on ASCII, or must use the same character set as the application.

The *host* parameter defines the host name of the server.

The *port* parameter defines the TCP port to connect to.

The opening *mode* can be one of the following:

- `r`: For read mode: only to read from the socket
- `w`: For write mode: only to write to the socket
- `u`: For read and write mode: To read and write from/to the socket

Any of these modes can be followed by `b`, to use binary mode and avoid CR/LF translation on Windows platforms.

> **Note:** The binary mode is only required in specific cases, and will only take effect when writing data.

If the opening *mode* is not one of the above letters, the method will raise error -8085.

When the *timeout* parameter is -1, the connection waits forever.

The method raises error -8084 if the channel cannot be opened.

Example

```
CALL ch.openClientSocket( "localhost", 80, "u", 5 )
```

base.Channel.openFile
Opening a file channel.

Syntax

```
openFile(
   path STRING,
   mode STRING )
```

1. *path* is the path to the file to open, can be `NULL` for stdin/stdout.
2. *mode* is the open mode. Can be `"r"`, `"w"`, `"a"` or `"u"` (combined with `"b"` if needed).

Usage

The `openFile()` method can be used to open a file for reading, writing, or both.

When passing `NULL` as file name, the channel can be used to read and/or write to stdout or stdin, according to *mode*.

The opening *mode* can be one of the following:

- `r`: For read mode: reads from a file (standard input if path is `NULL`).
- `w`: For write mode: starts with an empty file (standard output if the path is `NULL`).
- `a`: For append mode: writes at the end of a file (standard output if the path is `NULL`).
- `u`: For read from standard input and write to standard output (path must be `NULL`).

Any of these modes can be followed by `b`, to use binary mode and avoid CR/LF translation on Windows platforms.

> **Note:** The binary mode is only required in specific cases, and will only take effect when writing data.

If the opening *mode* is not one of the above letters, the method will raise error -8085

When you use the `w` or `a` modes, the file is created if it does not exist.

The method raises error -6340 sif the file cannot be opened.

Example

```
CALL ch.openFile( "file.txt", "w" )
```

base.Channel.openPipe
Opening a pipe channel to a sub-process.

Syntax

```
openPipe(
   cmd STRING,
   mode STRING )
```

1. *cmd* is the system command to be executed.
2. *mode* is the open mode. Can be `"r"`, `"w"`, `"a"` or `"u"` (combined with `"b"` if needed).

Usage

With the `openPipe()` method, you can read from the standard output of a subprocess, write to the standard input, or both.

> **Important:** This feature is not supported on mobile platforms.

The opening *mode* can be one of the following:

- `r`: For read only from standard output of the command.
- `w`: For write only to standard input of the command.
- `a`: For write only to standard input of the command.
- `u`: For read from standard output and write to standard input of the command.

Any of these modes can be followed by `b`, to use binary mode and avoid CR/LF translation on Windows platforms.

> **Note:** The binary mode is only required in specific cases, and will only take effect when writing data.

If the opening *mode* is not one of the above letters, the method will raise error -8085.

Example

```
CALL ch.openPipe( "ls", "r" )
```

base.Channel.openServerSocket
Open a TCP server socket channel.

Syntax

```
openServerSocket(
   interface STRING, port INTEGER,
   mode STRING )
```

1. *interface* is the name of the network interface to be used.
2. *port* is the port number of the service.
3. *mode* is the open mode. Only `"u"` is allowed (combined with `"b"` if needed).

Usage

The `openServerSocket()` method initializes the channel object to listen to a given TCP interface and port.

The server socket accepts multiple client connects: After calling the `openServerSocket()` method, a call to `readLine()` waits until the first client connects and returns after reading a complete line. Only one client connection can be serviced at time: it's not possible to select a specific client connection. A client connection must be closed by writing the EOF character to the channel. The EOF character is ASCII 26. Do not call `base.Channel.close()` to close a client/server connection: This would close the sever socket and reject any pending client connection. The next call to `readLine()` after writing EOF will wait until the next client connects or select the next pending client.

Pay attention to character set used by the network protocol you want to use by opening a channel with this method: The protocol must be based on ASCII, or must use the same character set as the application.

The *interface* parameter defines the network interface to be used, in case if the server uses different network adapters. Use `NULL` to listen to all network interfaces, or when the server has only one network interface.

The *port* parameter defines the TCP port to listen to.

The opening *mode* must be `"u"`, to read and write from/to the socket. The method will raise error -8085 if the mode is different from `"u"`.

The `"u"` mode can be combined with the `"b"` binary mode, to avoid CR/LF translation on Windows platforms.

> **Note:** The binary mode is only required in specific cases, and will only take effect when writing data.

The method raises error -8084 if the socket cannot be opened.

Example

```
MAIN
    DEFINE io base.Channel
    DEFINE s STRING
    LET io = base.Channel.create()
    CALL io.openServerSocket("127.0.0.1", 4711, "u")
    WHILE TRUE
        LET s = io.readLine()
        CALL io.writeLine(s)
        -- next line closes the current connection
        CALL io.writeLine(ASCII 26) -- EOF
    END WHILE
END MAIN
```

base.Channel.read
Reads a list of data delimited by a separator from the channel.

Syntax

```
read(
   [ variable-list ] )
  RETURNING result INTEGER
```

1. *variable-list* is a list of program variables separated by a comma, or record.*

Usage

After opening the channel object, use the `read()` method to read a record of data from the channel.

The `read()` method uses the field delimiter defined by `setDelimiter()`.

The `read()` method takes a modifiable list of variables as parameter, by using the `[ ]` square brace notation.

A call to `read()` is blocking until the read operation is complete.

If the `read()` method returns less data than expected, then the remaining variables will be initialized to NULL. If the `read()` method returns more data than expected, the data is silently ignored.

Any target variable must have a primitive type (`BOOLEAN`, `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `SMALLFLOAT`, `FLOAT`, `DECIMAL`, `DATE`, `DATETIME`, `INTERVAL`, `BYTE`, `TEXT`, `CHAR`, `VARCHAR`, `STRING`) or be a `RECORD` that contains only primitive members.

If data could be read, the `read()` method returns `TRUE`. Otherwise it returns `FALSE`, indicating the end of the file or stream.

Example

```
WHILE ch.read([cust_rec.*])
  ...
END WHILE
```

base.Channel.readLine
Read a complete line from the channel.

Syntax

```
readLine()
  RETURNING result STRING
```

Usage

After opening the channel object, use the `readLine()` method to read a complete line from the channel.

The `readLine()` method returns an empty string if the line is empty.

A call to `readLine()` is blocking until the read operation is complete.

The `readLine()` function returns `NULL` if end of file is reached. To distinguish empty lines from `NULL`, you must use the `STRING` data type. If you use a `CHAR` or `VARCHAR`, you will get `NULL` for empty lines. To detect the end of file, use the `isEof()` method.

Example

```
WHILE TRUE
```

```
  LET s = ch.readLine()
  IF ch.isEof() THEN EXIT WHILE END IF
  ...
END WHILE
```

base.Channel.readOctets
Read a given number of bytes and return it as a character string.

Syntax

```
readOctets(
   bytes INTEGER)
  RETURNING result STRING
```

1. *bytes* is the number of bytes to read, not the number of characters.

Usage

After opening the channel object, call the `readOctets()` method to read a given number of bytes from the channel. The bytes are returned as a character string. The bytes read must match the current encoding.

The `readOctets()` function returns `NULL` if end of file is reached. To distinguish empty lines from `NULL`, you must use the `STRING` data type. If you use a `CHAR` or `VARCHAR`, you will get `NULL` for empty lines. To properly detect end of file, use the `isEof()` method.

Before reading the actual bytes in a `readOctets()` call, you typically get the number of bytes to read from the sender, as shown in the example.

A valid use case of the method is the HTTP protocol. Reading HTML content with `readLine()` is not possible: The body consists of multiple lines, and the last line might not be terminated by a line-terminator, and the stream gets not EOF:

```
HTTP/1.0 200 OK
Date:  Wed, 16 Apr 2014 18:50:51 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>My title</h1>
  :
</body>
</html>
```

Example

```
WHILE TRUE
  ...
  -- Get the number of bytes to read.
  LET len = ch.readLine()
  -- Read the bytes as character string.
  LET s = ch.readOctets(len)
  IF ch.isEof() THEN EXIT WHILE END IF
  ...
END WHILE
```

base.Channel.setDelimiter
Define the value delimiter for a channel.

Syntax

```
setDelimiter(
   delim STRING )
```

1. *delim* is the value delimiter to be used.

Usage

After creating the channel object, define the field value delimiter with the `setDelimiter()` method.

```
CALL ch.setDelimiter("^")
```

The default delimiter is defined by the DBDELIMITER environment variable, or a pipe (`|`) if DBDELIMITER is not defined.

Specify `CSV` as the delimiter to read/write in Comma Separated Value format.

```
CALL ch.setDelimiter("CSV")
```

> **Important:** Setting a `NULL` delimiter is allowed for backward compatibility, but must be avoid: This was a workaround to read/write complete lines. If the delimiter is set to `NULL`, the `read()` and `write()` methods do not use the backslash (`\`) escape character. As a result, data with special characters like backslash, delimiter or line-feed will be written as is, and reading data will ignore escaped characters in the source stream. If you need to read or write non-formatted data, you should use the `readLine()`/`writeLine()` methods instead. These methods do not use a delimiter, nor do they use the backslash escape character.

base.Channel.write
Writes a list of data delimited by a separator to the channel.

Syntax

```
write(
   [ variable-list ] )
```

1. *variable-list* is a list of program variables separated by a comma, or record.*

Usage

After opening a channel, use the `write()` method to write a record of data to the channel.

The `write()` method uses the field delimiter defined by `setDelimiter()`.

The `write()` method takes a modifiable list of variables as the parameter, using the `[ ]` square brace notation.

The method raises error -6345 if the channel fails to write data.

Example

```
CALL ch.write([cust_rec.*])
```

base.Channel.writeLine
Write a complete line to the channel.

Syntax

```
writeLine(
   line STRING )
```

1. *line* is the string expression to be written to the channel.

Usage

After opening a channel, use the `writeLine()` method to write a line of text to the channel.

The `writeLine()` method does not use the field delimiter, it write the text data to the stream, with an ending newline character.

To write a string with no ending newline character, use the `writeNoNL()` method.

The method raises error -6345 if the channel fails to write data.

Example

```
CALL ch.writeLine("Customer number: "|| custno)
```

base.Channel.writeNoNL
Writes a string to the channel (without newline character).

Syntax

```
writeNoNL(
     string STRING )
```

1. *string* is the character string to be written to the channel.

Usage

After opening a channel, use the `writeNoNL()` method to write a string to the channel, without a trailing newline character.

> **Important:** Do not confuse the `writeNoNL()` method with the `write()` method. The first is provided to write raw character strings to the stream, while the second is designed to write records with formatted data and field delimiters. Note also that the Channel class provides the `writeLine()` method to write a string with a ending newline character.

The method raises error -6345 if the channel fails to write data.

Example

```
CALL ch.writeNoNL("Some text ...")
```

Usage

The `base.Channel` class is a built-in class providing basic input/output functionality for:

- text file reading/writing
- subprocess communication (through pipes)
- basic network communication (through TCP sockets)

> **Important:** No character set conversion is done when reading or writing data with channel objects. The character set used in the data file must correspond to the locale of the runtime system, for both input and output.

Steps to use a channel object:

- Define a variable with the `base.Channel` type.
- Create a channel object with `base.Channel.create()` and assign it to the variable.
- Open the channel for a file, piped process or socket (as a client).
- Read or write data in formatted mode or in line mode.
- Close the channel.

Channel methods may raise exceptions. Exceptions can be trapped with the `WHENEVER ERROR` or `TRY/CATCH` instructions.

When reading or writing strings, the escape character is the backslash (\).

The are three modes to read and write data with Channels:

1. Reading/writing formatted data as a set of fields in a line (i.e. records), with the `read()` and `write()` methods, needing a value separator defined by `setDelimiter()`. This mode follows the same formatting rules as the `LOAD/UNLOAD` instructions, and can also be used to read/write CSV (Comma Separated Value) formatted data.
2. Reading/writing complete lines with the `readLine()` and `writeLine()` methods. This mode is typically used to read/write simple data files.
3. Handling raw character string data by reading/writing pieces of strings, with the readOctets() and writeNoNL() methods.

Read and write formatted data

When the channel is open, use the `read()`/`write()` methods to read and write data records where field values are separated by a delimiter defined by `setDelimiter()`.

> **Note:** The `LOAD`/ `UNLOAD` SQL instructions follow the same formatting rules as the `read()`/`write()` channel methods.

The input or output stream is text data where each line contains the string representation of a record. Field values are separated by the delimiter character defined.

For example, a formatted text file can look like this, when using a default pipe (`|`) delimiter:

```
8712|David|Cosneski|24-12-1978|
3422|Frank|Zapinetti|13-04-1968|
323|Mark|Kelson|03-10-1988|
```

In the serialized data, empty fields (`||`) have a length of zero and are considered as `NULL`.

To read the above formatted data, the code could be:

```
MAIN
  DEFINE ch base.Channel
  DEFINE custinfo RECORD
            cust_num INTEGER,
            cust_fname VARCHAR(40),
            cust_lname VARCHAR(40),
            cust_bdate DATE
          END RECORD
  LET ch = base.Channel.create()
  CALL ch.setDelimiter("|")
  CALL ch.openFile("custinfo.txt","r")
  WHILE ch.read([custinfo.*])
    DISPLAY custinfo.*
  END WHILE
```

```
  CALL ch.close()
END MAIN
```

The backslash \ is the escape character: When writing data with `write()`, special characters like the backslash, line-feed or the delimiter character will be escaped. When reading data with `read()`, any escaped *char* character will be converted to *char*.

The next code writes a single field value where the character string contains a backslash, the pipe delimiter and a line-feed character. The backslash is also the escape character for string literals, therefore we need to double the backslash to get a backslash in the string, while the line-feed character (<lf>) is represented by backslash-n (\n) in string literals:

```
CALL ch.setDelimiter("|")
CALL ch.write("aaa\\bbb|ccc\nddd")   -- [aaa<bs>bbb|ccc<lf>ddd]
```

This code will produce the following text file:

```
aaa\\bbb\|ccc\
ddd|
```

When reading such a line back into memory with the `read()` method, all escaped characters are converted back to the single character. In this example, \\ becomes \, \| becomes | and \<lf> becomes <lf>.

When using the `read()`/`write()` methods, the escaped line-feed (LF, \n) characters are written as BS + LF to the output, and when reading with `read()`, BS + LF are detected and interpreted, to be restored as if the value was assigned by a `LET` instruction, with the same string used in the `write()` function.

If you want to write a LF as part of a value, the string must contain the backslash and line-feed as two independent characters. You need to escape the backslash when you write the string constant in the .4gl source file.

```
CALL ch.setDelimiter("|")
CALL ch.write("aaa\\\nbbb")  -- [aaa<bs><lf>bbb]
CALL ch.write("ccc\nddd")    -- [aaa<lf>bbb]
```

would generate the following output:

```
aaa\
bbb|
ccc|
ddd|
```

where the first two lines contain data for the same line, in the meaning of a Channel record.

When you read these lines back with a `read()` call, you get the following strings in memory:

```
Read 1: aaa<bs><lf>bbb
Read 2: ccc
Read 3: ddd
```

These reads would correspond to the following assignments when using string constants:

```
LET s = "aaa\\\nbbb"
LET s = "ccc"
LET s = "ddd"
```

Data can also be formatted as CVS (Comma Separated Values), when defining `"CVS"` as delimiter value:

```
CALL ch.setDelimiter("CVS")
```

This CVS format is similar to the standard channel format, with the following differences:

- Values in the file might be surrounded with double quotes (").
- If a value contains a comma or a NEWLINE, it is not escaped; the value must be quoted in the file.
- Double-quote characters in values are doubled in the output file and the output value must be quoted.
- Backslash characters are not escaped and are read as is; the value must be quoted.
- Leading and trailing blanks are kept (no truncation).
- No ending delimiter is expected at the end of the record line.

Read and write simple lines

When the channel is open, use the `readLine()`/`writeLine()` methods to read and write simple lines of data terminated by a line terminator.

When using the `readLine()` and `writeLine()` functions, a LF character represents the end of a line.

For example, a simple text file can look like this:

```
first line
second line
third line
```

To read the above text file, the code could be:

```
MAIN
  DEFINE i INTEGER
  DEFINE s STRING
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL ch.openFile("file.txt","r")
  LET i = 1
  WHILE TRUE
    LET s = ch.readLine()
    IF ch.isEof() THEN EXIT WHILE END IF
    DISPLAY i, " ", s
    LET i = i + 1
  END WHILE
  CALL ch.close()
END MAIN
```

LF characters escaped by a backslash are **not** interpreted as part of the line during a `readLine()` call.

When a line is written, any LF characters in the string will be written as is to the output. When a line is read, the LF escaped by a backslash is **not** interpreted as part of the line.

For example, this code:

```
CALL ch.writeLine("aaa\\\nbbb")  -- [aaa<bs><lf>bbb]
CALL ch.writeLine("ccc\nddd")    -- [aaa<lf>bbb]
```

would generate this output:

```
aaa\
bbb
ccc
ddd
```

and the subsequent `readLine()` will read four different lines, where the first line is ended by a backslash:

```
Read 1 aaa<bs>
Read 2 bbb
Read 3 ccc
```

```
Read 4 ddd
```

Line terminators on Windows™ and UNIX™

On Windows™ platforms, DOS formatted text files use CR/LF as line terminators. You can manage these type of files with the `base.Channel` class.

By default, on both Windows™ and UNIX™ platforms, when records are read from a DOS file with the `base.Channel` class, the CR/LF line terminator is removed. When a record is written to a file on Windows™, the lines are terminated with CR/LF in the file; on UNIX™, the lines are terminated with LF only.

To avoid the automatic translation of CR/LF on Windows™, you can use the `b` option of the `openFile()` and `openPipe()` methods. You can combine the `b` option with `r` or `w`, based on the read or write operations that you want to do.

```
CALL ch.openFile( "mytext.txt", "rb" )
```

On Windows™, when lines are read with the `b` option, only LF is removed from CR/LF line terminators; CR will be copied as a character part of the last field. In contrast, when lines are written with the `b` option, LF characters will not be converted to CR/LF.

On UNIX™, writing lines with or without the binary mode option does not matter.

Handle channel exceptions

Channel errors can be trapped with the `WHENEVER ERROR` exception handler:

```
WHENEVER ERROR CONTINUE
CALL ch.write([num,label])
IF STATUS THEN
  ERROR "An error occurred while reading from Channel"
  CALL ch.close()
  RETURN -1
END IF
WHENEVER ERROR STOP
```

Or with a `TRY/CATCH` block:

```
TRY
  CALL ch.write([num,label])
CATCH
  ERROR "An error occurred while reading from Channel"
  CALL ch.close()
  RETURN -1
END TRY
```

Implementing a TCP socket channel

The `base.Channel` class provides methods to implement basic TCP client and server programs. Consider character set encodings when designing such programs: No implicit character set conversion is done by the runtime system. Both client and server must use the same character set and length semantics.

The following code example implements a client program connecting to a TCP port, using the `openClientSocket()` method:

```
MAIN
  DEFINE ch base.Channel,
         time DATETIME HOUR TO SECOND,
         data STRING
  LET ch = base.Channel.create()
  CALL ch.openClientSocket("localhost",99999,"u",3)
  CALL ch.writeLine("get_time")
  LET time = ch.readLine()
```

```
  DISPLAY "client 1: ", time
  CALL ch.writeLine("get_string")
  LET data = ch.readLine()
  DISPLAY "client 2: ", data
  CALL ch.writeLine("disconnect")
  CALL ch.close()
END MAIN
```

The next code example implements the server program that can be used with the above client program. The server program uses the `openServerSocket()` and `readLine()` methods to listen to a given TCP interface/port. Note that the connection with a client must be ended by sending an EOF character (ASCII 26) to the client, the next `readLine()` call will wait for a new client connection, or select a pending client connection:

```
MAIN
  DEFINE ch base.Channel,
         cmd, data STRING
  LET ch = base.Channel.create()
  DISPLAY "starting server..."
  CALL ch.openServerSocket(null, 99999, "u")
  WHILE TRUE
      LET cmd = ch.readLine()
      IF ch.isEof() THEN
        DISPLAY "Connection ended by client..."
        EXIT WHILE
      END IF
      DISPLAY "cmd: ", cmd
      IF cmd == "get_time" THEN
        CALL ch.writeLine(CURRENT HOUR TO SECOND)
      END IF
      IF cmd == "get_string" THEN
        LET data = "This is a string..."
        CALL ch.writeLine(data)
      END IF
      IF cmd == "disconnect" THEN
         CALL ch.writeLine(ASCII 26) -- EOF
      END IF
  END WHILE
  DISPLAY "end of server..."
END MAIN
```

Examples

Example 1: Reading formatted data from a file

This program reads data from `file.txt`, which contains two columns separated by a pipe (`|`) character. It writes this data to the end of `fileout.txt`, using a percent sign (`%`) as the delimiter.

```
MAIN
  DEFINE custinfo RECORD
            cust_num INTEGER,
            cust_name VARCHAR(40)
         END RECORD
  DEFINE ch_in, ch_out base.Channel
  LET ch_in = base.Channel.create()
  CALL ch_in.setDelimiter("|")
  LET ch_out = base.Channel.create()
  CALL ch_out.setDelimiter("%")
  CALL ch_in.openFile("file.txt","r")
  CALL ch_out.openFile("fileout.txt","w")
  WHILE ch_in.read([custinfo.*])
    CALL ch_out.write([custinfo.*])
  END WHILE
```

```
  CALL ch_in.close()
  CALL ch_out.close()
END MAIN
```

Example 2: Executing the ls UNIX™ command

This program executes the `ls` command and displays the filenames and extensions separately.

```
MAIN
  DEFINE fn CHAR(40)
  DEFINE ex CHAR(10)
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL ch.setDelimiter(".")
  CALL ch.openPipe("ls -l","r")
  WHILE ch.read([fn,ex])
    DISPLAY fn, "   ", ex
  END WHILE
  CALL ch.close()
END MAIN
```

Example 3: Reading lines from a file

```
MAIN
  DEFINE i INTEGER
  DEFINE s STRING
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL ch.openFile("file.txt","r")
  LET i = 1
  WHILE TRUE
    LET s = ch.readLine()
    IF ch.isEof() THEN EXIT WHILE END IF
    DISPLAY i, " ", s
    LET i = i + 1
  END WHILE
  CALL ch.close()
END MAIN
```

Example 4: Communicating with an HTTP server

```
MAIN
  DEFINE ch base.Channel, eof INTEGER
  LET ch = base.Channel.create()
  -- HTTP protocol forces every line to be terminate by \r\n
  -- So we use channel binary mode to avoid CR+LF translation on Windows.
  -- In text mode, each line would be terminated by \r\r\n on Windows.
  WHENEVER ERROR CONTINUE
  CALL ch.openClientSocket("localhost", 80, "ub", 30)
  IF STATUS != 0 THEN
    DISPLAY "Could not open socket: error ", STATUS
    EXIT PROGRAM 1
  END IF
  WHENEVER ERROR STOP
  -- HTTP expects CR+LF: Note that LF is added by writeLine()!
  CALL ch.writeLine("GET / HTTP/1.0\r")
  -- No HTTP headers...
  -- Empty line = end of headers
  CALL ch.writeLine("\r")
  WHILE NOT eof
    DISPLAY ch.readLine()
    LET eof = ch.isEof()
  END WHILE
```

```
  CALL ch.close()
END MAIN
```

Example 5: Sending mails through an SMTP server

```
MAIN
    DEFINE mc base.Channel
    DEFINE i, res INTEGER
    DEFINE subject, emFrom, emRcpt, msg STRING
    DEFINE mailbody DYNAMIC ARRAY OF STRING
    LET subject = "Hello..."
    LET emFrom = "ted.fisher@4js.com"
    LET emRcpt = "ted.fisher@4js.com"
    LET mailbody[1] = "Hello,"
    LET mailbody[2] = "What's new?"
    LET mc = base.Channel.create()
    -- We use channel binary mode to avoid CR+LF translation on Windows.
    -- In text mode, each line would be terminated by \r\r\n on Windows.
    CALL mc.openClientSocket("mail.strasbourg.4js.com", 25, "ub", 5)
    CALL readSmtpAnswer(mc) RETURNING res, msg
    CALL smtpSend(mc, "HELO xxx\r") RETURNING res, msg
    CALL smtpSend(mc, SFMT("MAIL FROM: %1\r", emFrom)) RETURNING res, msg
    CALL smtpSend(mc, SFMT("RCPT TO: %1\r", emRcpt)) RETURNING res, msg
    CALL smtpSend(mc, "DATA\r") RETURNING res, msg
    DISPLAY "Sending mail body:"
    CALL mc.writeLine(SFMT("Subject: %1\r", subject))
    FOR i = 1 TO mailbody.getLength()
        CALL mc.writeLine(mailbody[i])
    END FOR
    CALL mc.writeLine(".")
    CALL readSmtpAnswer(mc) RETURNING res, msg
    DISPLAY "  Result: ", res
    CALL smtpSend(mc, "QUIT\r") RETURNING res, msg
    CALL mc.close()
END MAIN

FUNCTION smtpSend(ch, command)
    DEFINE ch base.Channel
    DEFINE command, msg STRING
    DEFINE res INTEGER
    DISPLAY "Sending command: ", command
    CALL ch.writeLine(command)
    CALL readSmtpAnswer(ch) RETURNING res, msg
    DISPLAY "  Result: ", res
    RETURN res, msg
END FUNCTION

FUNCTION readSmtpAnswer(ch)
    DEFINE ch base.Channel
    DEFINE line, msg STRING
    DEFINE res INTEGER
    LET msg = ""
    WHILE TRUE
        LET line = ch.readLine() -- Note: /r/n is already removed!
        IF line IS NULL THEN
            RETURN -1, "COULD NOT READ SMTP ANSWER"
        END IF
        IF line MATCHES "[0-9][0-9][0-9] *" THEN
            IF msg.getLength() != 0 THEN
                LET msg=msg || "\n"
            END IF
            LET msg=msg.append(line.subString(4, line.getLength()))
            LET res = line.subString(1,3)
```

```
            RETURN res, msg
        END IF
        IF line MATCHES "[0-9][0-9][0-9]-*" THEN
            IF msg.getLength() != 0 THEN
                LET msg=msg || "\n"
            END IF
            LET msg=msg.append(line.subString(4, line.getLength()))
        END IF
    END WHILE
END FUNCTION
```

The SqlHandle class

The `base.SqlHandle` class is a built-in class providing an API to execute parameterized SQL statements, with or without result sets.

base.SqlHandle methods

Table 371: Class methods

Name	Description
`create()` `RETURNING` *`handle`* `base.SqlHandle`	Create a new base.SqlHandle object.

Table 372: Object methods

Name	Description
`close()`	Closes the SQL handle (cursor).
`execute()`	Executes a simple SQL statement (without result set).
`fetch()`	Fetches a new row from the SQL result set.
`fetchAbsolute(`*`position`* `INTEGER)`	Fetches to a specified row in a scrollable SQL result set.
`fetchFirst()`	Fetches the first row in a scrollable SQL result set.
`fetchLast()`	Fetches the last row in a scrollable SQL result set.
`fetchPrevious()`	Fetches the previous row in a scrollable SQL result set.
`fetchRelative(`*`offset`* `INTEGER)`	Fetches a row relative to the current row in a scrollable SQL result set.
`flush()`	Flushes the rows from the insert cursor buffer.
`getResultCount()`	Returns the number of result set columns produced by the SQL statement.

Name	Description
`RETURNING count INTEGER`	
`getResultName( index INTEGER ) RETURNING name STRING`	Returns the name of a column in the result set produced by the SQL statement.
`getResultType( index INTEGER ) RETURNING type STRING`	Returns the Genero type name of a column in the result set produced by the SQL statement.
`getResultValue( index INTEGER ) RETURNING value fgl-type`	Returns the value of a column in the result set produced by the SQL statement.
`open()`	Opens the SQL handle (SELECT or INSERT cursor).
`openScrollCursor()`	Opens the SQL handle (with scrollable option).
`prepare( sql-text STRING )`	Prepares an SQL statement for the SQL handle.
`put()`	Put a new row in the insert cursor buffer.
`setParameter( index INTEGER, value fgl-type )`	Sets the value of an SQL parameter for this SQL handle.

base.SqlHandle.create
Create a new base.SqlHandle object.

Syntax

```
create()
 RETURNING handle base.SqlHandle
```

Usage

Use the `create()` method to create a base.SqlHandle object to execute SQL statements.

The value returned by this method must be assigned to a variable defined with the base.SqlHandle type.

As with other built-in classes, the SqlHandle object will be automatically destroyed if no longer referenced.

Example

```
DEFINE sh base.SqlHandle
LET sh = base.SqlHandle.create()
...
```

base.SqlHandle.close
Closes the SQL handle (cursor).

Syntax

```
close()
```

Usage

Call the `close()` method when you are done with the SQL handle.

The statement can be re-opened after it has been closed.

> **Note:** An SqlHandle object is automatically closed when the object is destroyed.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.close()
```

base.SqlHandle.execute
Executes a simple SQL statement (without result set).

Syntax

```
execute()
```

Usage

Call the `execute()` method to execute the prepared SQL statement, without producing a result set (INSERT, UPDATE, DELETE, CREATE TABLE, ...).

The SQL statement must have been prepared with a `prepare()` call.

If the SQL statement contains ? parameter place holders, issue a `setParameter()` call for each parameter before executing the SQL statement.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.execute()
```

base.SqlHandle.fetch
Fetches a new row from the SQL result set.

Syntax

```
fetch()
```

Usage

Call the `fetch()` method to fetch a new row from the SQL result set.

The SQL statement must have been opened with a `open()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(`*`index`*`)`, `getResultType(`*`index`*`)` and `getResultValue(`*`index`*`)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.fetch()
```

base.SqlHandle.fetchAbsolute
Fetches to a specified row in a scrollable SQL result set.

Syntax

```
fetchAbsolute(position INTEGER)
```

1. *position* is the absolute row position in the result set (starts at 1).

Usage

Call the `fetchAbsolute()` method to fetch to the specified row in a scrollable SQL result set.

The SQL statement must have been opened with a `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(`*`index`*`)`, `getResultType(`*`index`*`)` and `getResultValue(`*`index`*`)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the specified position does not correspond to a row position in the result set, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.fetchAbsolute(10)
```

base.SqlHandle.fetchFirst
Fetches the first row in a scrollable SQL result set.

Syntax

```
fetchFirst()
```

Usage

Call the `fetchFirst()` method to fetch the first row in a scrollable SQL result set.

The SQL statement must have been opened with a `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(`*`index`*`)`, `getResultType(`*`index`*`)` and `getResultValue(`*`index`*`)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.fetchFirst()
```

base.SqlHandle.fetchLast
Fetches the last row in a scrollable SQL result set.

Syntax

```
fetchLast()
```

Usage

Call the `fetchLast()` method to fetch the last row in a scrollable SQL result set.

The SQL statement must have been opened with a `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(`*`index`*`)`, `getResultType(`*`index`*`)` and `getResultValue(`*`index`*`)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.fetchLast()
```

base.SqlHandle.fetchPrevious
Fetches the previous row in a scrollable SQL result set.

Syntax

```
fetchPrevious()
```

Usage

Call the `fetchPrevious()` method to fetch to the previous row in a scrollable SQL result set.

The SQL statement must have been opened with a `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(`*`index`*`)`, `getResultType(`*`index`*`)` and `getResultValue(`*`index`*`)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, or if the current row is already the first row, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.fetchPrevious()
```

base.SqlHandle.fetchRelative
Fetches a row relative to the current row in a scrollable SQL result set.

Syntax

```
fetchRelative(offset INTEGER)
```

1. *offset* is the row offset in the result set. The offset can be negative, to fetch backwards.

Usage

Call the `fetchRelative()` method to fetch the row at the specified offset, relative to the current row in a scrollable SQL result set.

The SQL statement must have been opened with a `openScrollCursor()` call.

After performing the fetch call, you can query for column information with the `getResultCount()`, `getResultName(index)`, `getResultType(index)` and `getResultValue(index)` methods.

If no row is found (end of result set), SQLCA.SQLCODE is set to 100 (NOTFOUND).

If the result set is empty, or if no row exists at the specified offset relative to the current row position, SQLCA.SQLCODE is set to 100 (NOTFOUND).

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.fetchRelative(-3)
```

base.SqlHandle.flush
Flushes the rows from the insert cursor buffer.

Syntax

```
flush()
```

Usage

With an insert cursor, call the `flush()` method to force the buffered rows to the database server.

The SQL statement must have been opened with a `open()` call.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.flush()
```

base.SqlHandle.getResultCount
Returns the number of result set columns produced by the SQL statement.

Syntax

```
getResultCount()
 RETURNING count INTEGER
```

Usage

Call the `getResultCount()` method to query the number of columns in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

Example

```
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultName(i)
END FOR
```

base.SqlHandle.getResultName
Returns the name of a column in the result set produced by the SQL statement.

Syntax

```
getResultName( index INTEGER )
 RETURNING name STRING
```

1. *index* is the ordinal position of the result set column (starts at 1).

Usage

Call the `getResultName()` method to query the name of a column in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

The method takes the position of the column as the parameter.

Example

```
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultName(i)
END FOR
```

base.SqlHandle.getResultType
Returns the Genero type name of a column in the result set produced by the SQL statement.

Syntax

```
getResultType( index INTEGER )
 RETURNING type STRING
```

1. *index* is the ordinal position of the result set column (starts at 1).

Usage

Call the `getResultType()` method to query the type of a column in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

The method takes the position of the column as the parameter.

The type name is a string that represents a Genero type. For example, "INTEGER", "DECIMAL(10,2)", "DATE", "DATETIME YEAR TO SECOND".

> **Important:** The type returned can differ, depending on the brand of database server used. The database driver provides the column type according to the described API of the client database software, which in turn queries the database server for the native type of the column. For example, if you create a table in the Genero program with a DATE type in a Oracle database, the resulting DATE native type in Oracle will correspond to a Genero type of DATETIME YEAR TO SECOND.

Example

```
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultType(i)
END FOR
```

base.SqlHandle.getResultValue
Returns the value of a column in the result set produced by the SQL statement.

Syntax

```
getResultValue( index INTEGER )
 RETURNING value fgl-type
```

1. *index* is the ordinal position of the result set column (starts at 1).

Usage

Call the `getResultValue()` method to get the value of a column in the result set, after executing the SQL statement with the `open()` method and fetching a row with `fetch()`.

The method takes the position of the column as the parameter.

The value returned can be assigned to a program variable of the type corresponding to the type name returned by `getResultType()`.

> **Important:** TEXT and BYTE values are returned by reference. In order to get the value of a TEXT or BYTE column, define a variable of this type and assign the `getResultValue()` return. The returned TEXT or BYTE variable is already located in memory, there is no need to LOCATE the variable before calling `getResultValue()`.

```
DEFINE p_text TEXT
...
LET p_text = h.getResultValue(3)
...
```

Example

```
FOR i=1 TO sh.getResultCount()
    DISPLAY sh.getResultValue(i)
END FOR
```

base.SqlHandle.open
Opens the SQL handle (SELECT or INSERT cursor).

Syntax

```
open()
```

Usage

Call the `open()` method to execute the prepared SQL statement, and open the result set cursor or insert cursor.

The SQL statement must have been prepared with a `prepare()` call.

If the SQL statement contains ? parameters:

- For a statement with a result set (SELECT), values must be provided for each parameter before the `open()` call.
- For an insert cursor, values must be provided after the `open()` call, before each `put()` call.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.open()
```

base.SqlHandle.openScrollCursor
Opens the SQL handle (with scrollable option).

Syntax

```
openScrollCursor()
```

Usage

Call the `openScrollCursor()` method to execute a prepared SQL statement, and open the result set for use with a scrollable SQL cursor.

The SQL statement must have been prepared with a `prepare()` call.

If the SQL statement contains ? parameters, values must be provided for each parameter before the `openScrollCursor()` call.

After opening the scrollable cursor, use methods such as `fetchFirst()`, `fetchPrevious()` and `fetchAbsolute(n)` to move forwards and backwards in the SQL result set.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.openScrollCursor()
```

base.SqlHandle.prepare
Prepares an SQL statement for the SQL handle.

Syntax

```
prepare( sql-text STRING )
```

Usage

Call the `prepare()` method to prepare the SQL statement that will be executed with either `execute()` or `open()`.

The SQL statement can contain ? parameter place holders, to be filled with the `setParameter()` method before executing the statement.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.prepare("INSERT INTO mytable VALUES (?,?)")
```

base.SqlHandle.put
Put a new row in the insert cursor buffer.

Syntax

```
put()
```

Usage

Call the `put()` method to create a new row for the insert cursor.

The SQL statement must have been prepared with a `prepare()` call.

All SQL parameter values must be provided before doing the `put()` call.

As with standard Genero SQL instructions, SQL errors can be trapped with `WHENEVER ERROR` or `TRY / CATCH` blocks and by testing SQLCA.SQLCODE.

Example

```
CALL sh.put()
```

base.SqlHandle.setParameter
Sets the value of an SQL parameter for this SQL handle.

Syntax

```
setParameter(
   index INTEGER,
   value fgl-type )
```

1. *index* is the ordinal position of the ? SQL parameter (starts at 1).
2. *value* is the variable containing the parameter value.

Usage

Call the `setParameter()` method to define the value of an SQL parameter specified with a ? place holder in the string passed to the `prepare()` method.

The SQL statement must have been prepared with a `prepare()` call.

It is possible to pass numeric and string constants directly to the method, but type conversion cannot be done without a program variable.

Example

```
DEFINE v_pk INT, v_crea DATETIME YEAR TO SECOND
...
CALL sh.setParameter(1,v_pk)
CALL sh.setParameter(2,v_crea)
```

Usage

The `base.SqlHandle` class is a built-in class providing dynamic SQL support with a 3GL API.

Compared to regular SQL cursor instructions, the main purpose of the `base.SqlHandle` class is to provide column name and SQL data type information with the `getResultName()` and `getResultType()` methods. It is also possible to write generic code for parameterized queries with the `setParameter()` method.

Important: A database connection must exist in order to use SqlHandle objects.

Unlike regular Genero cursors, SQL handle objects are created dynamically, and can be passed as parameter or returned from functions:

```
MAIN
   DEFINE h base.SqlHandle
   CONNECT TO "mydb"
   LET h = base.SqlHandle.create()
   CALL my_prepare(h)
   CALL my_execute(h)
END MAIN

FUNCTION my_prepare(h)
   DEFINE h base.SqlHandle
   CALL h.prepare("INSERT INTO cust VALUES ( ...
END FUNCTION

FUNCTION my_execute(h)
   DEFINE h base.SqlHandle
   CALL h.execute()
END FUNCTION
```

Executing a simple SQL statement without a result set

Perform the following steps, to execute an SQL statement without a result set:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create an SQL handle object `base.SqlHandle.create()`
3. `prepare(`*`sql-text`*`)`
4. For each SQL parameter:
   a. `setParameter(`*`index, value`*`)`
5. `execute()` -- test for `SQLCA.SQLCODE`
6. Repeat from (5), (4), or (3)

Executing a SQL statement returning a result set

Perform the following steps, to execute an SQL statement with a result set:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create an SQL handle object `base.SqlHandle.create()`
3. `prepare(`*`sql-text`*`)`
4. For each SQL parameter:
   a. `setParameter(`*`index, value`*`)`
5. `open()`
6. `fetch()` -- test for `SQLCA.SQLCODE == 100`
7. `getResultCount()` -- for each column index:
   a. `getResultName(`*`index`*`)`
   b. `getResultType(`*`index`*`)`
   c. `getResultValue(`*`index`*`)`
8. `close()`
9. Repeat from (6), (4), (5), or (3)

Executing a SQL statement returning a result set, as scrollable cursor

Perform the following steps, to execute an SQL statement with a result set and scroll forwards and backwards in the rows:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create an SQL handle object `base.SqlHandle.create()`
3. `prepare(`*`sql-text`*`)`
4. For each SQL parameter:
   a. `setParameter(`*`index, value`*`)`
5. `openScrollCursor()`
6. `fetch()` (next row), `fetchLast()`, `fetchFirst()`, `fetchPrevious()`, `fetchRelative(n)` or `fetchAbsolute(n)` -- test for `SQLCA.SQLCODE == 100`
7. `getResultCount()` -- for each column index:
   a. `getResultName(`*`index`*`)`
   b. `getResultType(`*`index`*`)`
   c. `getResultValue(`*`index`*`)`
8. `close()`
9. Repeat from (6), (4), (5), or (3)

Creating rows with an insert cursor

Perform the following steps, to insert many rows with an SQL handle insert cursor:

1. Define the SQL handle variable as `base.SqlHandle`
2. Create an SQL handle object `base.SqlHandle.create()`
3. `prepare(`*`insert-stmt-with-params`*`)`
4. `BEGIN WORK`
5. `open()`
6. For each row to insert:
   a. For each SQL parameter:
      a. `setParameter(`*`index, value`*`)`
   b. `put()`
7. `close()`

8. `COMMIT WORK`
9. Repeat from (4) or (3)

SQL error handling with SqlHandle

Handling SQL error and status information (such as `NOTFOUND`) can be done with SqlHandle objects as with regular SQL instruction, by testing the `SQLCA.SQLCODE` register, and by using `TRY/CATCH` blocks or `WHENEVER ERROR`.

```
MAIN
   DEFINE h base.SqlHandle
   CONNECT TO "mydb"
   LET h = base.SqlHandle.create()
   TRY
       CALL h.prepare("SELECT * FROM mytab")
       CALL h.open()
       CALL h.fetch()
       DISPLAY h.getResultValue(1)
       CALL h.close()
   CATCH
       DISPLAY "SQL ERROR:", SQLCA.SQLCODE
   END TRY
END MAIN
```

Examples

Example 1: SQL statement without a result set

The following code executes a simple UPDATE statement with the base.SqlHandle API:

```
MAIN
    DEFINE h base.SqlHandle

    CONNECT TO "mydb"

    LET h = base.SqlHandle.create()

    CALL h.prepare("UPDATE t1 SET name = ? WHERE pk = ?")

    CALL h.setParameter(1, "Scott")
    CALL h.setParameter(2, "8723")

    TRY
         CALL h.execute()
    CATCH
         DISPLAY "Error detected: ", SQLCA.SQLCODE
    END TRY

END MAIN
```

Example 2: SQL statement with a result set

The following code executes a simple SELECT statement with the base.SqlHandle API:

```
MAIN
    DEFINE h base.SqlHandle,
           d DATE,
           i INTEGER

    CONNECT TO "mydb"

    LET h = base.SqlHandle.create()
```

```
    CALL h.prepare("SELECT * FROM t1 WHERE created > ?")

    LET d = TODAY
    CALL h.setParameter(1, d)

    CALL h.open()

    WHILE TRUE
        CALL h.fetch()
        IF SQLCA.SQLCODE==NOTFOUND THEN EXIT WHILE END IF
        DISPLAY "-----------------"
        FOR i=1 TO h.getResultCount()
            DISPLAY i, ":", h.getResultName(i),
                    " / ", h.getResultType(i),
                    " = ", h.getResultValue(i)
        END FOR
    END WHILE

    CALL h.close()

END MAIN
```

The StringBuffer class

The `base.StringBuffer` class is a built-in class designed to manipulate character strings.

This class is optimized for string operations such as scanning, replacements, concatenation.

Use the `base.StringBuffer` class instead of `STRING` variables to implement heavy string manipulations. When you use a `base.StringBuffer` object, you work directly on the internal string buffer. When you use the `STRING` data type and modify a string, the runtime system creates a new buffer. While this does not impact the performance of programs with a user interface or even batch programs doing SQL, it can impact performance when you need to rapidly process large character strings. For example, if you need to process 500 KB of text (such as when you are performing a global search-and-replace of specific words), you get much better performance with a `base.StringBuffer` object than you would with a `STRING` variable.

When you pass a `base.StringBuffer` object as a function parameter, the function receives a variable that references the object. Passing the object by reference is much more efficient than using a `STRING` that is passed by value, because `STRING` data is copied on the stack. The function manipulates the original string, not a copy of the string.

> **Important:** The methods of this class use character positions and string length. When using byte length semantics, the length is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

base.StringBuffer methods

Table 373: Class methods

Name	Description
`base.StringBuffer.create()` `  RETURNING` *result* `base.StringBuffer`	Create a string buffer object.

Table 374: Object methods

Name	Description
`append(`	Append a string at the end of the current string.

Name	Description
part STRING)	
clear()	Clear the string buffer.
equals(*reference* STRING) RETURNING *result* BOOLEAN	Compare strings (case sensitive).
equalsIgnoreCase(*reference* STRING) RETURNING *result* BOOLEAN	Compare strings (case insensitive)
getCharAt(*position* INTEGER) RETURNING *result* STRING	Return the character at a specified position.
getIndexOf(*substr* STRING, *start* INTEGER) RETURNING *result* INTEGER	Return the position of a substring.
getLength() RETURNING *result* INTEGER	Return the length of a string.
insertAt(*part* STRING, *pos* INTEGER)	Insert a string at a given position.
replace(*old* STRING, *new* STRING, *occ* INTEGER)	Replace one string with another.
replaceAt(*start* INTEGER, *length* INTEGER, *new* STRING)	Replace part of a string with another string.
subString(*start* INTEGER, *end* INTEGER)	Return the substring at the specified position.

Name	Description
RETURNING *result* STRING	
toLowerCase()	Converts the string in the buffer to lower case.
toUpperCase()	Converts the string in the buffer to upper case.
toString() RETURNING *result* STRING	Create a STRING from the string buffer.
trim()	Remove leading and trailing blanks.
trimLeft()	Removes leading blanks.
trimRight()	Removes trailing blanks.

base.StringBuffer.create
Create a string buffer object.

Syntax

```
base.StringBuffer.create()
  RETURNING result base.StringBuffer
```

Usage

Use the `base.StringBuffer.create()` class method to create a string buffer object.

The new created object must be assigned to a program variable defined with the `base.StringBuffer` type.

Example

```
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
```

base.StringBuffer.append
Append a string at the end of the current string.

Syntax

```
append(
   part STRING )
```

1. *part* is the string to append to the string buffer.

Usage

The `append()` method appends a string to the internal string buffer.

Example

```
LET buf = base.StringBuffer.create()
CALL buf.append("abc")
```

base.StringBuffer.clear
Clear the string buffer.

Syntax

```
clear()
```

Usage

Use the `clear()` method to clear the string buffer.

After clearing, the string buffer is empty and the length is zero.

Example

```
CALL buf.clear()
```

base.StringBuffer.equals
Compare strings (case sensitive).

Syntax

```
equals(
   reference STRING )
  RETURNING result BOOLEAN
```

1. *reference* is the string to compare with.

Usage

Use the `equals()` method to determine whether the value of a `base.StringBuffer` object is identical to a specified string.

This method is case-sensitive.

Since the parameter for the method must be a string, you can use the `toString()` method to convert a `base.StringBuffer` object in order to compare it.

The method returns `TRUE` if the strings are identical, otherwise it returns `FALSE`.

Example

```
DEFINE buf, buf2 base.StringBuffer,
       mystring STRING
LET buf = base.StringBuffer.create()
CALL buf.append("there")

-- compare to a STRING
IF buf.equals("there") THEN
   DISPLAY "buf matches there"
END IF

-- compare to a STRING variable
```

```
LET mystring = "there"
IF buf.equals(mystring) THEN
   DISPLAY "buf matches mystring"
END IF

-- compare to another StringBuffer object
LET buf2 = base.StringBuffer.create()
CALL buf2.append("there")
IF buf.equals(buf2.toString()) THEN
   DISPLAY "buf matches buf2"
END IF
```

Output:

```
buf matches there
buf matches mystring
buf matches buf2
```

base.StringBuffer.equalsIgnoreCase
Compare strings (case insensitive)

Syntax

```
equalsIgnoreCase(
   reference STRING )
  RETURNING result BOOLEAN
```

1. *reference* is the string to compare with.

Usage

The `equalsIgnoreCase()` method compares the current string buffer with the passed string, ignoring the character case.

Since the parameter for the method must be a string, you can use the `toString()` method to convert a `base.StringBuffer` object in order to compare it.

The method returns `TRUE` if the strings are identical, otherwise it returns `FALSE`.

Example

```
DEFINE buf3 base.StringBuffer
LET buf3 = base.StringBuffer.create()
CALL buf3.append("there")
IF buf3.equalsIgnoreCase("There") THEN
   DISPLAY "buf matches There ignoring case"
END IF
```

Output:

```
buf matches There ignoring case
```

base.StringBuffer.getCharAt
Return the character at a specified position.

Syntax

```
getCharAt(
   position INTEGER )
```

```
   RETURNING result STRING
```

1. *position* is the character position in the string.

Usage

The `getCharAt()` method returns the character from the string buffer at the position that you specify.

The first character position is 1.

The method returns `NULL` if the position is lower as 1 or greater as the length of the string.

> **Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

Example

```
DEFINE buf base.StringBuffer
LET buf = base.StringBuffer.create()
CALL buf.append("abcdef")
DISPLAY buf.getCharAt(3)  -- Shows c
```

base.StringBuffer.getIndexOf
Return the position of a substring.

Syntax

```
getIndexOf(
   substr STRING,
   start INTEGER )
  RETURNING result INTEGER
```

1. *substr* is the substring to be found.
2. *start* is the starting position.

Usage

The `getIndexOf()` method returns the position of a substring in the string buffer. Specify the substring and an integer specifying the position at which the search should begin. Use 1 if you want to start at the beginning of the string buffer.

The method returns zero if the substring is not found.

```
CALL buf.append("abcdef")
DISPLAY buf.getIndexOf("def",1) -- Shows 4
```

> **Important:** When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

Example

This example iterates through the complete string to display the position of multiple occurrences of the same substring.

```
MAIN
  DEFINE b base.StringBuffer
  DEFINE pos INTEGER
  DEFINE s STRING
```

```
  LET b = base.StringBuffer.create()
  CALL b.append("---abc-----abc--abc----")
  LET pos = 1
  LET s = "abc"
  WHILE TRUE
    LET pos = b.getIndexOf(s,pos)
    IF pos == 0 THEN
      EXIT WHILE
    END IF
    DISPLAY "Pos: ", pos
    LET pos = pos + length(s)
  END WHILE
END MAIN
```

base.StringBuffer.getLength
Return the length of a string.

Syntax

```
getLength()
  RETURNING result INTEGER
```

Usage

Use the `getLength()` method to return the number of characters in the current string buffer, including trailing spaces.

The length of an empty string buffer is 0.

> **Important:** When using byte length semantics, the string length is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

Example

```
CALL buf.append("abc")
DISPLAY buf.getLength() -- Shows 3
-- append three spaces to the end of the string
CALL buf.append("   ")
DISPLAY buf.getLength() -- Shows 6
```

base.StringBuffer.insertAt
Insert a string at a given position.

Syntax

```
insertAt(
   part STRING,
   pos INTEGER )
```

1. *part* is the string part to be inserted.
2. *pos* is the position where the string must be inserted.

Usage

The `insertAt()` method inserts a string before the specified position in the string buffer.

Important: When using byte length semantics, the position is expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

Example

```
CALL buf.append("abcdef")
CALL buf.insertAt(3, "xx")
DISPLAY buf.toString() -- Shows abcxxdef
```

base.StringBuffer.replace
Replace one string with another.

Syntax

```
replace(
   old STRING,
   new STRING,
   occ INTEGER )
```

1. *old* is the string to be replaced.
2. *new* is the new string replacing the old string.
3. *occ* is the number of replacements to do.

Usage

The `replace()` method replaces a string within the current string buffer with a different string. Specify the original string, replacement string, and the number of occurrences to replace. Use 0 to replace all occurrences.

Example

```
CALL buf.append("aaxxbbxxcc")
CALL buf.replace("xx", "zz", 1)
DISPLAY buf.toString() -- Shows aazzbbxxcc
```

base.StringBuffer.replaceAt
Replace part of a string with another string.

Syntax

```
replaceAt(
   start INTEGER,
   length INTEGER,
   new STRING )
```

1. *start* is position where the replacement starts.
2. *length* is the number of characters to be replaced.
3. *new* is the replacement string.

Usage

The `replaceAt()` method replaces part of the current string with another string.

The parameters are integers indicating the position at which the replacement should start, the number of characters to be replaced, and the replacement string.

The first position in the string is 1.

> **Important:** When using byte length semantics, the position and length are expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

Example

```
CALL buf.append("abxxxxef")
CALL buf.replaceAt(3,4,"cd")
DISPLAY buf.toString() -- Shows abcdef
```

base.StringBuffer.subString
Return the substring at the specified position.

Syntax

```
subString(
   start INTEGER,
   end INTEGER )
  RETURNING result STRING
```

1. *start* is the substring to be found.
2. *end* is the ending position.

Usage

The `subString()` method returns the substring defined by the start and end positions passed as parameter.

The first character is at position 1.

> **Important:** When using byte length semantics, the positions are expressed in bytes. When using char length semantics, the unit is characters. This is matters when using a multibyte locale such as UTF-8.

Example

```
CALL buf.append("abcdefg")
DISPLAY buf.subString(2,5) -- Shows bcde
```

base.StringBuffer.toLowerCase
Converts the string in the buffer to lower case.

Syntax

```
toLowerCase()
```

Usage

The `toLowerCase()` method converts the current string to lower case.

Example

```
CALL buf.append("AbC")
CALL buf.toLowerCase()
```

```
DISPLAY buf.toString() -- Shows abc
```

base.StringBuffer.toString
Create a `STRING` from the string buffer.

Syntax

```
toString()
  RETURNING result STRING
```

Usage

The `toString()` method creates a `STRING` value from the current string buffer.

Use this method if you need to pass the string to another method or instruction that expects a `STRING` as parameter.

Example

```
CALL buf.append("abc")
DISPLAY buf.toString() -- Shows abc
```

base.StringBuffer.toUpperCase
Converts the string in the buffer to upper case.

Syntax

```
toUpperCase()
```

Usage

The `toUpperCase()` method converts the current string to upper case.

Example

```
CALL buf.append("AbC")
CALL buf.toUpperCase()
DISPLAY buf.toString() -- Shows ABC
```

base.StringBuffer.trim
Remove leading and trailing blanks.

Syntax

```
trim()
```

Usage

The `trim()` method removes the leading and trailing blanks in the string buffer.

Example

```
CALL buf.append("  abc  ")
CALL buf.trim()
```

```
DISPLAY "["||buf.toString()||"]" -- Shows [abc]
```

base.StringBuffer.trimLeft
Removes leading blanks.

Syntax

```
trimLeft()
```

Usage

The `trimLeft()` method removes the leading blanks in the string buffer.

Example

```
CALL buf.append("  abc  ")
CALL buf.trimLeft()
DISPLAY "["||buf.toString()||"]" -- Shows [abc  ]
```

base.StringBuffer.trimRight
Removes trailing blanks.

Syntax

```
trimRight()
```

Usage

The `trimRight()` method removes the trailing blanks in the string buffer.

```
CALL buf.append("  abc  ")
CALL buf.trimRight()
DISPLAY "["||buf.toString()||"]" -- Shows [  abc]
```

Examples
Example 1: Add strings to a StringBuffer

```
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL buf.append("abc")
  DISPLAY buf.toString()
  CALL buf.append("def")
  DISPLAY buf.toString()
  CALL buf.append(123456)
  DISPLAY buf.toString()
END MAIN
```

Output:

```
abc
abcdef
abcdef123456
```

Example 2: Modify a StringBuffer with a function

```
MAIN
  DEFINE buf base.StringBuffer
  LET buf = base.StringBuffer.create()
  CALL modify(buf)
  DISPLAY "buf is ", buf.toString()
END MAIN

FUNCTION modify(sb)
  DEFINE sb base.StringBuffer
  CALL sb.append("more")
  DISPLAY "sb is ", sb.toString()
END FUNCTION
```

Output:

```
sb is more
buf is more
```

The StringTokenizer class

The `base.StringTokenizer` class is designed to parse a string to extract tokens according to delimiters.

The steps to use a string tokenizer are:

1. Define a variable with the `base.StringTokenizer` type.
2. Create the string tokenizer object with one of the create methods, passing the string to be parsed as parameter.
3. Optionally, count the number of tokens with `countTokens()` before processing.
4. Use a `WHILE` loop to process the different tokens, by using `hasMoreTokens()` as loop condition and `nextToken()` inside the loop body to get the next token.

base.StringTokenizer methods

Table 375: Class methods

Name	Description
`base.StringTokenizer.create( source STRING, delims STRING ) RETURNING result base.StringTokenizer`	Create a string tokenizer object.
`base.StringTokenizer.createExt( source STRING, delims STRING, escape STRING, nulls BOOLEAN ) RETURNING result base.StringTokenizer`	Create a string tokenizer object with escape char and null handling.

Table 376: Object methods

Name	Description
`countTokens()`	Returns the number of tokens left to be returned.

Name	Description
`RETURNING result INTEGER`	
`hasMoreTokens()` `RETURNING result BOOLEAN`	Returns TRUE if there are more tokens to return.
`nextToken()` `RETURNING result STRING`	Returns the next token found in the source string.

base.StringTokenizer.create
Create a string tokenizer object.

Syntax

```
base.StringTokenizer.create(
   source STRING,
   delims STRING )
  RETURNING result base.StringTokenizer
```

1. *source* is the character string to be parsed.
2. *delims* defines the delimiters to be used.

Usage

Use the `base.StringTokenizer.create()` class method to create a string tokenizer object.

The new created object must be assigned to a program variable defined with the `base.StringTokenizer` type.

The method can take a unique or multiple delimiters into account. A delimiter is always one character long.

The empty tokens are not taken into account, and no escape character is defined for the delimiters. The `nextToken()` method will never return `NULL` strings.

> **Note:** To specify a backslash as a delimiter, you must use double backslashes in both the source string and as the delimiter, as shown in Example 3: Specify a backslash as a delimiter on page 1752

Example

```
DEFINE tok base.StringTokenizer
-- Using a single pipe delimiter
LET tok = base.StringTokenizer.create("aaa|bbb|ccc","|")
-- Using several delimiters
LET tok = base.StringTokenizer.create("aaa|bbb;ccc+ddd","|+;")
```

base.StringTokenizer.createExt
Create a string tokenizer object with escape char and null handling.

Syntax

```
base.StringTokenizer.createExt(
   source STRING, delims STRING,
   escape STRING, nulls BOOLEAN )
  RETURNING result base.StringTokenizer
```

1. *source* is the character string to be parsed.
2. *delims* defines the delimiters to be used.
3. *escape* defines the escape character.
4. *nulls* indicates if empty tokens must be returned.

Usage

Use the `base.StringTokenizer.createExt()` class method to create a string tokenizer object, with escape character and null token handling.

The new created object must be assigned to a program variable defined with the `base.StringTokenizer` type.

The method can take a unique or multiple delimiters into account. A delimiter is always one character long.

When defining an escape character with the third parameter, the delimiters can be escaped in the source string.

When passing `TRUE` to the last parameter, the empty tokens are taken into account. The `nextToken()` method might return `NULL` strings. In the source string, leading and trailing delimiters or the amount of delimiters between two tokens affects the number of tokens.

> **Note:** To specify a backslash as a delimiter, you must use double backslashes in both the source string and as the delimiter, as shown in Example 3: Specify a backslash as a delimiter on page 1752

Example

```
DEFINE tok base.StringTokenizer
LET tok = base.StringTokenizer.createExt("|aaa||b\\|bb|
ccc","|","\\",TRUE)
```

base.StringTokenizer.countTokens
Returns the number of tokens left to be returned.

Syntax

```
countTokens()
  RETURNING result INTEGER
```

Usage

Use the `countTokens()` method to count the number of tokens left to be returned by the string tokenizer.

This method can be used to know the number of tokens before processing the source string with the `hasMoreTokens()` and `nextToken()` methods.

base.StringTokenizer.hasMoreTokens
Returns `TRUE` if there are more tokens to return.

Syntax

```
hasMoreTokens()
  RETURNING result BOOLEAN
```

Usage

The `hasMoreTokens()` method indicates if there are other tokens in the source string that are not yet processed.

Use the `hasMoreTokens()` method typically as the expression of a `WHILE` block.

base.StringTokenizer.nextToken
Returns the next token found in the source string.

Syntax

```
nextToken()
  RETURNING result STRING
```

Usage

The `nextToken()` method parses the source string for tokens, according to the creation method used, and returns the next token if found.

The method returns `NULL` if no token is found, or if an empty token was found and the nulls parameter of the createExt() method was set to `TRUE`.

Use the `hasMoreTokens`() method to check if more tokens are to be read.

Examples

Example 1: Split a UNIX™ directory path

```
MAIN
  DEFINE tok base.StringTokenizer
  LET tok = base.StringTokenizer.create("/home/tomy","/")
  WHILE tok.hasMoreTokens()
    DISPLAY tok.nextToken()
  END WHILE
END MAIN
```

Example 2: Escaped delimiters and NULL tokens

```
MAIN
  DEFINE tok base.StringTokenizer
  LET tok = base.StringTokenizer.createExt("||\\|aaa||bbc|","|","\\",TRUE)
  WHILE tok.hasMoreTokens()
    DISPLAY tok.nextToken()
  END WHILE
END MAIN
```

Example 3: Specify a backslash as a delimiter

```
MAIN
  DEFINE tok base.StringTokenizer
  LET tok = base.StringTokenizer.create("C:\\My Documents\\My Pictures","\
\")
  WHILE tok.hasMoreTokens()
    DISPLAY tok.nextToken()
  END WHILE
END MAIN
```

The TypeInfo class

The `base.TypeInfo` class creates a DOM node from a structured program variable.

This class does not have to be instantiated.

Steps to use the class:

- Define a variable with the `om.DomNode` type.

- Create a channel object with `base.TypoInfo.create( var )` and assign it to the DOM node variable.
- Use the new created DOM node.

For example. to convert a list of database records to XML, fetch rows from a database table in a structured array, specify the array as the input parameter for the `base.TypeInfo.create()` method to create a new `base.DomNode` object, and serialize the resulting DOM node to a file by using the `node.writeXml()` method. You can then pass the resulting file to any application that is able to read XML for input.

> **Note:** Consider using the JSON interface to serialize and de-serialize program variables.

base.TypeInfo methods

Table 377: Class methods

Name	Description
`base.TypeInfo.create()` `  RETURNING result om.DomNode`	Create a DomNode from a structured program variable.

base.TypeInfo.create()
Create a DomNode from a structured program variable.

Syntax

```
base.TypeInfo.create()
  RETURNING result om.DomNode
```

Usage

Use the `base.TypeInfo.create()` class method to create a `om.DomNode` object from a program variable.

The program variable is typically a `RECORD`, but it can be any sort of structured variable, including arrays.

The `om.DomNode` is created with type information and values.

The data is formatted according to current environment settings (DBDATE, DBFORMAT, and DBMONEY).

Example

```
MAIN
  DEFINE n om.DomNode
  DEFINE r RECORD
      key INTEGER,
      lastname CHAR(20),
      birthdate DATE
  END RECORD
  LET r.key = 234
  LET r.lastname = "Johnson"
  LET r.birthdate = MDY(12,24,1962)
  LET n = base.TypeInfo.create( r )
  CALL n.writeXml( "r.xml" )
END MAIN
```

The generated node contains variable values and data type information. The example creates this file:

```
<?xml version="1.0"? encoding="ISO-8859-1">
<Record>
  <Field type="INTEGER" value="234" name="key"/>
  <Field type="CHAR(20)" value="Johnson" name="lastname"/>
  <Field type="DATE" value="12/24/1962" name="birthdate"/>
</Record>
```

The MessageServer class

The `base.MessageServer` class allows a program to send a key action over the network to other programs using this service.

This class can be used to join a group of programs to be notified by simple messages (i.e. key events). The programs can run on different machines connected together in a network.

> **Important:** This feature is experimental and subject to change.

The `base.MessageServer` uses network API capabilities with Sockets and the UDP protocol. The computers must be configured with a network. The UDP protocol does not guarantee the transmission of datagrams, therefore messages sent with the MessageServer can arrive out of order, duplicated, or go missing without notice.

The UDP port is `6600` and the IP address group is `224.0.1.1`. These cannot be changed.

> **Important:** This feature is only supported in direct connection with the GDC front-end. It is not supported when using other front-ends or when using the GAS.

base.MessageServer methods

Table 378: Class methods

Name	Description
`base.MessageServer.connect()`	Connects to the group of programs to be notified by a message.
`base.MessageServer.send(` `  keyname STRING )`	Sends a key event to the group of programs connected together.

base.MessageServer.connect

Connects to the group of programs to be notified by a message.

Syntax

```
base.MessageServer.connect()
```

Usage

Use the `connect()` method to join the group of programs that can be notified by a key event message.

base.MessageServer.send
Sends a key event to the group of programs connected together.

Syntax

```
base.MessageServer.send(
   keyname STRING )
```

1. *keyname* is a string expression defining the key event to be sent over the network.

Usage

Once connected to the message server group with `base.MessageServer.connect()`, a program calls the `base.MessageServer.send()` class method to notify other programs registered to the group.

```
CALL base.MessageServer.send("f1")
```

All programs registered to the message server group are notified, including the program which has sent the message. The messages can be treated by the current dialog with a simple `ON KEY()` interaction block.

Examples

Example 1: Simple MessageServer usage

```
MAIN
  CALL base.MessageServer.connect()
  MENU "test"
    COMMAND "send F1" CALL base.MessageServer.send("f1")
    ON KEY (F1) DISPLAY "Key F1 received..."
    COMMAND "quit" EXIT MENU
  END MENU
END MAIN
```

The ui package

These topics cover the built-in classes for the ui class

- The Interface class on page 1755
- The Window class on page 1769
- The Form class on page 1774
- The Dialog class on page 1784
- The ComboBox class on page 1820
- The DragDrop class on page 1827

The Interface class

The `ui.Interface` class provides methods to manipulate the user interface.

This class does not have to be instantiated.

ui.Interface methods

Methods of the `ui.Interface` class

Table 379: Class methods

Name	Description

`ui.Interface.frontCall(`
`  module STRING,`
`  function STRING,`
`  [ parameter-list ],`
	`ui.Interface.frontCall` performs a function call to the current front-end.

Name	Description
`[ returning-list ] )`	
`ui.Interface.filenameToURI( filename STRING ) RETURNING uri INTEGER`	Converts a file name to an URI to be used as a web component image resource.
`ui.Interface.getChildCount() RETURNING result INTEGER`	Get the number of children in a parent container.
`ui.Interface.getChildInstances( name STRING ) RETURNING result INTEGER`	Get the number of child instances for a given program name.
`ui.Interface.getContainer() RETURNING result STRING`	Get the parent container of the curren program.
`ui.Interface.getDocument() RETURNING result om.DomDocument`	Returns the DOM document of the abstract user interface tree.
`ui.Interface.getFrontEndName() RETURNING result STRING`	Returns the type of the front-end currently in use.
`ui.Interface.getFrontEndVersion() RETURNING result STRING`	Returns the version of the front-end currently in use.
`ui.Interface.getName() RETURNING result STRING`	`ui.Interface.frontCall` performs a function call to the current front-end.
`ui.Interface.getImage() RETURNING result STRING`	`ui.Interface.frontCall` performs a function call to the current front-end.
`ui.Interface.getText() RETURNING result STRING`	Returns the title of the program.
`ui.Interface.getType() RETURNING result STRING`	Returns the type of the program.
`ui.Interface.getRootNode() RETURNING result om.DomNode`	Get the root DOM node of the abstract user interface.
`ui.Interface.loadActionDefaults(`	Load the default action defaults file.

Name	Description
filename STRING)	
ui.Interface.loadStartMenu(*filename* STRING)	Load the start menu file.
ui.Interface.loadToolBar(*filename* STRING)	Load a default toolbar file.
ui.Interface.loadTopMenu(*filename* STRING)	Load a default topmenu file.
ui.Interface.loadStyles(*filename* STRING)	Load the presentation styles file.
ui.Interface.setImage(*icon* STRING)	Defines the icon image of the program.
ui.Interface.setName(*name* STRING)	Define the name of the current program for the front-end.
ui.Interface.setText(*title* STRING)	Defines the title for the program.
ui.Interface.setType(*type* STRING)	Defines the type of the program for the front-end.
ui.Interface.setSize(*height* INTEGER, *width* INTEGER)	Specify the initial size of the parent container window.
ui.Interface.setContainer(*name* STRING)	Define the parent container for the current program.
ui.Interface.refresh()	Synchronize the user interface with the front-end.

ui.Interface.frontCall
`ui.Interface.frontCall` performs a function call to the current front-end.

Syntax

```
ui.Interface.frontCall(
   module STRING,
   function STRING,
   [ parameter-list ],
```

```
    [ returning-list ] )
```

1. *module* defines the shared library or classpath where the function is implemented.
2. *function* defines the name of the function to be called.
3. *parameter-list* is a list of input parameters.
4. *returning-list* is a list of output parameters.

> **Important:** The *returning-list* variables are passed by reference to the `frontCall()` method.

Usage

The `ui.Interface.frontCall()` class method can be used to execute a procedure on the front-end workstation through the front-end software component. You can for example launch a front-end specific application like a browser or a text editor, or manage the clipboard content.

The method takes four parameters:

1. The module, identifying the shared library (.so or .DLL) or the Java class (GMA) implementing the front call function.
2. The function of the module the be executed.
3. The list of input parameters, using the square brace notation.
4. The list of output parameters, using the square brace notation.

Input and output parameters are provided as a variable list of parameters, by using the square braces notation (`[param1,param2,...]`). Input parameters can be an expression supported by the language; output parameters must be variables only, to receive the returning values. An empty list is specified with `[]`. Output parameters are optional: If the front call returns values, they will be ignored by the runtime system.

Simple front call example:

```
FUNCTION call()
  DEFINE info STRING
  CALL ui.Interface.frontCall( "standard", "feInfo", ["feName"], [info] )
END FUNCTION
```

Some front calls need a file path as parameter. File paths must follow the syntax of the front end workstation file system. You may need to escape backslash characters in such parameters. The next example shows how to pass a file path with a space in a directory name to a front-end running on a Microsoft™ Windows™ workstation:

```
FUNCTION call()
   DEFINE path STRING, res INTEGER
   LET path = "\"c:\\work dir\\my report.doc\""
   -- This is: "c:\work dir\my report.doc"
   CALL ui.Interface.frontCall( "standard", "shellExec", [path], [res] )
END FUNCTION
```

Front call error handling

Exception handling instructions can be used to check the execution status of a front call. Both `WHENEVER ERROR` directives or `TRY/CATCH` block can surround the front call to avoid program stop in case of error, and check the error number returned in the STATUS variable.

> **Note:** There is not need to surround front calls with exception handlers such as `TRY/CATCH`, if the front call is always supposed to execute without error. For example, the feInfo front call will never produce an exception.

Example of front call error handling with a `TRY/CATCH` block:

```
FUNCTION takePhoto()
```

```
   DEFINE path STRING
   TRY -- This front call may fail if the front-end is not a mobile device:
       CALL ui.Interface.frontCall( "mobile", "takePhoto", [], [path] )
   CATCH
       MESSAGE "Cannot take photo: ", STATUS, " ", err_get(STATUS)
       LET path = NULL
   END TRY
   RETURN path
END FUNCTION
```

If the front call module name or the function name is invalid, the errors -6331 or -6332 will be raised, respectively.

If the front call execution failed for some reason, the error -6333 will be raised. The description of the problem can be found in the second part of the error message, returned by a call to the ERR_GET() function.

The error -6334 can be raised in case of input or output parameter mismatch. The control of the number of input and output parameters is in the hands of the front-end. Most of the standard front calls have optional returning parameters and will not raise error -6334, if the output parameter list is left empty. However, front-end specific extensions or user-defined front-end functions may return an invalid execution status in case of input or output parameter mismatch, raising error -6334. If the front-end sends an call execution status of zero (OK), and the number of returned values does not match the number of program variables, the runtime system will set unmatched program variables to `NULL`. As a general rule, the program should provide the expected input and output parameters as specified in the documentation.

ui.Interface.filenameToURI
Converts a file name to an URI to be used as a web component image resource.

Syntax

```
ui.Interface.filenameToURI(
   filename STRING )
  RETURNING uri INTEGER
```

1. *filename* is the local file name to be converted to a URI.
2. *uri* is the resulting URI.

Usage

The `ui.Interface.filenameToURI()` class method converts a local (VM context / server) file name to an URI that can be accessed by the front ends to get the resource. This method is typically used to provide application image files in Web Components.

> **Note:** The runtime system uses the same mechanism to provide the front-end with images referenced in form elements: Thus, there is no need to call this method except when using application images in web components.

This method is typically used when executing applications behind a GAS, but it can also be used with direct connection to the front-end (typical GDC desktop connection), or when running apps on a mobile device.

The VM context file name to URI mapping is done as follows:

- If the *filename* parameter is already an URI (i.e. has a scheme like `http:`, `https:`, `file:`), the file name is returned as is.
- If the *filename* parameter is an absolute, relative file path, or a simple file name:
  - When the program is executing behind a GAS, user agents can access files via HTTP. In this architecture, the method will produce an URI that can be referenced in HTML elements of a web component: The image resource will be available from this location.

- When using a direct connection to the (GDC) front-end without using the GAS, the method returns the file name as is, and the image resources will be transmitted to the GDC through the FGLIMAGEPATH mechanism.
- When executing an app on a mobile device, both front-end and runtime system cohexist on the same platform and can access to the same file system. In this architecture, the method builds the complete local path to the file, following the list of directories defined in the FGLIMAGEPATH environment variable.

  Note: The URI or file path returned by the `filenameToURI()` method are only valid during the program live time: Do not stores these values in a persistent way.

For more details, see Providing the image resource on page 784 and Using image resources with the gICAPI web component on page 1430

Example

```
LET uri = ui.Interface.filenameToURI("myimage.png")
```

ui.Interface.getChildCount
Get the number of children in a parent container.

Syntax

```
ui.Interface.getChildCount()
  RETURNING result INTEGER
```

Usage

The `ui.Interface.getChildCount()` class method returns the number of child programs attached to the current parent WCI program.

WCI child programs are attached to a given container by using the `ui.Interface.setContainer()` method. Container and child program identifiers/names are defined by the `ui.Interface.setName()` method.

ui.Interface.getChildInstances
Get the number of child instances for a given program name.

Syntax

```
ui.Interface.getChildInstances(
   name STRING )
  RETURNING result INTEGER
```

1. *name* is the name of a child program attached to the container of the current program.

Usage

The `ui.Interface.getChildInstances()` class method returnes the number of child instances of a program attached to the current parent WCI program, according to the name of the child program passed as parameter.

The name of a child program is defined by the `ui.Interface.setName()` method.

The `getChildInstances()` method is typically used to check if a give child program is already started, to avoid multiple instances of the same program in a given WCI container.

ui.Interface.getContainer
Get the parent container of the curren program.

Syntax

```
ui.Interface.getContainer()
  RETURNING result STRING
```

Usage

The `ui.Interface.getContainer()` class method returns the name of the parent WCI container defined with `ui.Interface.setContainer()`.

ui.Interface.getDocument
Returns the DOM document of the abstract user interface tree.

Syntax

```
ui.Interface.getDocument()
  RETURNING result  om.DomDocument
```

Usage

The `ui.Interface.getDocument()` method returns the DOM document of the abstract user interface tree.

Define a variable with the type `om.DomDocument` to receive the result of this method.

Consider using the `getRootNode()` method instead to get directely the root DOM node of the AUI tree.

ui.Interface.getFrontEndName
Returns the type of the front-end currently in use.

Syntax

```
ui.Interface.getFrontEndName()
  RETURNING result STRING
```

Usage

The `ui.Interface.getFrontEndName()` class method returns the type of the front end used by the program.

Table 380: Front-end names

Front-end name	Description
`GDC`	Desktop front-end
`GMA`	Mobile front-end for Android™
`GMI`	Mobile front-end for iOS
`GWC`	Web browser front-end
`Console`	Text front-end (dumb terminal)

ui.Interface.getFrontEndVersion
Returns the version of the front-end currently in use.

Syntax

```
ui.Interface.getFrontEndVersion()
  RETURNING result STRING
```

Usage

The `ui.Interface.getFrontEndVersion()` class method returns the version number of the front end used by the program.

> **Note:** This method is primarily used for debugging purposes.

ui.Interface.getImage
Returns the icon image of the program.

Syntax

```
ui.Interface.getImage()
  RETURNING result STRING
```

Usage

Use the `ui.Interface.getImage()` class method to get the icon image name of the program previously set by `setImage()`.

ui.Interface.getName
Returns the name of the program.

Syntax

```
ui.Interface.getName()
  RETURNING result STRING
```

Usage

The `ui.Interface.getName()` class method returns the name of the program that was defined with the `setName()` method.

ui.Interface.getRootNode
Get the root DOM node of the abstract user interface.

Syntax

```
ui.Interface.getRootNode()
  RETURNING result om.DomNode
```

Usage

The `ui.Interface.getRootNode()` method returns the root DOM node of the abstract user interface tree.

Define a variable with the type `om.DomNode` to receive the result of this method.

```
DEFINE rn om.DomNode
```

```
LET rn = ui.Interface.getRootNode()
-- use d to inspect/change the AUI tree
```

ui.Interface.getText
Returns the title of the program.

Syntax

```
ui.Interface.getText()
  RETURNING result STRING
```

Usage

Use the `ui.Interface.getText()` class method to get the title of the program previously set by `setText()`.

ui.Interface.getType
Returns the type of the program.

Syntax

```
ui.Interface.getType()
  RETURNING result STRING
```

Usage

Use the `ui.Interface.getType()` class method to get the type of the program previously set by `setType()`.

ui.Interface.loadActionDefaults
Load the default action defaults file.

Syntax

```
ui.Interface.loadActionDefaults(
   filename STRING )
```

1. *filename* is the name of action defaults file, without the extension.

Usage

Use the `ui.Interface.loadActionDefaults()` class method to load a .4ad file defining action defaults for all program forms.

Specify the filename without the "4ad" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH/FGLRESOURCEPATH environment variable.

Example

```
CALL ui.Interface.loadActionDefaults("mydefaults")
```

ui.Interface.loadStartMenu
Load the start menu file.

Syntax

```
ui.Interface.loadStartMenu(
   filename STRING )
```

1. *filename* is the name of a start menu file, without the extension.

Usage

Use the `ui.Interface.loadStartMenu()` class method to load a .4sm file defining a start menu.

Specify the filename without the "`4sm`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH/FGLRESOURCEPATH environment variable.

Example

```
CALL ui.Interface.loadStartMenu("mystartmenu")
```

ui.Interface.loadStyles
Load the presentation styles file.

Syntax

```
ui.Interface.loadStyles(
   filename STRING )
```

1. *filename* is the name of presentation styles file, without the extension.

Usage

Use the `ui.Interface.loadStyles()` class method to load a .4st file defining presentation styles for all program forms.

Specify the filename without the "`4st`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH/FGLRESOURCEPATH environment variable.

Example

```
CALL ui.Interface.loadStyles("mystyles")
```

ui.Interface.loadToolBar
Load a default toolbar file.

Syntax

```
ui.Interface.loadToolBar(
   filename STRING )
```

1. *filename* is the name of a toolbar file, without the extension.

Usage

Use the `ui.Interface.loadToolBar()` class method to load a .4tb file defining a default global toolbar for all forms.

Specify the filename without the "`4tb`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH/FGLRESOURCEPATH environment variable.

The default toolbar loaded by this method is also used for the WCI container.

Example

```
CALL ui.Interface.loadToolBar("mytoolbar")
```

ui.Interface.loadTopMenu
Load a default topmenu file.

Syntax

```
ui.Interface.loadTopMenu(
   filename STRING )
```

1. *filename* is the name of a topmenu file, without the extension.

Usage

Use the `ui.Interface.loadTopMenu()` class method to load a .4tm file defining a default topmenu for all forms.

Specify the filename without the "`4tm`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH/FGLRESOURCEPATH environment variable.

The default topmenu loaded by this method is also used for the WCI container.

Example

```
CALL ui.Interface.loadTopMenu("mytopmenu")
```

ui.Interface.setContainer
Define the parent container for the current program.

Syntax

```
ui.Interface.setContainer(
   name STRING )
```

1. *name* is the name of the parent container.

Usage

The `ui.Interface.setContainer(name)` class method to specify the name of the parent WCI container where the current program windows must be displayed. This creates a WCI relation between two independent programs running with distinct fglrun processes.

Each WCI program must be identified by a name, to be set with the `ui.Interface.setName()` class method.

ui.Interface.setImage
Defines the icon image of the program.

Syntax

```
ui.Interface.setImage(
   icon STRING )
```

1. *icon* is the image file name to be used as program icon.

Usage

Use the `ui.Interface.setImage()` class method to define the icon image for the program to be used by the front-ends.This icon will be used in task bars, for example.

Call the method at the beginning of the program, before any interactive instruction.

ui.Interface.setName
Define the name of the current program for the front-end.

Syntax

```
ui.Interface.setName(
   name STRING )
```

1. *name* is the identifier of the program.

Usage

Use the `ui.Interface.setName()` class method to define the identifier for the program to be used by the front-ends, for example in case of window container usage.

The name passed to this method will be passed to the front-end in order to identify the program.

Call the method at the beginning of the program, before any interactive instruction.

By default, it is the program name (without .42m or .42r extension).

ui.Interface.setSize
Specify the initial size of the parent container window.

Syntax

```
ui.Interface.setSize(
   height INTEGER,
   width INTEGER )
```

1. *height* is the initial height of the main window.
2. *width* is the initial width of the main window.

Usage

Use the `ui.Interface.setSize(height,width)` class method to define the initial size of the parent container window of an window container application. The parameters can be integer or string values.

By default, the unit is the character grid cells, but you can add the **px** unit to specify the height and width in pixels.

The `setSize()` method can also be used to configure the size of the main window when using traditional mode, as a replacement of `fgl_setsize()` built-in function.

Call the method at the beginning of the program, before any interactive instruction.

ui.Interface.setText
Defines the title for the program.

Syntax

```
ui.Interface.setText(
   title STRING )
```

1. *title* is the text to be used as program title.

Usage

Use the `ui.Interface.setText()` class method to define the title for the program to be used by the front-ends, for example in case of window container usage (as title for the main window), or for the text to be displayed in the task bars.

Call the method at the beginning of the program, before any interactive instruction.

ui.Interface.setType
Defines the type of the program for the front-end.

Syntax

```
ui.Interface.setType(
   type STRING )
```

1. *type* is the identifier of the program.

Usage

Use the `ui.Interface.setType()` class method to define the type for the program to be used by the front-ends, for example in case of window container usage.

Possible values are: `normal`, `container`, `child`.

The type passed to this method will be passed to the front-end in order to define the rendering and behavior of the program.

Call the method at the beginning of the program, before any interactive instruction.

ui.Interface.refresh
Synchronize the user interface with the front-end.

Syntax

```
ui.Interface.refresh()
```

Usage

The `ui.Interface.refresh()` class method forces a synchronization of the abstract user interface tree with front-end .

By default, during an interactive instruction like `DIALOG`, the AUI tree is refreshed automatically when the runtime system gets the control back after user code execution. There is no need to call the refresh method in regular code.

> **Important:** This method should be used with care; It is only provided to synchronize with the front-end in specific cases. For example, when you need to display batch processing information to the user. Calling this method too frequently will produce a lot of network traffic.

Example

```
FOR i=1 TO 10
   DISPLAY i TO step_num
   CALL ui.Interface.refresh()
   SLEEP 1
END FOR
```

Examples

Example 1: Get the type and version of the front end

```
MAIN
  MENU "Test"
    COMMAND "Get"
      DISPLAY "Name = " || ui.Interface.getFrontEndName()
      DISPLAY "Version = " || ui.Interface.getFrontEndVersion()
    COMMAND "Exit"
      EXIT MENU
  END MENU
END MAIN
```

Example 2: Get the AUI root node and save it to a file in XML format

```
MAIN
  DEFINE n om.DomNode
  MENU "Test"
    COMMAND "SaveUI"
      LET n = ui.Interface.getRootNode()
      CALL n.writeXml("auitree.xml")
    COMMAND "Exit"
      EXIT MENU
  END MENU
END MAIN
```

Example 3: Using the Window Container Interface

The WCI parent program:

```
MAIN
  CALL ui.Interface.setName("main1")
  CALL ui.Interface.setText("This is the parent container")
  CALL ui.Interface.setType("container")
  CALL ui.Interface.setSize("600px","600px")
  CALL ui.Interface.loadStartMenu("appmenu")
  MENU "Main"
    COMMAND "Help" CALL help()
    COMMAND "About" CALL aboutbox()
    COMMAND "Exit"
      IF ui.Interface.getChildCount()>0 THEN
        ERROR "You must first exit the child programs."
      ELSE
        EXIT MENU
      END IF
  END MENU
END MAIN
```

The WCI child program:

```
MAIN
  CALL ui.Interface.setName("prog1")
  CALL ui.Interface.setText("This is module 1")
```

```
  CALL ui.Interface.setType("child")
  CALL ui.Interface.setContainer("main1")
  MENU "Test"
    COMMAND "Exit"
      EXIT MENU
  END MENU
END MAIN
```

Example 4: Synchronizing the AUI tree with the front end

```
MAIN
  DEFINE cnt INTEGER
  OPEN WINDOW w WITH FORM "myform"
  FOR cnt=1 TO 10
    DISPLAY BY NAME cnt
    CALL ui.Interface.refresh()
    SLEEP 1
  END FOR
END MAIN
```

The Window class
The `ui.Window` class provides an interface to the window objects create with the `OPEN WINDOW` instruction.

A windows is typically created with a form with the `OPEN WINDOW WITH FORM` instruction. If the window contains a form, consider using the `ui.Form` class instead of `ui.Window`.

ui.Window methods
Methods of the `ui.Window` class.

Table 381: Class methods

Name	Description
`ui.Window.forName( name STRING ) RETURNING result ui.Window`	Get a window object by name.
`ui.Window.getCurrent() RETURNING result ui.Window`	Get the current window object.

Table 382: Object methods

Name	Description
`createForm( name STRING ) RETURNING result ui.Form`	Create a new empty form in a window.
`findNode( type STRING, name STRING ) RETURNING result om.DomNode`	Search for a specific element in the window.
`getForm()`	Get the current form of a window.

Name	Description
RETURNING *result* ui.Form	
getNode() RETURNING *result* om.DomNode	Get the DOM node of a window.
getImage() RETURNING *result* STRING	Get the window icon.
getText() RETURNING *result* STRING	Get the window title.
setImage(*image* STRING)	Set the window icon.
setText(*text* STRING)	Set the window title.

ui.Window.forName
Get a window object by name.

Syntax

```
ui.Window.forName(
   name STRING )
  RETURNING result ui.Window
```

1. *name* defines the name of the window.

Usage

The `ui.Window.forName()` class method returns the `ui.Window` object corresponding to an identifier used to create the window with the `OPEN WINDOW` instruction.

Declare a variable of type `ui.Window` to hold the window object reference.

Example

```
DEFINE w ui.Window
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.forName("w1")
```

ui.Window.createForm
Create a new empty form in a window.

Syntax

```
createForm(
   name STRING )
  RETURNING result ui.Form
```

1. *name* is the name for the form.

Usage

The `createForm()` method can be used to create a new empty form in the window object. This is typically used to build forms dynamically, by creating the elements with the OM API.

> **Important:** It is mandatory to create a form in a window with the `createForm()` method, otherwise it is not usable.

The method returns a new `ui.Form` instance or `NULL` if the form name passed as the parameter identifies an existing form used by the window.

Example

```
DEFINE w ui.Window,
       f ui.Form,
       n, g om.DomNode
OPEN WINDOW w1 WITH 10 ROWS, 20 COLUMNS
LET w = ui.Window.getCurrent()
LET f = w.createForm("myform")
LET n = f.getNode()
LET g = f.createChild("Grid")
```

ui.Window.getCurrent
Get the current window object.

Syntax

```
ui.Window.getCurrent()
  RETURNING result ui.Window
```

Usage

The `ui.Window.getCurrent()` class method returns the `ui.Window` object corresponding to the current window.

Declare a variable of type `ui.Window` to hold the window object reference.

Example

```
DEFINE w ui.Window
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.getCurrent()
```

ui.Window.getForm
Get the current form of a window.

Syntax

```
getForm()
  RETURNING result ui.Form
```

Usage

The `getForm()` method returns the `ui.Form` object corresponding to the current form used by the window object.

Declare a variable of type `ui.Form` to hold the form object reference.

Consider using the `ui.Dialog.getForm()` method to get the form used by the current dialog.

Example

```
DEFINE f ui.Form
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.getCurrent()
LET f = w.getForm()
```

ui.Window.getNode
Get the DOM node of a window.

Syntax

```
getNode()
  RETURNING result om.DomNode
```

Usage

The `getNode()` method returns the `om.DomNode` object corresponding to the window object.

Declare a variable of type `om.DomNode` to hold the DOM node object reference.

Consider using the `ui.Dialog.getForm()` method to get the form used by the current dialog.

Example

```
DEFINE w ui.Window, n om.DomNode
OPEN WINDOW w1 WITH FORM "custform"
LET w = ui.Window.getCurrent()
LET n = w.getNode()
```

ui.Window.findNode
Search for a specific element in the window.

Syntax

```
findNode(
   type STRING,
   name STRING )
  RETURNING result om.DomNode
```

1. *type* defines the type of the node.
2. *name* defines the name of the node.

Usage

The `findNode()` method allows you to search for a specific DOM node in the abstract representation of the window. You search for a child node by giving its type and the name of the element (i.e. the tagname and the value of the 'name' attribute).

The method returns the first element found matching the specified type (tagname) and node name. Window element names must be unique for the same type of nodes, if you want to distinguish all elements.

The `findNode()` method is provided for `ui.Window` class for specific cases when the window does not contain a form. For windows containing a form, use the `ui.Form.findNode()` method instead.

ui.Window.getImage
Get the window icon.

Syntax

```
getImage()
  RETURNING result STRING
```

Usage

Use the `getImage()` method to get the current icon of a window.

ui.Window.getText
Get the window title.

Syntax

```
getText()
  RETURNING result STRING
```

Usage

Use the `getText()` method to get the current title of a window.

ui.Window.setImage
Set the window icon.

Syntax

```
setImage(
   image STRING )
```

1. *image* is the image name for the icon of the window.

Usage

The `setImage()` method defines the icon of the window.

By default, the icon of a window is defined by the `IMAGE` attribute of the `LAYOUT` definition in form files.

ui.Window.setText
Set the window title.

Syntax

```
setText(
   text STRING )
```

1. *text* is the title of the window.

Usage

The `setText()` method defines the title of the window.

By default, the title of a window is defined by the `TEXT` attribute of the `LAYOUT` definition in form files.

Examples
Example 1: Get a window by name and change the title

```
MAIN
  DEFINE w ui.Window
  OPEN WINDOW w1 WITH FORM "customer" ATTRIBUTES(TEXT="Unknown")
  LET w = ui.Window.forName("w1")
  IF w IS NULL THEN
    EXIT PROGRAM
  END IF
  CALL w.setText("Customer")
  MENU "Test"
     COMMAND "exit" EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

Example 2: Get a the current form and hide a groupbox

```
MAIN
  DEFINE w ui.Window
  DEFINE f ui.Form
  OPEN WINDOW w1 WITH FORM "customer"
  LET w = ui.Window.getCurrent()
  IF w IS NULL THEN
    EXIT PROGRAM
  END IF
  LET f = w.getForm()
  MENU "Test"
    COMMAND "hide" CALL f.setElementHidden("gb1",1)
    COMMAND "exit" EXIT MENU
  END MENU
  CLOSE WINDOW w1
END MAIN
```

The Form class
The `ui.Form` class provides an interface to form objects created by an `OPEN WINDOW WITH FORM` or `DISPLAY FORM` instruction.

A form object allows you to manipulate form elements by program. For example, you can hide parts of a form with the `setElementHidden()` method. The runtime system is able to handle hidden fields during a dialog instruction. You can, for example, hide a `GROUP` containing fields and labels.

Outside dialogs, get a `ui.Form` instance of the current form with the `ui.Window.getForm()` method. When executing a dialog, use the `ui.Dialog.getForm()` method.

Note that the `OPEN FORM` instruction does not load a form; it simply declares a handle. The form will be created in the AUI tree when executing the `DISPLAY FORM` instruction. Therefore, the corresponding `ui.Form` object is only available after `DISPLAY FORM` is executed.

ui.Form methods
Methods of the `ui.Form` class.

Table 383: Class methods

Name	Description
`ui.Form.setDefaultInitializer(` `  funcname STRING )`	Define the default initializer for all forms.

Table 384: Object methods

Name	Description
`findNode( type STRING, name STRING ) RETURNING result om.DomNode`	Search for a child node in the form.
`getNode() RETURNING result om.DomNode`	Get the DOM node of the form.
`loadActionDefaults( filename STRING )`	Load form action defaults.
`loadToolBar( filename STRING )`	Load the form toolbar.
`loadTopMenu( filename STRING )`	Load the form topmenu.
`setElementHidden( name STRING, hide INTEGER )`	Show or hide form elements.
`setElementImage( name STRING, text STRING )`	Change the image of form elements.
`setElementStyle( name STRING, style STRING )`	Change the style of form elements.
`setElementText( name STRING, text STRING )`	Change the text of form elements.
`setFieldHidden( name STRING, hide INTEGER )`	Show or hide a form field.
`setFieldStyle( name STRING, style STRING )`	Change the style of a form field.
`ensureElementVisible(`	Ensure the visibility of a form element.

Name	Description
name STRING)	
ensureFieldVisible(*name* STRING)	Ensure visibility of a form field.

ui.Form.setDefaultInitializer
Define the default initializer for all forms.

Syntax

```
ui.Form.setDefaultInitializer(
   funcname STRING )
```

1. *funcname* is the name of a function in the program.

Usage

Specify a default initialization function with the `ui.Form.setDefaultInitializer()` method, to implement global processing when a form is opened.

The method takes the name of the initialization function as a parameter.

> **Important:** The initialization function name must be in lowercase letters. The language syntax allows case-insensitive functions names, but the runtime system must reference functions in lowercase letters internally.

When a form is loaded with `OPEN FORM / DISPLAY FORM` or with `OPEN WINDOW ... WITH FORM`, the initialization function will be called with a `ui.Form` object as a parameter.

Example

```
MAIN
    ...
    CALL ui.Form.setDefaultInitializer("form_init")
    ...
    OPEN FORM f1 FROM "customers"
    DISPLAY FORM f1 -- initialization function is called
    ...
END MAIN

FUNCTION form_init(f)
    DEFINE f ui.Form
    CALL f.loadToolBar("common_toolbar")
END FUNCTION
```

ui.Form.ensureElementVisible
Ensure the visibility of a form element.

Syntax

```
ensureElementVisible(
   name STRING )
```

1. *name* defines the name of the form element.

Usage

Use the `ensureElementVisible()` method to make sure that the given form element (not form field) is visible to the user. This method can for example be used to show a folder page by passing a field that is located in the folder page, even if the field is not used in a dialog.

This method must be used for static form elements, to make form fields visible, use the `ensureFieldVisible()` method instead.

The form element is identified by its name. If several form elements can have the same name, the first form element found is selected.

Note that the `ensureElementVisible()` method can only show the specified element if it's possible, according to the focus handling in the current active dialog. For more details, see the `ensureFieldVisible()` instead.
ui.Form.ensureFieldVisible
Ensure visibility of a form field.

Syntax

```
ensureFieldVisible(
   name STRING )
```

1. *name* defines the name of the form field.

Usage

The `ensureFieldVisible()` method makes the given form field visible to the user. This method can for example be used to show a folder page by passing a field that is located in the folder page, even if the field is not used in a dialog.

The form field is identified by name, with an optional prefix (`table.column` or `column`).

This method does not give the focus to the field passed as parameter: The folder page or screen area shown by this method call is temporarily visible and can disappear at the next user interaction, according to focus management.

For example, consider a folder having two pages. The focus is in a field on the first page. A call to the `ensureFieldVisible()` method makes the second folder page visible, passing a field located in the second page. When the user presses the TAB key, the focus goes to the next field on the first page, bringing the first page to the top. If you want to show a folder page and give the focus to a specific field in that page, you must explicitly give the focus to a field of the page, with `NEXT FIELD`.

The `ensureFieldVisible()` method is used for form fields, to show static form elements such as labels or images, use the `ensureElementVisible()` method instead.

ui.Form.getNode
Get the DOM node of the form.

Syntax

```
getNode()
  RETURNING result om.DomNode
```

Usage

The `getNode()` method returns the DOM node containing the abstract representation of the window/form.

After loading and displaying a form with `OPEN FORM / DISPLAY FORM` or with `OPEN WINDOW ... WITH FORM`, get the form object for example with `ui.Dialog.getForm()` and use the `getNode()` method to query the DOM node corresponding to the form.

Example

```
DEFINE n om.DomNode,
       f ui.Form

INPUT BY NAME ...
   ...
   LET f = DIALOG.getForm()
   LET n = f.getNode()
   ...
```

ui.Form.findNode
Search for a child node in the form.

Syntax

```
findNode(
   type STRING,
   name STRING )
  RETURNING result om.DomNode
```

1. *type* defines the type of the node.
2. *name* defines the name of the node.

Usage

The `findNode()` method allows you to search for a specific DOM node in the abstract representation of the form. You search for a child node by giving its type and the name of the element (i.e. the tagname and the value of the 'name' attribute).

The method returns the first element found matching the specified type (tagname) and node name. Form element names must be unique for the same type of nodes, if you want to distinguish all elements.

Example

```
DEFINE n om.DomNode,
       f ui.Form

INPUT BY NAME ...
   ...
   LET f = DIALOG.getForm()
   LET n = f.findNode("Label", "lb_name")
   ...
```

ui.Form.loadActionDefaults
Load form action defaults.

Syntax

```
loadActionDefaults(
   filename STRING )
```

1. *filename* is the name of the action defaults file without extension.

Usage

Load form specific action defaults at runtime with the `loadActionDefaults()` method.

The `loadActionDefaults()` method is commonly used in the form initialization function.

Specify the filename without the "`4ad`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH / FGLRESOURCEPATH environment variable.

If a form contains already action defaults, it will be replaced by the new action defaults loaded by this method.

ui.Form.loadToolBar
Load the form toolbar.

Syntax

```
loadToolBar(
   filename STRING )
```

1. *filename* is the name of the toolbar file without extension.

Usage

Load a toolbar XML definition file into the form with the `loadToolBar()` method.

The `loadToolBar()` method is commonly used in the form initialization function.

Specify the filename without the "`4tb`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH / FGLRESOURCEPATH environment variable.

If the form already contains a toolbar, it will be replaced by the new toolbar loaded from this method.

ui.Form.loadTopMenu
Load the form topmenu.

Syntax

```
loadTopMenu(
   filename STRING )
```

1. *filename* is the name of the topmenu file without extension.

Usage

Load a topmenu XML definition file into the form with the `loadTopMenu()` method.

The `loadTopMenu()` method is commonly used in the form initialization function.

Specify the filename without the "`4tm`" extension.

If the file does not exist in the current directory, it is searched in the directories defined in the DBPATH / FGLRESOURCEPATH environment variable.

If the form already contains a topmenu, it will be replaced by the new topmenu loaded by this method.

ui.Form.setElementHidden
Show or hide form elements.

Syntax

```
setElementHidden(
   name STRING,
```

```
   hide INTEGER )
```

1. *name* defines the name of the node.
2. *hide* the integer value to show or hide the element.

Usage

Change the visibility of a form element with the `setElementHidden()` method. You must pass the identifier of the form element. The identifier is the element name as defined in the form definition.

All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

The `setElementHidden()` method changes the hidden attribute of all form elements identified by the name.

The value passed to hide/show the element can be 0, 1 or 2:

Table 385: Hidden attribute integer values

Hidden value	Description
`0`	Makes the element visible.
`1`	The element is hidden and the user cannot make it visible. Typically used to hide information the user is not allowed to see.
`2`	The element is hidden and the user can make it visible.

> **Note:** Do not hide all fields of a dialog, otherwise the dialog execution stops. At least one field must get the focus during a dialog execution.

ui.Form.setElementImage
Change the image of form elements.

Syntax

```
setElementImage(
   name STRING,
   text STRING )
```

1. *name* defines the name of the node.
2. *image* is the image to be set.

Usage

Change the image/icon of a form element with the `setElementImage()` method. You must pass the identifier of the form element. The identifier is the element name as defined in the form definition.

All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

ui.Form.setElementStyle
Change the style of form elements.

Syntax

```
setElementStyle(
   name STRING,
   style STRING )
```

1. *name* defines the name of the node.
2. *style* is the style name to be set.

Usage

Change the style of a form element with the `setElementStyle()` method. You must pass the identifier of the form element. The identifier is the element name as defined in the form definition.

All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

ui.Form.setElementText
Change the text of form elements.

Syntax

```
setElementText(
   name STRING,
   text STRING )
```

1. *name* defines the name of the node.
2. *text* is the text to be set.

Usage

Change the text of a form element with the `setElementText()` method, for example to modify the text of a static label or group box during program execution. You must pass the identifier of the form element. The identifier is the element name as defined in the form definition file (per) or the name attribute for the element as defined in the form file.

All elements with this name will be affected. If you want to distinguish all form elements, use unique names in the form definition file.

ui.Form.setFieldHidden
Show or hide a form field.

Syntax

```
setFieldHidden(
   name STRING,
   hide INTEGER )
```

1. *name* defines the name of the form field.
2. *hide* the integer value to show or hide the element.

Usage

Change the visibility of a form field with the `setFieldHidden()` method. You must pass the identifier of the form field, as defined in the .per form definition. The form field is identified by column name, with an optional prefix (`table.column` or `column`). The form field can be a regular field or a column of a list container such as a TABLE.

The value passed to hide/show the element can be 0, 1 or 2:

Table 386: Hidden attribute integer values

Hidden value	Description
`0`	Makes the field visible.

Hidden value	Description
1	The field is hidden and the user cannot make it visible. Typically used to hide information the user is not allowed to see.
2	The element is hidden and the user can make it visible.

Note: Do not hide all fields of a dialog, otherwise the dialog execution stops. At least one field must get the focus during a dialog execution.

ui.Form.setFieldStyle
Change the style of a form field.

Syntax

```
setFieldStyle(
   name STRING,
   style STRING )
```

1. *name* defines the name of the form field.
2. *style* is the style name to be set.

Usage

Change the style of a form field with the `setFieldStyle()` method. You must pass the identifier of the form field, as defined in the .per form definition. The form field is identified by column name, with an optional prefix (`table.column` or `column`). The form field can be a regular field or a column of a list container such as a `TABLE`.

Examples

Example 1: Implement a global form initialization function

```
MAIN
  CALL ui.Form.setDefaultInitializer("init")
  OPEN FORM f1 FROM "items"
  DISPLAY FORM f1 -- Form appears in the default SCREEN window
  OPEN WINDOW w1 WITH FORM "customer"
  OPEN WINDOW w2 WITH FORM "orders"
  DISPLAY FORM f1 -- Form appears in w2 window
  MENU "Test"
    COMMAND "exit" EXIT MENU
  END MENU
END MAIN

FUNCTION init(f)
  DEFINE f ui.Form
  DEFINE n om.DomNode
  CALL f.loadTopMenu("mymenu")
  LET n = f.getNode()
  DISPLAY "Init: ", n.getAttribute("name")
END FUNCTION
```

Example 2: Hide form elements dynamically

```
MAIN
  DEFINE w ui.Window
  DEFINE f ui.Form
  DEFINE custid INTEGER
```

```
  DEFINE custname CHAR(10)
  OPEN WINDOW w1 WITH FORM "customer"
  LET w = ui.Window.getCurrent()
  LET f = w.getForm()
  INPUT BY NAME custid, custname
    ON ACTION hide
      CALL f.setFieldHidden("customer.custid",1)
      CALL f.setElementHidden("label_custid",1)
    ON ACTION show
      CALL f.setFieldHidden("customer.custid",0)
      CALL f.setElementHidden("label_custid",0)
  END INPUT
END MAIN
```

Example 3: Change the title of table column headers

The form file (coltitle.per):

```
LAYOUT
GRID
{
<TABLE t1             >
 Id      Name
[c1     |c2           ]
[c1     |c2           ]
[c1     |c2           ]
[c1     |c2           ]
[c1     |c2           ]
}
END
END
ATTRIBUTES
c1 = FORMONLY.col1;
c2 = FORMONLY.col2;
END
INSTRUCTIONS
SCREEN RECORD sr(FORMONLY.*);
END
```

The program file:

```
MAIN
    DEFINE f ui.Form, i INT
    DEFINE arr DYNAMIC ARRAY OF RECORD
                   id INT,
                   name VARCHAR(40)
               END RECORD
    OPEN FORM f1 FROM "coltitle"
    DISPLAY FORM f1
    FOR i=1 TO 10
        LET arr[i].id = i
        LET arr[i].name = "aaa"||i
    END FOR
    DISPLAY ARRAY arr TO sr.* ATTRIBUTES(UNBUFFERED)
        BEFORE DISPLAY
           let f = dialog.getForm()
        ON ACTION change_title
           CALL f.setElementText("formonly.col1","ID")
           CALL f.setElementText("formonly.col2","NAME")
    END DISPLAY
END MAIN
```

The Dialog class

The `ui.Dialog` class provides a set of methods to configure, query and control the current interactive instruction.

A `ui.Dialog` object can for example be used to enable or disable actions and form fields dynamically during the dialog execution.

A dialog object is typically available inside a dialog block, with the predefined `DIALOG` keyword, and can only be referenced during the execution of that interactive instruction. After the interactive instruction, the dialog object is destroyed and its reference becomes invalid.

Dialog objects can also be created dynamically to handle forms created at runtime. This feature is only provided for specific needs.

ui.Dialog methods

Methods of the `ui.Dialog` class.

Table 387: Class methods

Name	Description

`ui.Dialog.createConstructByName(`
`  fields DYNAMIC ARRAY OF RECORD`
`  name STRING,`
`  type STRING`
`  END RECORD`
`  )`
	Creates a new ui.Dialog object to handle a CONSTRUCT BY NAME.

`ui.Dialog.createDisplayArrayTo(`
`  fields DYNAMIC ARRAY OF RECORD`
`  name STRING,`
`  type STRING`
`  END RECORD,`
`  tabname STRING )`
	Creates a new ui.Dialog object to handle a DISPLAY ARRAY.

`ui.Dialog.createInputByName(`
`  fields DYNAMIC ARRAY OF RECORD`
`  name STRING,`
`  type STRING`
`  END RECORD`
`  )`
	Creates a new ui.Dialog object to handle an INPUT BY NAME.

`ui.Dialog.getCurrent()` `  RETURNING result ui.Dialog`	Returns the current dialog object.
`ui.Dialog.setDefaultUnbuffered(` `  value BOOLEAN )`	Set the default unbuffered mode for all dialogs.

Table 388: Object methods

Name	Description
`accept()`	Validates and terminates the dialog.
`addTrigger(` `  name STRING )`	Adds an event trigger to the dynamic dialog

`appendNode(`
`  name STRING,`
`  index INTEGER )`
	Appends a new node in the specified tree-view.

`appendRow(` `  name STRING )`	Appends a new row in the specified list.
`deleteAllRows(` `  name STRING )`	Deletes all rows from the specified list.

`arrayToVisualIndex(`
`  name STRING,`
`  index INTEGER )`
	Converts the program array index to the visual index for a given screen array.

`deleteNode(`
`  name STRING,`
`  index INTEGER )`
	Deletes a node from the specified tree-view.

`deleteRow(`
`  name STRING,`
`  index INTEGER )`
	Deletes a row from the specified list.

`getArrayLength(`
`  name STRING )`
`  RETURNING result INTEGER`
	Returns the total number of rows in the specified list.

`getCurrentItem()` `  RETURNING result STRING`	Returns the current item having focus.

`getCurrentRow(`
`  name STRING )`
`  RETURNING result INTEGER`
	Returns the current row of the specified list.

`getFieldBuffer(` `  field STRING )`	Returns the input buffer of the specified field.

Name	Description
RETURNING *result* STRING	
getFieldTouched(*field-list* STRING) RETURNING *result* BOOLEAN	Returns the modification flag for a field.
getFieldValue(*name* STRING)	Returns the value of a field controlled by a dynamic dialog.
getForm() RETURNING *result* ui.Form	Returns the current form used by the dialog.
getSortKey(*screen-array* STRING) RETURNING *field-name* STRING	Returns the name of the sort field selected by the user.
getSortReverse(*screen-array* STRING) RETURNING *result* BOOLEAN	Indicates the sort order direction (FALSE=ascending, TRUE=descending)
insertNode(*name* STRING, *index* INTEGER)	Inserts a new node in the specified tree.
insertRow(*name* STRING, *index* INTEGER)	Inserts a new row in the specified list.
isRowSelected(*name* STRING, *index* INTEGER) RETURNING *result* BOOLEAN	Queries row selection for a give list and row.
nextField(*name* STRING)	Registering the next field to jump to.
nextEvent() RETURNING event STRING	Waits for a dialog event.
selectionToString(*name* STRING) RETURNING *result* STRING	Serializes data of the selected rows.
setActionActive(*name* STRING,	Enabling and disabling dialog actions.

Name	Description
active BOOLEAN)	
setActionHidden(*name* STRING, *hide* INTEGER)	Handling default action view visibility.
setArrayAttributes(*name* STRING, *attributes* ARRAY)	Define cell decoration attributes array for the specified list (singular or multiple dialogs).
setArrayLength(*name* STRING, *len* INTEGER)	Sets the total number of rows in the specified list.
setCellAttributes(*attributes* ARRAY)	Define cell decoration attributes array for the specified list (singular dialog only).
setCompleterItems(*items-array* DYNAMIC ARRAY OF STRING)	Define autocompletion items for the a field defined with COMPLETER attribute.
setCurrentRow(*name* STRING, *row* INTEGER)	Sets the current row in the specified list.
setFieldActive(*field-list* STRING, *active* BOOLEAN)	Enable and disable form fields.
setFieldTouched(*field-list* STRING, *touched* BOOLEAN)	Sets the modification flag of the specified field.
setFieldValue(*name* STRING, *value fgl-type*)	Sets the value of a field controlled by the dialog object.
setSelectionMode(*name* STRING, *mode* INTEGER)	Defines the row selection mode for the specified list.
setSelectionRange(*name* STRING, *start* INTEGER, *end* INTEGER,	Sets the row selection flags for a range of rows.

Name	Description
value BOOLEAN)	
validate(*field-list* STRING) RETURNING *result* INTEGER	Check form level validation rules.
visualToArrayIndex(*name* STRING, *index* INTEGER)	Converts the visual index to the program array index for a given screen array.

ui.Dialog.createConstructByName
Creates a new ui.Dialog object to handle a CONSTRUCT BY NAME.

Syntax

```
ui.Dialog.createConstructByName(
   fields DYNAMIC ARRAY OF RECORD
                          name STRING,
                          type STRING
                       END RECORD
   )
```

1. *fields* is the list of form fields controlled by the dialog. This must be a `DYNAMIC ARRAY` of `RECORD` structure, with a `name` and `type` member of type `STRING`.

Usage

The `ui.Dialog.createConstructByName()` class method creates a new dialog object to implement the equivalent of a static `CONSTRUCT` block.

> **Note:** The current form will be attached to the new created dialog.

The method takes a list of field definitions as parameter. The parameter must be defined as a `DYNAMIC ARRAY OF RECORD`, with `name` and `type` members:

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                          name STRING,
                          type STRING
                 END RECORD
```

These names provided in the field definition list must identify form fields defined in the current form. For example, if the current form file defines the following fields:

```
LAYOUT
...
END
TABLES
customer
END
ATTRIBUTES
EDIT f1 = customer.cust_id;
EDIT f2 = customer.cust_name;
...
END
```

The field names provided to the `createConstructByName()` method must be defined as follows:

```
LET fields[1].name = "customer.cust_id"
LET fields[2].name = "customer.cust_name"
...
```

The types provided in the field definition list will identify the data type to be used for data input and display.

Possible values for types are the string equivalents of the Genero BDL built-in types. For example:

- `"INTEGER"`
- `"VARCHAR(50)"`
- `"DATE"`
- `"DECIMAL(10,2)"`
- `"DATETIME YEAR TO FRACTION(5)"`

Note: The type used to define form fields can be the returning value of a `base.SqlHandle.getResultType()` method.

Example

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                        name STRING,
                        type STRING
              END RECORD
DEFINE d ui.Dialog

OPEN FORM f1 FROM "custform"
DISPLAY FORM f1

LET fields[1].name = "customer.cust_id"
LET fields[1].type = "INTEGER"

LET fields[2].name = "customer.cust_name"
LET fields[2].type = "VARCHAR(50)"
...
LET d = ui.Dialog.createConstructByName(fields)
...
```

ui.Dialog.createInputByName
Creates a new ui.Dialog object to handle an INPUT BY NAME.

Syntax

```
ui.Dialog.createInputByName(
   fields DYNAMIC ARRAY OF RECORD
                       name STRING,
                       type STRING
                  END RECORD
   )
```

1. *fields* is the list of form fields controlled by the dialog. This must be a `DYNAMIC ARRAY` of a `RECORD` structure, with a `name` and `type` member of type `STRING`.

Usage

The `ui.Dialog.createInputByName()` class method creates a new dialog object to implement the equivalent of a static `INPUT BY NAME` block.

Note: The current form will be attached to the new created dialog.

The method takes a list of field definitions as parameter. The parameter must be defined as a `DYNAMIC ARRAY OF RECORD`, with `name` and `type` members:

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                        name STRING,
                        type STRING
              END RECORD
```

These names provided in the field definition list must identify form fields defined in the current form. For example, if the current form file defines the following fields:

```
LAYOUT
...
END
TABLES
customer
END
ATTRIBUTES
EDIT f1 = customer.cust_id;
EDIT f2 = customer.cust_name;
...
END
```

The field names provided to the `createInputByName()` method must be defined as follows:

```
LET fields[1].name = "customer.cust_id"
LET fields[2].name = "customer.cust_name"
...
```

The types provided in the field definition list will identify the data type to be used for data input and display.

Possible values for types are the string equivalents of the Genero BDL built-in types, for example:

- `"INTEGER"`
- `"VARCHAR(50)"`
- `"DATE"`
- `"DECIMAL(10,2)"`
- `"DATETIME YEAR TO FRACTION(5)"`

Note: The type used to define form fields can be the returning value of a `base.SqlHandle.getResultType()` method.

Example

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                        name STRING,
                        type STRING
              END RECORD
DEFINE d ui.Dialog

OPEN FORM f1 FROM "custform"
DISPLAY FORM f1

LET fields[1].name = "customer.cust_id"
LET fields[1].type = "INTEGER"

LET fields[2].name = "customer.cust_name"
LET fields[2].type = "VARCHAR(50)"
...
```

```
LET d = ui.Dialog.createInputByName(fields)
...
```

ui.Dialog.createDisplayArrayTo
Creates a new ui.Dialog object to handle a DISPLAY ARRAY.

Syntax

```
ui.Dialog.createDisplayArrayTo(
   fields DYNAMIC ARRAY OF RECORD
                          name STRING,
                          type STRING
                       END RECORD,
   tabname STRING )
```

1. *fields* is the list of form fields controlled by the dialog. This must be a `DYNAMIC ARRAY` of a `RECORD` structure, with a `name` and `type` member of type `STRING`.
2. *tabname* is the name of the screen array (defined with the `SCREEN RECORD` instruction in form files).

Usage

The `ui.Dialog.createDisplayArrayTo()` class method creates a new dialog object to implement the equivalent of a static `DISPLAY ARRAY TO` block.

> **Note:** The current form will be attached to the new created dialog.

The method takes a list of field definitions as first parameter. This parameter must be defined as a `DYNAMIC ARRAY OF RECORD`, with `name` and `type` members:

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                          name STRING,
                          type STRING
                END RECORD
```

These names provided in the field definition list must identify form fields defined in the current form. For example, if the current form file defines the following fields:

```
LAYOUT
...
END
TABLES
customer
END
ATTRIBUTES
EDIT f1 = customer.cust_id;
EDIT f2 = customer.cust_name;
...
END
```

The field names provided to the `createDisplayArrayTo()` method must be defined as follows:

```
LET fields[1].name = "customer.cust_id"
LET fields[2].name = "customer.cust_name"
...
```

The types provided in the field definition list will identify the data type to be used for data input and display.

Possible values for types are the string equivalents of the Genero BDL built-in types, for example:

- `"INTEGER"`

- "VARCHAR(50)"
- "DATE"
- "DECIMAL(10,2)"
- "DATETIME YEAR TO FRACTION(5)"

Note: The type used to define form fields can be the returning value of a `base.SqlHandle.getResultType()` method.

The second parameter passed to the `createDisplayArrayTo()` method is the name of the screen record which groups the fields together, for the list view of the form. For example, in the next form definition, the screen record name is "`sr_custlist`":

```
...
INSTRUCTIONS
SCREEN RECORD sr_custlist
(
 customer.cust_id,
 customer.cust_name,
 ...
);
END
```

For more details, see Screen records on page 866.

Example

```
DEFINE fields DYNAMIC ARRAY OF RECORD
                        name STRING,
                        type STRING
              END RECORD
DEFINE d ui.Dialog

OPEN FORM f1 FROM "custform"
DISPLAY FORM f1

LET fields[1].name = "customer.cust_id"
LET fields[1].type = "INTEGER"

LET fields[2].name = "customer.cust_name"
LET fields[2].type = "VARCHAR(50)"
...
LET d = ui.Dialog.createDisplayArrayTo(fields, "sr_custlist")
...
```

ui.Dialog.getCurrent
Returns the current dialog object.

Syntax

```
ui.Dialog.getCurrent()
  RETURNING result ui.Dialog
```

Usage

To get the current active dialog object, use the `ui.Dialog.getCurrent()` class method.

The method returns NULL if there is no current active dialog.

Example

```
FUNCTION field_disable(name)
  DEFINE name STRING
  DEFINE d ui.Dialog
  LET d = ui.Dialog.getCurrent()
  IF d IS NOT NULL THEN
    CALL d.setFieldActive(name, FALSE)
  END IF
END FUNCTION
```

ui.Dialog.setDefaultUnbuffered
Set the default unbuffered mode for all dialogs.

Syntax

```
ui.Dialog.setDefaultUnbuffered(
   value BOOLEAN )
```

1. *value* is a boolean to enable the unbuffered mode.

Usage

By default, modal dialogs are not sensitive to variable changes. To make a dialog sensitive, use the `UNBUFFERED` attribute in the dialog instruction definition.

To defined the default for all subsequent dialogs, use the `setDefaultUnbuffered()` class method:

```
CALL ui.Dialog.setDefaultUnbuffered(TRUE)
```

> **Note:** Only singular and multiple dialogs are sensitive to this API, parallel dialogs are implicitely using the unbuffered mode.

ui.Dialog.accept
Validates and terminates the dialog.

Syntax

```
accept()
```

Usage

Use the `accept()` method to validate field input and terminate the dialog. This method is equivalent to the `ACCEPT INPUT` / `ACCEPT DISPLAY` / `ACCEPT DIALOG` instructions.

The method is provided as a 3GL alternative to the `ACCEPT` control instructions, for example to terminate the dialog in a function, outside the context of a dialog block, where control instructions cannot be used.

Typical dialog validation rules are performed when calling this method. See `ACCEPT DIALOG` for more details.

ui.Dialog.addTrigger
Adds an event trigger to the dynamic dialog

Syntax

```
addTrigger(
   name STRING )
```

1. *name* is the name of the dialog.

Usage

When implementing a dynamic dialog, the `addTrigger()` method must be used to register the triggers, that need to be handled with user code:

```
CALL d.addTrigger("ON ACTION print")
```

Registered dialog triggers are then typically managed in a `WHILE` loop using the nextEvent() method, to wait for dialog events.

The following triggers are accepted by the `addTrigger()` method:

> **Note:** More predefined triggers such as "`BEFORE ROW`" are existing for dynamic dialogs, but these triggers do not have to be added with the `addTrigger()` method, since they are implicit.

Table 389: User-defined triggers for dynamic dialogs

Trigger name	Description	Dialog block equivalent
`ON ACTION` *`action-name`*	Action handler for the action identified by *action-name*.	`ON ACTION` block
`ON APPEND`	Row addition action handler for a display array dynamic dialog.	`ON APPEND` block
`ON DELETE`	Row deletion action handler for a display array dynamic dialog.	`ON DELETE` block
`ON INSERT`	Row insertion action handler for a display array dynamic dialog.	`ON INSERT` block
`ON UPDATE`	Row modification action handler for a display array dynamic dialog.	`ON UPDATE` block

ui.Dialog.appendRow
Appends a new row in the specified list.

Syntax

```
appendRow(
   name STRING )
```

1. *name* is the screen array name.

Usage

The `appendRow()` method appends a row to the end of the array controlled by the dialog.

> **Important:** This method is designed to be used in an `ON ACTION` block. It must not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, or `AFTER DELETE`.

The method is similar to appending a new element to the program array, except the internal dialog registers are automatically updated (like the total number of rows returned by `getArrayLength()`). If the list is decorated with cell attributes, the program array defining the attributes will also be synchronized. If multi-row selection is enabled, selection flags of existing rows are kept. The new row is inserted at the end of the list with the selection flag set to zero.

> **Note:** The purpose of this method is to implement business logic required to modify the record list in the current dialog. It is typically used in a `DISPLAY ARRAY` dialog. Avoid using this method in `INPUT ARRAY`. To allow the end user to append, modify or delete rows in a `DISPLAY ARRAY`, use list modification interaction blocks.

After the method is called, a new row is created in the program array. You can assign values to the variables before the control goes back to the user. The `getArrayLength()` method will return the new row count.

The method does not set the current row and does not give the focus to the list; you need to call `setCurrentRow()` and execute `NEXT FIELD` to give the focus.

This method does not execute any `BEFORE ROW`, `BEFORE INSERT`, `AFTER INSERT` or `AFTER ROW` control blocks.

The `appendRow()` method does not create a temporary row as the implicit append action of `INPUT ARRAY`; The row is considered permanent once it is added.

Example

This example implements a user-defined action to append ten rows at the end of the list.

```
ON ACTION append_some_rows
  FOR i = 1 TO 10
    CALL DIALOG.appendRow("sa")
    LET r = DIALOG.getArrayLength("sa")
    LET p_items[r].item_quantity = 1.00
  END FOR
```

ui.Dialog.appendNode
Appends a new node in the specified tree-view.

Syntax

```
appendNode(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index of the parent node in the program array (starts at 1).

Usage

The `appendNode()` method adds a new node under a given parent, when the dialog controls a tree view.

This method must be used when modifying the array of a tree view during the execution of the dialog, for example when implementing a dynamic tree with `ON EXPAND` / `ON COLLAPSE` triggers. Before the execution of the dialog, you can fill the program array directly. This includes the context of `BEFORE DISPLAY` or `BEFORE DIALOG` control blocks.

When adding rows for a tree view, the id of the parent node and new node matters because that information is used to build the internal tree structure. When calling `appendNode()`, you pass the index of the parent node under which the new node will be appended. In the program array, the parent-id member of the new node will automatically be initialized with the value of the id of the parent node identifier by the index passed as parameter, then the internal tree structure is rebuilt.

If the parent index is zero, a new root node will be appended.

The method returns the index of the new inserted node.

In the program array, the parent-id member of the new node will automatically be initialized with the value of the id member of the parent node identified by the index.

```
DISPLAY ARRAY mytree TO sr.*
    ...
    ON EXPAND(id)
       CALL DIALOG.appendNode("sr", id)
       ...
    ...
```

ui.Dialog.arrayToVisualIndex
Converts the program array index to the visual index for a given screen array.

Syntax

```
arrayToVisualIndex(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index of the program array row.

Usage

When the end user sorts rows in a table, the program array index may differ from the visual row index.

Use this method to convert a program array row index (`arr_curr()`) to a row index as seen by the end user. For example, if you want to display a typical message with (*current-row* / *total-rows*), convert the current program array row to a visual row index before displaying the value:

```
MESSAGE SFMT( "Row: %1/%2",
   DIALOG.arrayToVisualIndex( "sr", DIALOG.getCurrentRow("sr") ),
   DIALOG.getArrayLength( "sr" )
 )
```

ui.Dialog.deleteAllRows
Deletes all rows from the specified list.

Syntax

```
deleteAllRows(
   name STRING )
```

1. *name* is the screen array name.

Usage

The `deleteAllRows()` method removes all the rows of a list driven by a `DISPLAY ARRAY` or `INPUT ARRAY`. This is equivalent to a `deleteRow()` call, but instead of deleting one particular row, it removes all rows of the specified list.

This method must not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, `AFTER DELETE`, it is designed to be used in an `ON ACTION` block.

After the method is called, all rows are deleted from the program array, and the `getArrayLength()` method will return zero.

The method takes the name of the screen-array as parameter.

If the `deleteAllRows()` method is called during an `INPUT ARRAY`, the dialog will automatically append a new temporary row if the focus is in the list, to let the user enter new data. When using `AUTO APPEND = FALSE` attribute, no temporary row will be created and the current row register will be automatically changed to make sure that it will not be greater than the total number of rows.

If `deleteAllRows()` method is called during an `INPUT ARRAY` or `DISPLAY ARRAY` that has the focus, the `BEFORE ROW` control block will be executed if you delete the current row. This is required to reset the internal state of the dialog.

If the list was decorated with cell attributes, the program array defining the attributes will be cleared. If multi-row selection is enabled, selection flags are cleared.

ui.Dialog.deleteNode
Deletes a node from the specified tree-view.

Syntax

```
deleteNode(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index of the node in the program array that has to be deleted (starts at 1).

Usage

The `deleteNode()` method is similar to `deleteRow()`, except that it has to be used when the dialog controls a tree view.

This method must be used when modifying the array of a tree view during the execution of the dialog, for example when implementing a dynamic tree with `ON EXPAND` / `ON COLLAPSE` triggers. Before the execution of the dialog, you can fill the program array directly. This includes the context of `BEFORE DISPLAY` or `BEFORE DIALOG` control blocks.

The main difference with `deleteRow()` is that `deleteNode()` will remove recursively all child nodes before removing the node identified by index.

If the index is zero, all root nodes will be deleted from the tree.

ui.Dialog.deleteRow
Deletes a row from the specified list.

Syntax

```
deleteRow(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index of the row to be deleted (starts a 1).

Usage

The `deleteRow()` method deletes the row in the array controlled by the dialog.

> **Important:** This method is designed to be used in an `ON ACTION` block. It must not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, or `AFTER DELETE`.

The method is similar to deleting an element to the program array, except that internal dialog registers are automatically updated (like the total number of rows returned by `getArrayLength()`). If the list is

decorated with cell attributes, the program array defining the attributes will also be synchronized. If multi-row selection is enabled, selection information is synchronized (i.e., selection flags are shifted up) for all rows after the deleted row.

> **Note:** The purpose of this method is to implement business logic required to modify the record list in the current dialog. It is typically used in a `DISPLAY ARRAY` dialog. Avoid using this method in `INPUT ARRAY`. To allow the end user to append, modify or delete rows in a `DISPLAY ARRAY`, use list modification interaction blocks.

After the method is called, the row no longer exists in the program array, and the `getArrayLength()` method will return the new row count.

If the `deleteRow()` method is called during an `INPUT ARRAY` that has the focus, control blocks such as `BEFORE ROW` and `BEFORE FIELD` will be executed, if you delete the current row. This is required to reset the internal state of the dialog. However, the method does not execute any `BEFORE ROW` or `AFTER ROW` control blocks in a `DISPLAY ARRAY` dialog.

If the `deleteRow()` method is called during an `INPUT ARRAY`, and if no more rows are in the list after the call, the dialog will automatically append a new temporary row if the focus is in the list, to let the user enter new data. When using `AUTO APPEND = FALSE` attribute, no temporary row will be created and the current row register will be automatically changed to make sure that it will not be greater than the total number of rows.

If you pass zero as row index, the method does nothing (if no rows are in the list, `getCurrentRow()` returns zero).

Example

This example implements a user-defined action to remove rows that have a specific property:

```
ON ACTION delete_invalid_rows
  FOR r = 1 TO DIALOG.getArrayLength("sa")
    IF NOT s_orders[t].is_valid THEN
      CALL DIALOG.deleteRow("sa",r)
      LET r = r - 1
    END IF
  END FOR
```

ui.Dialog.getArrayLength
Returns the total number of rows in the specified list.

Syntax

```
getArrayLength(
   name STRING )
  RETURNING result INTEGER
```

1. *name* is the screen array name.

Usage

The `getArrayLength()` method returns the total number of rows of an `INPUT ARRAY` or `DISPLAY ARRAY` list. The name of the screen array is passed as parameter to identify the list.

Example

```
DIALOG
  DISPLAY ARRAY custlist TO sa_custlist.*
    BEFORE ROW
```

```
      MESSAGE "Row count: " ||
  DIALOG.getArrayLength("sa_custlist")
    ...
  END DISPLAY
  INPUT ARRAY ordlist TO sa_ordlist.*
    BEFORE ROW
      MESSAGE "Row count: " ||
  DIALOG.getArrayLength("sa_ordlist")
    ...
  END INPUT
  ...
```

ui.Dialog.getCurrentItem
Returns the current item having focus.

Syntax

```
getCurrentItem()
  RETURNING result STRING
```

Usage

The `getCurrentItem()` method returns the name of the current form item having the focus.

- If the focus is on an action view (typically, a `BUTTON` in the form layout), `getCurrentItem()` returns the name of the corresponding action. If several action views a bound to the same action handler with a unique name, there is no way to distinguish which action view has the focus.
- If the focus is in a simple field controlled by an `INPUT` or `CONSTRUCT` sub-dialog, `getCurrentItem()` returns the *[tab-name.]field-name* of that current field. The *tab-name.* prefix is added if a `FROM` clause is used with an explicit list of fields. No prefix is added if `FROM screen-record.*` is used or if `BY NAME` clause is used.
- If the focus is in a list controlled by a `DISPLAY ARRAY` sub-dialog, `getCurrentItem()` returns the *screen-array* name identifying the list.
- If the focus is in a field of a list controlled by an `INPUT ARRAY` sub-dialog, `getCurrentItem()` returns *screen-array.field-name*, identifying both the list and the current field. In some context, the current field is undefined. For example when entering the `INPUT ARRAY` sub-dialog, `getCurrentItem()` will return the *screen-array* only when in the `BEFORE INPUT` control block.

ui.Dialog.getCurrentRow
Returns the current row of the specified list.

Syntax

```
getCurrentRow(
   name STRING )
  RETURNING result INTEGER
```

1. *name* is the screen array name.

Usage

Use the `getCurrentRow()` method to retrieve the current row of an `INPUT ARRAY` or `DISPLAY ARRAY` list.

You must pass the name of the screen array to identify the list.

```
DIALOG
  DISPLAY ARRAY custlist TO sa_custlist.*
```

```
    BEFORE ROW
      MESSAGE "Current row: " || DIALOG.getCurrentRow("sa_custlist")
    ...
  END DISPLAY
  INPUT ARRAY ordlist TO sa_ordlist.*
    BEFORE ROW
      MESSAGE "Current row: " || DIALOG.getCurrentRow("sa_ordlist")
    ...
  END INPUT
  ...
```

ui.Dialog.getFieldBuffer
Returns the input buffer of the specified field.

Syntax

```
getFieldBuffer(
   field STRING )
  RETURNING result STRING
```

1. *field* is the field specification.

Usage

The `getFieldBuffer()` method returns the current input buffer of the specified field. The input buffer is used by the dialog to synchronize form fields and program variables. In some situations, especially when using the buffered mode or in a `CONSTRUCT`, you may want to access the field input buffer.

> **Note:** This method should only be used in dialogs allowing field input (`INPUT`, `INPUT ARRAY`, `CONSTRUCT`). The behavior is undefined when used in `DISPLAY ARRAY`.

The parameter is a field specification, a string containing the field qualifier, with an optional prefix ("`[table.]column`").

```
LET buff = DIALOG.getFieldBuffer("customer.cust_name")
```

The input buffer can be set with:

- A `DISPLAY TO` or `DISPLAY BY NAME` instruction
- The `FGL_DIALOG_SETBUFFER()` function (only for the current field)

For more details about field name specification, see Identifying fields in dialog methods on page 1818.

ui.Dialog.getFieldTouched
Returns the modification flag for a field.

Syntax

```
getFieldTouched(
   field-list STRING )
  RETURNING result BOOLEAN
```

1. *field-list* is the string with the list of field specification.

Usage

The `getFieldTouched()` method returns `TRUE` if the modification flag of the specified field(s) is set.

The *field-list* is a string containing the field qualifier, with an optional prefix ("`[table.]column`"), a table prefix followed by a dot and an asterisk ("`table.*`"), or a simple asterisk ("`*`").

This code checks if a specific field has been touched:

```
AFTER FIELD cust_name
  IF DIALOG.getFieldTouched("customer.cust_address") THEN
     ...
```

If the parameter is a screen record following by dot-asterisk, the method checks the touched flags of all the fields that belong to the screen record:

```
ON ACTION quit
  IF DIALOG.getFieldTouched("customer.*") THEN
    ...
```

When passing a simple asterisk (*) to the method, the runtime system will check all fields used by the dialog:

```
ON ACTION quit
  IF DIALOG.getFieldTouched("*") THEN
    ...
```

For more details about field name specification, see Identifying fields in dialog methods on page 1818.

ui.Dialog.getFieldValue
Returns the value of a field controlled by a dynamic dialog.

Syntax

```
getFieldValue(
   name STRING )
```

1. *name* is the name of the dialog.

Usage

The `getFieldValue()` method can be used when implementing a dynamic dialog, to return the value of a field:

```
DISPLAY d.getFieldValue( "customer.cust_addr" )
```

The first parameter defines the field to be set. For more details about field name specification, see Identifying fields in dialog methods on page 1818.

When used in a dynamic dialog controlling a list of records, this methods returns the value for a field in the current row. The current row can be set with the `setCurrentRow()` method.

ui.Dialog.getForm
Returns the current form used by the dialog.

Syntax

```
getForm()
  RETURNING result ui.Form
```

Usage

The `getForm()` method returns a `ui.Form` object as a handle to the current form used by the dialog.

Use this form object to modify elements of the current form. For example, you can hide some parts of the form with the `ui.Form.setElementHidden()` method.

ui.Dialog.getQueryFromField
Returns the SQL condition of a field used in a query by example dialog.

Syntax

```
getQueryFromField(
   field-name STRING )
  RETURNING sql-condition STRING
```

1. *field-name* is the name of the form field.

Usage

The `getQueryFromField()` method generates the SQL condition from the value entered in the field specified by the *field-name* parameter.

This method is used in the context of a construct dynamic dialog.

The result of this method can be used to build the `WHERE` part of a `SELECT` statement to find rows in a database.

Collect and concatenate field conditions returned from `getQueryFromField()`, then add `AND` or `OR` boolean operators to create an executable SQL query.

> **Note:** The SQL condition is generated according to the current type of database. The SQL syntax may vary according to the target database. Therefore you should not reuse the generated SQL conditions. However, the user input of a query by example dialog can be reused for different type of databases (see ui.Dialog.setFieldValue on page 1813 and ui.Dialog.getFieldValue on page 1801)

ui.Dialog.getSortKey
Returns the name of the sort field selected by the user.

Syntax

```
getSortKey(
   screen-array STRING )
  RETURNING field-name STRING
```

1. *screen-array* is the name of the screen array.

Usage

The `getSortKey()` method returns the form field name selected by the user to sort rows.

This method is used in the context of the `ON SORT` trigger.

> **Note:** If the sort is reset, the `getSortKey()` method returns `NULL`.

ui.Dialog.getSortReverse
Indicates the sort order direction (FALSE=ascending, TRUE=descending)

Syntax

```
getSortReverse(
   screen-array STRING )
  RETURNING result BOOLEAN
```

1. *screen-array* is the name of the screen array.

Usage

The `getSortReverse()` method returns `FALSE` if the sort order is ascending, and `TRUE` if the sort is in descending order.

This method is used in the context of the `ON SORT` trigger.

ui.Dialog.insertNode
Inserts a new node in the specified tree.

Syntax

```
insertNode(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index of the next sibling node in the program array (starts at 1).

Usage

The `insertNode()` method is similar to `insertRow()`, except that it has to be used when the list dialog controls a tree view.

This method must be used when modifying the array of a tree view during the execution of the dialog, for example when implementing a dynamic tree with `ON EXPAND` / `ON COLLAPSE` triggers. Before the execution of the dialog, you can fill the program array directly. This includes the context of `BEFORE DISPLAY` or `BEFORE DIALOG` control blocks.

When adding rows for a tree view, the id of the parent node and new node matters because that information is used to build the internal tree structure. When calling `insertNode()`, you pass the index of the next sibling node. In the program array, the parent-id member of the new node will automatically be initialized with the value of the parent-id of the next sibling node, then the internal tree structure is rebuilt.

ui.Dialog.insertRow
Inserts a new row in the specified list.

Syntax

```
insertRow(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index where the row must be inserted (starts at 1).

Usage

The `insertRow()` method inserts a row in the list, at a given position.

> **Important:** This method is designed to be used in an `ON ACTION` block. It must not be called in control blocks such as `BEFORE ROW`, `AFTER ROW`, `BEFORE INSERT`, `AFTER INSERT`, `BEFORE DELETE`, or `AFTER DELETE`.

The method is similar to inserting a new element in the program array, except the internal dialog registers are automatically updated (like the total number of rows returned by `getArrayLength()`). If the list is decorated with cell attributes, the program array defining the attributes will also be synchronized. If multi-row selection is enabled, selection flags of existing rows are kept. Selection information is synchronized (i.e., flags are shifted down) for all rows after the new inserted row.

> **Note:** The purpose of this method is to implement business logic required to modify the record list in the current dialog. It is typically used in a `DISPLAY ARRAY` dialog. Avoid using this method in `INPUT ARRAY`. To allow the end user to append, modify or delete rows in a `DISPLAY ARRAY`, use list modification interaction blocks.

After the method is called, a new row is created in the program array, so you can assign values to the variables before the control goes back to the user. The `getArrayLength()` method will return the new row count.

The method does not set the current row and does not give the focus to the list; you need to call `setCurrentRow()` and execute `NEXT FIELD` to give the focus.

The `insertRow()` method must not be used when controlling a tree view. Use the `insertNode()` method instead.

This method does not execute any `BEFORE ROW`, `BEFORE INSERT`, `AFTER INSERT` or `AFTER ROW` control blocks.

If the index is greater than the number of rows, a new row is appended at the end or the list. This is the equivalent of calling the `appendRow()` method.

If the list is empty, `getCurrentRow()` returns zero. If zero is returned, use 1 to reference the first row, otherwise you can get a -1326 error when using the program array.

Example

This example shows a user-defined action to insert ten rows in the list at the current position:

```
ON ACTION insert_some_rows
  LET r = DIALOG.getCurrentRow("sa")
  IF r == 0 THEN LET r = 1 END IF
  FOR i = 10 TO 1 STEP -1
    CALL DIALOG.insertRow("sa", r)
    LET p_items[r].item_quantity = 1.00
  END FOR
```

ui.Dialog.isRowSelected
Queries row selection for a give list and row.

Syntax

```
isRowSelected(
   name STRING,
   index INTEGER )
  RETURNING result BOOLEAN
```

1. *name* is the screen array name.
2. *index* is a row index.

Usage

If multi-row selection is enabled with `setSelectionMode()`, you can check whether a row is selected with the `isRowSelected()` method:

```
ON ACTION check_current_row_selected
  IF DIALOG.isRowSelected( "sr", DIALOG.getCurrentRow("sr") ) THEN
    MESSAGE "Current row is selected."
  END IF
```

If multi-row selection is off, the method returns `TRUE` for the current row and `FALSE` for other rows.

ui.Dialog.nextEvent
Waits for a dialog event.

Syntax

```
nextEvent()
   RETURNING event STRING
```

1. *event* is the name of the dialog event that raised.

Usage

The `nextEvent()` waits for a dialog event to occur, and returns a string that identifies the dialog event that has raised.

This method is typically used in a `WHILE` loop, to implement a dynamic dialog.

A dialog event can be a user-defined trigger such as `"ON ACTION print"`, or an implicit trigger such as `"BEFORE ROW"`, corresponding to the control blocks that can be defined in static dialog instructions such as `DISPLAY ARRAY`.

User-defined triggers are added to the dynamic dialog with the `addTrigger()` method:

Table 390: User-defined triggers for dynamic dialogs

Trigger name	Description	Dialog block equivalent
`ON ACTION` *action-name*	Action handler for the action identified by *action-name*.	`ON ACTION` block
`ON APPEND`	Row addition action handler for a display array dynamic dialog.	`ON APPEND` block
`ON DELETE`	Row deletion action handler for a display array dynamic dialog.	`ON DELETE` block
`ON INSERT`	Row insertion action handler for a display array dynamic dialog.	`ON INSERT` block
`ON UPDATE`	Row modification action handler for a display array dynamic dialog.	`ON UPDATE` block

Implicit dialog triggers are predefined and can be detected and handled in the dialog `WHILE` loop if needed:

Table 391: Implicit triggers for dynamic dialogs

Trigger name	Description	Dialog block equivalent
`BEFORE DISPLAY`	Initialization of the display array dynamic dialog.	`BEFORE DISPLAY` block
`BEFORE INPUT`	Initialization of the input by name dynamic dialog.	`BEFORE INPUT` block
`AFTER DISPLAY`	End of the display array dynamic dialog.	`AFTER DISPLAY` block
`AFTER INPUT`	End of the input by name dynamic dialog.	`AFTER INPUT` block
`BEFORE ROW`	Moving to a new row in a display array dynamic dialog.	`BEFORE ROW` block
`AFTER ROW`	Leaving the current row in a display array dynamic dialog.	`AFTER ROW` block

Trigger name	Description	Dialog block equivalent
`BEFORE FIELD field-name`	Entering the field *field-name* in an input dynamic dialog.	`BEFORE FIELD` block
`AFTER FIELD field-name`	Leaving the field *field-name* in an input dynamic dialog.	`AFTER FIELD` block

ui.Dialog.nextField
Registering the next field to jump to.

Syntax

```
nextField(
   name STRING )
```

1. *name* is the form field name.

Usage

The `nextField()` method registers the name of the next field that must get the focus when control goes back to the dialog.

The first parameter identifies the form field, see Identifying fields in dialog methods for more details.

This method is similar to the `NEXT FIELD` instruction, except that it does not implicitly break the program flow. If you want to get the same behavior as `NEXT FIELD`, the method call must be followed by a `CONTINUE DIALOG` instruction, or an equivalent instruction, in case of singular dialog.

Since this method takes an expression as parameter, you can write generic code, when the name of the target field is not known at compile time. In the next example, the `check_value()` function returns a field name where the value does not satisfy the validation rules.

```
DEFINE fn STRING
...
ON ACTION save
  IF ( fn:= check_values() ) IS NOT NULL THEN
    CALL DIALOG.nextField(fn)
    CONTINUE DIALOG
  END IF
  CALL save_data()
  ...
```

ui.Dialog.selectionToString
Serializes data of the selected rows.

Syntax

```
selectionToString(
   name STRING )
  RETURNING result STRING
```

1. *name* is the screen array name.

Usage

The `selectionToString()` method can be used to get a tab-separated value list of the selected rows.

When multi-row selection is disabled, the method serializes the current row.

You typically use this method in conjunction with drag & drop to fill the buffer, by using a text/plain MIME type, to export data to external applications.

```
ON ACTION serialize
  LET buff = DIALOG.selectionToString( "sr" )
```

Numeric and date data will be formatted according to current locale settings (DBMONEY, DBDATE).

The visual presentation of data is respected: The dialog will copy the rows in the sort order specified by the user, moved columns will appear in the same positions as in the table and hidden columns will be ignored. Note that phantom columns are not copied.

Items in the tab-separated record will be surrounded by double-quotes if the value contains special characters such as a newline, a double-quote, or controls characters with ASCII code < 0x20. Double-quotes in the value will be doubled.

ui.Dialog.setActionActive
Enabling and disabling dialog actions.

Syntax

```
setActionActive(
  name STRING,
  active BOOLEAN )
```

1. *name* is the name of the action.
2. *active* is a boolean value.

Usage

Use the `setActionActive()` method to enable or disable an action.

```
CALL DIALOG.setActionActive("zoom", FALSE)
```

The second parameter of the method must be a boolean expression that evaluates to 0 (FALSE) or 1 (TRUE).

For more details about action names, see Identifying actions in dialog methods on page 1817.

ui.Dialog.setActionHidden
Handling default action view visibility.

Syntax

```
setActionHidden(
   name STRING,
   hide INTEGER )
```

1. *name* is the name of the action.
2. *hide* is 1 to hide, 0 to show.

Usage

Use the `setActionHidden()` method to hide the default view (and context menu option) of an action.

```
CALL DIALOG.setActionHidden( "confirm", 1 )
```

The first parameter identifies the action object of the dialog

For more details about action names, see Identifying actions in dialog methods.

ui.Dialog.setArrayAttributes
Define cell decoration attributes array for the specified list (singular or multiple dialogs).

Syntax

```
setArrayAttributes(
   name STRING,
   attributes ARRAY )
```

1. *name* is the screen array name.
2. *attributes* is a program array defining the cell attributes.

Usage

In an `INPUT ARRAY` or `DISPLAY ARRAY` dialog, the `setArrayAttributes()` method can be used to specify display attributes for each cell.

The `setArrayAttributes()` when several screen arrays are defined, to be able to idenfify the list by the name of the screen array. An equivalent method called `setCellAttributes()` can be used, for dialogs where only one screen array is defined.

Possible values for cell attributes are a combination of the following:

- The TTY attribute `reverse`
- The TTY attribute `blink`
- The TTY attribute `underline`
- One of the TTY colors: `white, yellow, magenta, red, cyan, green, blue, black`

Define an array with the same number of record elements as the data array used by the `INPUT ARRAY` or `DISPLAY ARRAY`. Each element must have the same name as in the data array, and must be defined with a character data type (typically: `STRING`):

```
DEFINE data DYNAMIC ARRAY OF RECORD
         pkey INTEGER,
         name VARCHAR(50)
       END RECORD
DEIFNE attributes DYNAMIC ARRAY OF RECORD
         pkey STRING,
         name STRING
       END RECORD
```

Fill the display attributes array with color and video attributes. These must be specified in lowercase characters and separated by a blank (ex: "`red reverse`"):

```
FOR i=1 TO data.getLength()  -- length from data array!
    LET attributes[i].name = "blue reverse"
END FOR
```

Then, attach the array to the dialog with the `setArrayAttributes()` method, in a `BEFORE DIALOG`, `BEFORE INPUT` or `BEFORE DISPLAY` block:

```
BEFORE DIALOG
   CALL DIALOG.setArrayAttributes( "sr", attributes )
```

Like data values, if you change the cell attributes during the dialog, these are not displayed automatically unless the `UNBUFFERED` mode is used.

```
ON ACTION modify_cell_attribute
   LET attributes[arr_curr()].name = "red reverse"
```

If you set NULL to a element, the default TTY attributes will be reset:

```
ON ACTION clean_cell_attribute
   LET attributes[arr_curr()].name = NULL
```

ui.Dialog.setArrayLength
Sets the total number of rows in the specified list.

Syntax

```
setArrayLength(
   name STRING,
   len INTEGER )
```

1. *name* is the screen array name.
2. *len* is the new size of the array.

Usage

The `setArrayLength()` method is used to specify the total number of rows when using the `DISPLAY ARRAY` paged mode. The name of the screen array is passed to identify the list, followed by an integer expression defining the number of rows.

When using a dynamic array without paged mode (i.e. without the `ON FILL BUFFER` clause), you don't need to specify the total number of rows to the `DIALOG` instruction: It is defined by the number of elements in the array. However, when using the paged mode in a `DISPLAY ARRAY`, the total number of rows does not correspond to the elements in the program array, because the program array holds only a page of the whole list. In any other cases, a call to this method is just ignored.

A call to `setArrayLength()` will not trigger the execution of the `ON FILL BUFFER` clause immediately. The trigger to fill a page of rows will be executed when all the user code has been execute and the control goes back to the dialog instruction.

The `setArrayLength()` method is also be used to fix the final number of rows when using `COUNT=-1` attribute, to implement a paged list without knowing the total number of rows when the dialog starts.

ui.Dialog.setCellAttributes
Define cell decoration attributes array for the specified list (singular dialog only).

Syntax

```
setCellAttributes(
   attributes ARRAY )
```

1. *attributes* is a program array defining the cell attributes.

Usage

In an `INPUT ARRAY` or `DISPLAY ARRAY` dialog, the `setCellAttributes()` method can be used to specify display attributes for each cell.

The `setCellAttributes()` method is designed for dialog programming, where only one screen array is used (for example, in a singular `DISPLAY ARRAY` dialog). An equivalent method called `setArrayAttributes()` can be used, when several screen arrays are defined in a multiple dialog, to be able to idenfify the list by the name of the screen array.

Possible values for cell attributes are a combination of the following:

- The TTY attribute `reverse`
- The TTY attribute `blink`

- The TTY attribute `underline`
- One of the TTY colors: `white, yellow, magenta, red, cyan, green, blue, black`

Define an array with the same number of record elements as the data array used by the `INPUT ARRAY` or `DISPLAY ARRAY`. Each element must have the same name as in the data array, and must be defined with a character data type (typically: `STRING`):

```
DEFINE data DYNAMIC ARRAY OF RECORD
         pkey INTEGER,
         name VARCHAR(50)
       END RECORD
DEIFNE attributes DYNAMIC ARRAY OF RECORD
         pkey STRING,
         name STRING
       END RECORD
```

Fill the display attributes array with color and video attributes. These must be specified in lowercase characters and separated by a blank (ex: "`red reverse`"):

```
FOR i=1 TO data.getLength()  -- length from data array!
    LET attributes[i].name = "blue reverse"
END FOR
```

Then, attach the array to the dialog with the `setCellAttributes()` method, in a `BEFORE INPUT` or `BEFORE DISPLAY` block:

```
BEFORE DISPLAY
   CALL DIALOG.setCellAttributes( attributes )
```

Like data values, if you change the cell attributes during the dialog, these are not displayed automatically unless the `UNBUFFERED` mode is used.

```
ON ACTION modify_cell_attribute
   LET attributes[arr_curr()].name = "red reverse"
```

If you set `NULL` to a element, the default TTY attributes will be reset:

```
ON ACTION clean_cell_attribute
   LET attributes[arr_curr()].name = NULL
```

ui.Dialog.setCompleterItems
Define autocompletion items for the a field defined with `COMPLETER` attribute.

Syntax

```
setCompleterItems(
   items-array DYNAMIC ARRAY OF STRING )
```

1. *items-array* defines the list of proposals to be passed to the front-end.

Usage

The `setCompleterItems(`*`items-array`*`)` dialog method defines the list of proposals for the current field, to implement autocompletion.

The field must be defined in the form with the `COMPLETER` attribute.

The list of proposal items is passed as a dynamic array of strings:

```
DEFINE items DYNAMIC ARRAY OF STRING
```

To cleanup the proposal list for a give field, pass `NULL` as second parameter to the function.

> **Important:** The method will raise error -8114, if the list of items contains more than 50 elements. Note that this error is not trappable with exception handlers like `TRY/CATCH`, the code must avoid to reach the limit.

See Enabling autocompletion on page 1274 for more details.

Example

```
DEFINE items DYNAMIC ARRAY OF STRING
...
   ON CHANGE firstname
      -- fill the array with items
      LET items[1] = "Ann"
      LET items[2] = "Anna"
      LET items[3] = "Annabel"
      CALL DIALOG.setCompleterItems(items)
```

ui.Dialog.setCurrentRow
Sets the current row in the specified list.

Syntax

```
setCurrentRow(
   name STRING,
   row INTEGER )
```

1. *name* is the screen array name.
2. *row* is the new row in the array.

Usage

Use the `setCurrentRow()` method to change the current row in an `INPUT ARRAY` or `DISPLAY ARRAY` list. You must pass the name of the screen array to identify the list, and the new row number.

```
DEFINE x INTEGER
DIALOG
  DISPLAY ARRAY custlist TO sa_custlist.*
    ...
  END DISPLAY
  ON ACTION goto_x
    CALL DIALOG.setCurrentRow("sa_custlist", x)
    ...
```

Moving to a different row with `setCurrentRow()` will not trigger control blocks such as `BEFORE ROW` / `AFTER ROW`, as the `fgl_set_arr_curr()` built-in function does.

The `setCurrentRow()` method will not set the focus; You need to use `NEXT FIELD` to set the focus to a list. (This works with `DISPLAY ARRAY` as well as with `INPUT ARRAY`.)

If the passed row index is lower than 1, the first row will be selected. If the row index is greater than the total number of rows, the last row will be selected.

If the new current row is out of the current page of rows, the dialog will adapt the list offset to make the new current row visible.

If multi-row selection is enabled, all selection flags of rows are cleared, and the new current row gets automatically selected.

ui.Dialog.setFieldActive
Enable and disable form fields.

Syntax

```
setFieldActive(
   field-list STRING,
   active BOOLEAN )
```

1. *field-list* is the string with the list of field specification.
2. *active* is a boolean value.

Usage

The `setFieldActive()` method can be used to enable / disable form fields.

The *field-list* is a string containing the field qualifier, with an optional prefix ("`[table.]column`"), or a table prefix followed by a dot and an asterisk ("`table.*`").

```
CALL DIALOG.setFieldActive( "customer.cust_addr",
            (rec.cust_name IS NOT NULL) )
```

Do not disable all fields of a dialog, otherwise the dialog execution stops (at least one field must get the focus during a dialog execution).

For more details about field name specification, see Identifying fields in dialog methods on page 1818.

ui.Dialog.setFieldTouched
Sets the modification flag of the specified field.

Syntax

```
setFieldTouched(
   field-list STRING,
   touched BOOLEAN )
```

1. *field-list* is the string with the list of field specification.
2. *touched* is the boolean value to set the modification flag.

Usage

The `setFieldTouched()` method can be used to change the modification flag of the specified field(s).

The *field-list* is a string containing the field qualifier, with an optional prefix ("`[table.]column`"), or a table prefix followed by a dot and an asterisk ("`table.*`").

You typically use this method to set the touched flag when assigning a variable, to emulate a user input. Remember when using the `UNBUFFERED` mode, you don't need to display the value to the fields. The `setFieldTouched()` method is provided as a 3GL replacement for the `DISPLAY BY NAME / TO` instructions to set the modification flags.

```
ON ACTION zoom_city
  LET p_cust.cust_city = zoom_city()
  CALL DIALOG.setFieldTouched("customer.cust_city", TRUE)
    ...
```

If the parameter is a screen record following by dot-asterisk, the method checks the modification flags of all the fields that belong to the screen record. You typically use this to reset the touched flags of a group of fields, after modifications have been saved to the database, to get back to the initial state of the dialog:

```
ON ACTION save
  CALL save_cust_record()
  CALL DIALOG.setFieldTouched("customer.*", FALSE)
    ...
```

The modification flags are reset to false when using an `INPUT ARRAY` list, every time you leave the modified row.

For more details about field name specification, see Identifying fields in dialog methods on page 1818.

ui.Dialog.setFieldValue
Sets the value of a field controlled by the dialog object.

Syntax

```
setFieldValue(
   name STRING,
   value fgl-type )
```

1. *name* is the name of the field.
2. *value* is the value to be set.

Usage

The `setFieldValue()` method can be used when implementing a dynamic dialog, to set the value of a field:

```
DEFINE default_address STRING,
       default_creadate DATE
...
CALL d.setFieldValue( "customer.cust_addr", default_address )
CALL d.setFieldValue( "customer.cust_creadate", default_creadate )
```

The first parameter defines the field to be set. For more details about field name specification, see Identifying fields in dialog methods on page 1818.

When used in a dynamic dialog controlling a list of records, this methods sets the value for a field in the current row. The current row can be set with the `setCurrentRow()` method. This is also true when filling the dynamic dialog with rows: You must first set the current row with `setCurrentRow()`, then set field (i.e. cell values) with `setFieldValue()`.

Example

The following code example implements a `FOR` loop to copy values of all fields of the `d_disparr` dialog to the field of the `d_recinp` dialog:

```
DEFINE row, i INTEGER,
       h base.SqlHandle,
       fields DYNAMIC ARRAY OF RECORD
                  name STRING,
                  type STRING
              END RECORD
       d_rec ui.Dialog,
       d_list ui.Dialog
...
    -- Fill the array with rows from an SqlHandle object
    CALL h.open()
```

```
    LET row = 0
    WHILE status == 0
        -- must set the current row before setting values
        CALL d_list.setCurrentRow("sr_custlist", row:=row+1 )
        FOR i = 1 TO h.getResultCount()
            CALL d_list.setFieldValue( h.getResultName(i),
 h.getResultValue(i) )
        END FOR
        CALL h.fetch()
    END WHILE
    CALL d_list.setCurrentRow("sr_custlist", 1)
...
    -- Copy field values from d_list to d_rec dialog
    FOR i=1 TO fields.getLength()
        CALL d_rec.setFieldValue( fields[i].name,
               d_list.getFieldValue( fields[i].name )
             )
    END FOR
```

ui.Dialog.setSelectionMode
Defines the row selection mode for the specified list.

Syntax

```
setSelectionMode(
   name STRING,
   mode INTEGER )
```

1. *name* is the screen array name.
2. *mode* defines the selection mode (0, 1).

Usage

In `DISPLAY ARRAY` instructions, the `setSelectionMode()` method can be used to enable/disable multi-row selection.

Possible values of the *mode* parameter are 0 (single row selection) or 1 (multi-range selection). Other values are reserved for future use.

If multi-row selection is switched off, selected rows get deselected.

For more details about multi-row selection, see Multiple row selection on page 1381.

ui.Dialog.setSelectionRange
Sets the row selection flags for a range of rows.

Syntax

```
setSelectionRange(
   name STRING,
   start INTEGER,
   end INTEGER,
   value BOOLEAN )
```

1. *name* is the screen array name.
2. *start* is the starting row index.
3. *end* is the ending row index.
4. *value* is the selection flag to set.

Usage

If multi-row selection is enabled with *setSelectionMode()*, you can set the selection flags for a range of rows with the `setSelectionRange()` method.

```
ON ACTION select_all
  CALL DIALOG.setSelectionRange( "sr", 1, -1, TRUE)
```

The start and end index must be in the range of possible row indexes (from 1 to `DIALOG.getArrayLength()`).

If you specify an end index of -1, it will set the flags from start index to the end of the list.

ui.Dialog.validate
Check form level validation rules.

Syntax

```
validate(
   field-list STRING )
  RETURNING result INTEGER
```

1. *field-list* is the string with the list of field specification.

Usage

Use the `validate()` method in order to execute `NOT NULL`, `REQUIRED` and `INCLUDE` validation rules defined in the form specification files.

Can the method by passing a list of fields or screen records as parameter.

The method returns zero if success and the input error code of the first field which does not satisfy the validation rules.

The current field is always checked, even if it is not part of the validation field list. This is mandatory, otherwise the current field may be left with invalid data.

If an error occurs, the `validate()` method automatically displays the corresponding error message, and registers the next field to jump to when the interactive instruction gets the control back.

The `validate()` method does not stop code execution if an error is detected. You must execute a `CONTINUE DIALOG` or `CONTINUE INPUT` instruction to cancel the code execution.

A typical usage is for a "save" action:

```
ON ACTION save
   IF DIALOG.validate("cust.*") < 0 THEN
      CONTINUE DIALOG
   END IF
   CALL customer_save()
```

For more details about field name specification, see Identifying fields in dialog methods on page 1818.

ui.Dialog.visualToArrayIndex
Converts the visual index to the program array index for a given screen array.

Syntax

```
visualToArrayIndex(
   name STRING,
   index INTEGER )
```

1. *name* is the screen array name.
2. *index* is the index of the row as seen by the end user.

Usage

When the end user sorts rows in a table, the visual row index may differ from the program array index.

Use this method to convert a row index as seen by the end user, to the program array index. For example, if the application implements a feature that allows the user to enter a row index to jump to that row, it will be entered as a visual row index. You must convert this index to the program array index, for example to make a `setCurrentRow()`.

```
CALL DIALOG.setCurrentRow("sr", DIALOG.visualToArrayIndex("sr", user_index))
```

Usage
Referencing the current dialog

In order to reference the current dialog, you can define a variable with the `ui.Dialog` type, and get the current dialog object with the `ui.Dialog.getCurrent()` method:

```
DEFINE d ui.Dialog

INPUT BY NAME ...
   BEFORE DIALOG
      LET d = ui.Dialog.getCurrent()
      CALL d.setActionActive("zoom", FALSE)
   ...
```

As an alternative and to simplify programming, you should use the `DIALOG` keyword in the context of the interactive instruction block. the `DIALOG` keyword is a predefined object variable referencing the current dialog. The `DIALOG` variable can only be used inside the interactive instruction block:

```
INPUT BY NAME custid, custname
  ON ACTION disable
    CALL DIALOG.setFieldActive("custid", FALSE)
END INPUT
```

Passing a dialog reference to functions

Using the `DIALOG` keyword outside a dialog instruction block results in a compilation error. However, you can pass the object to a function that defines the dialog parameter with the `ui.Dialog` type.

The next example passes the `DIALOG` object reference to the `setupDialog()` function, which implements action activation rules that must be applied after different events, during the dialog execution:

```
INPUT BY NAME custid, custname, custaddr
  BEFORE INPUT
    CALL setupDialog(DIALOG)
  ...
  ON ACTION check_address
    ...
    CALL setupDialog(DIALOG)
  ...
END INPUT

FUNCTION setupDialog(d)
  DEFINE d ui.Dialog
  DEFINE isAdmin BOOLEAN
  LET isAdmin = (global_params.user_group == "admin")
  CALL d.setActionActive("delete", isAdmin)
  CALL d.setActionActive("convert", isAmdin)
  CALL d.setActionActive("check_address",
```

```
           isAdmin AND rec.custaddr IS NOT NULL)
END FUNCTION
```

Identifying actions in dialog methods

In `ui.Dialog` methods such as `setActionActive()`, the first parameter identifies the action object to be modified. This parameter can be full-qualified or partly-qualified. If you don't specify a full-qualified name, the action object will be identified according to the focus context.

The action name specification can be any of the following:

- *action-name*
- *dialog-name.action-name*
- *dialog-name.field-name.action-name*
- *field-name.action-name* (singular dialogs only)

Here *action-name* identifies the name of the action specified in `ON ACTION` *`action-name`* or `COMMAND` "*`action-name`*" handlers, while *dialog-name* identifies the singular dialog or sub-dialog and *field-name* defines the field bound to the action `INFIELD` clause of `ON ACTION`.

The action name must be passed in lowercase letters.

The runtime system will raise the error **-8089** if the action specified by *[dialog-name.][field-name.]action-name* can not be found within the current dialog.

> **Note:** As a general rule, assign unique action names for each specific dialog action, to avoid the usage of dialog and/or field identifiers.

In the `DIALOG` instruction, actions can be prefixed with the sub-dialog identifier. However, if methods like `setActionActive()` are called in the context of the sub-dialog, the prefix can be omitted. When using a field-specific action defined with the `INFIELD` clause of `ON ACTION`, you can identify the action with the full-qualified name *dialog-name.field-name.action-name*. Like sub-dialog actions, if you specify only *action-name*, the runtime system will search for the action object according to the focus context.

Note that an `INPUT` or `CONSTRUCT` sub-dialogs have no identifier by default. The dialog name can be defined with the `NAME` attribute. For more details, see Identifying sub-dialogs in procedural DIALOG on page 1152.

When using a singular dialog like `INPUT`, you can identify field-specific actions by *field-name.action-name*, if the dialog was defined without a `NAME` attribute.

Example

```
DIALOG ...
   ...
   INPUT BY NAME cust_rec.* ATTRIBUTES(NAME="cust")
      ON ACTION compare
         ...
      ON ACTION check INFIELD cust_city
         ...
   END INPUT
   ...
   DISPLAY ARRAY orders TO sr_ord.*
      ...
      ON ACTION archive
         ...
   END DISPLAY
   ...
   ON ACTION print
      ...
   ON ACTION disable_all
      CALL DIALOG.setActionActive("cust.compare", FALSE)
      CALL DIALOG.setActionActive("cust.cust_city.check", FALSE)
```

```
      CALL DIALOG.setActionActive("sr_ord.archive", FALSE)
      CALL DIALOG.setActionActive("print", FALSE)
END DIALOG
```

Identifying fields in dialog methods

In `ui.Dialog` methods such as `setFieldActive()`, the first parameter identifies the form field (or, for some methods, a list of fields) to be modified. The form field names can be full-qualified or partly-qualified.

Fields are identified by the form field name specification, not the program variable name used by the dialog. Remember form fields are bound to program variables with the binding clause of dialog instruction (`INPUT` *`variable-list`* `FROM` *`field-list`*, `INPUT BY NAME` *`variable-list`*, `CONSTRUCT BY NAME` *`sql`* `ON` *`column-list`*,`CONSTRUCT` *`sql`* `ON` *`column-list`* `FROM` *`field-list`*, `INPUT ARRAY` *`array-name`* `FROM` *`screen-array.*`*).

The field name specification can be any of the following:

- *field-name*
- *table-name.field-name*
- *screen-record-name.field-name*
- FORMONLY.*field-name*

Here are some examples:

- `"cust_name"`,
- `"customer.cust_name"`,
- `"cust_screen_record.cust_name"`,
- `"item_screen_array.item_label"`,
- `"formonly.total"`,
- `"customer.*"` (only some methods accept the "dot asterisk" notation)

When no field name prefix is used, the first form field matching that field name will be used. If the field specification is invalid (i.e. no field in the current dialog matches the field specification), the method will throw the error -1373.

When using a prefix in the field name specification, it must match the field prefix assigned by the dialog according to the field binding method used at the beginning of the interactive statement: When no screen-record has been explicitly specified in the field binding clause (for example, when using `INPUT BY NAME` *`variable-list`*,) the field prefix must be the database table name (or `FORMONLY`) used in the form file, or any valid screen-record using that field. But when the `FROM` clause of the dialog specifies an explicit screen-record (for example, in `INPUT` *`variable-list`* `FROM` *`screen-record.* /field-list-with-screen-record-prefix`* or `INPUT ARRAY` *`array-name`* `FROM` *`screen-array.*`*) the field prefix must be the screen-record name used in the `FROM` clause.

The methods `validate()`, `setFieldActive()`, `setFieldTouched()`, `getFieldTouched()` can take a list of fields as parameter, by using the "dot-asterisk " notation (screen-record.*). This way you can check, query or change a complete list of fields in one method call:

```
ON ACTION save
  CALL save_cust_record()
  CALL DIALOG.setFieldTouched("customer.*", FALSE)
    ...
```

Examples

Example 1: Disable fields dynamically

```
FUNCTION input_customer()
  DEFINE custid INTEGER
  DEFINE custname CHAR(10)
  INPUT BY NAME custid, custname
```

```
    ON ACTION enable
      CALL DIALOG.setFieldActive("custid",1)
    ON ACTION disable
      CALL DIALOG.setFieldActive("custid",0)
  END INPUT
END FUNCTION
```

Example 2: Get the form and hide fields

```
FUNCTION input_customer()
  DEFINE f ui.Form
  DEFINE custid INTEGER
  DEFINE custname CHAR(10)
  INPUT BY NAME custid, custname
    BEFORE INPUT
      LET f = DIALOG.getForm()
      CALL f.setElementHidden("customer.custid",1)
  END INPUT
END FUNCTION
```

Example 3: Pass a dialog object to a function

```
 FUNCTION input_customer()
   DEFINE custid INTEGER
   DEFINE custname CHAR(10)
   INPUT BY NAME custid, custname
     BEFORE INPUT
       CALL setup_dialog(DIALOG)
   END INPUT
 END FUNCTION

 FUNCTION setup_dialog(d)
   DEFINE d ui.Dialog
   CALL d.setActionActive("print",user.can_print)
   CALL d.setActionActive("query",user.can_query)
 END FUNCTION
```

Example 4: Set display attributes for cells

```
FUNCTION display_items()
   DEFINE i INTEGER
   DEFINE items DYNAMIC ARRAY OF RECORD
                key INTEGER,
                name CHAR(10)
              END RECORD
   DEFINE attributes DYNAMIC ARRAY OF RECORD
                key STRING,
                name STRING
              END RECORD

   FOR i=1 TO 10
     CALL items.appendElement()
     LET items[i].key = i
     LET items[i].name = "name " || i
     CALL attributes.appendElement()
     IF i MOD 2 = 0 THEN
       LET attributes[i].key = "red"
       LET attributes[i].name = "blue reverse"
     ELSE
       LET attributes[i].key = "green"
       LET attributes[i].name = "magenta reverse"
     END IF
```

```
   END FOR

   DISPLAY ARRAY items TO sr.* ATTRIBUTES(UNBUFFERED)
       BEFORE DISPLAY
         CALL DIALOG.setCellAttributes(attributes)
       ON ACTION att_modify_cell
         LET attributes[2].key = "red reverse"
       ON ACTION att_clear_cell
         LET attributes[2].key = NULL
   END DISPLAY

END FUNCTION
```

The ComboBox class
The `ui.ComboBox` class provides an interface to the `COMBOBOX` form field view in the abstract user interface tree.

In .per form specification files, a `COMBOBOX` form field defines both a form field and a view for that model. The `ui.ComboBox` class is an interface to the view of a `COMBOBOX` form field. It is typically used to configure the widget dynamically in programs, for example to create the list of items shown in the drop down box.

ui.ComboBox methods
Methods of the `ui.ComboBox` class.

Table 392: Class methods

Name	Description
`ui.ComboBox.forName( name STRING ) RETURNING result ui.ComboBox`	Search for a combobox in the current form.
`ui.ComboBox.setDefaultInitializer( funcname STRING )`	Define the default initializer for combobox form items.

Table 393: Object methods

Name	Description
`addItem( value STRING, label STRING )`	Add an element to the item list.
`clear()`	Clear the item list of a combobox.
`getColumnName() RETURNING result STRING`	Get the column name of the form field.
`getIndexOf( name STRING )`	Get an item position by name.

Name	Description
`RETURNING result INTEGER`	
`getItemCount()` `RETURNING result INTEGER`	Get the number of items.

`getItemName(`
`position INTEGER )`
`RETURNING result STRING`
	Get an item name by position.

`getItemText(`
`position INTEGER )`
`RETURNING result STRING`
	Get the item text by position.

`getTableName()` `RETURNING result STRING`	Get the table prefix of the form field.
`getTag()` `RETURNING result STRING`	Get the combobox tag value.

`getTextOf(`
`name STRING )`
`RETURNING result STRING`
	Get the item text by name.

`removeItem(` `name STRING )`	Remove an item by name.

ui.ComboBox.setDefaultInitializer
Define the default initializer for combobox form items.

Syntax

```
ui.ComboBox.setDefaultInitializer(
   funcname STRING )
```

1. *funcname* is the name of the initialization function.

Usage

The `ui.ComboBox.setDefaultInitializer()` class method defines the default initialization function to be called each time a `COMBOBOX` form field is created when loading forms. Use this method if you want to define a global/default initialization function for all comboboxes of the program. For individual comboboxes, consider using the `INITIALIZER` form field attribute instead.

> **Tip:** Consider defining the initialization function name in lowercase letters. The language syntax allows case-insensitive functions names, but to avoid mistakes, it is recommended to use a common naming convention with lowercase letters.

The function is called with the `ui.ComboBox` object as the parameter.

The combobox initialization functions are typically used to fill the drop down list with items.

Example

```
MAIN
    ...
    CALL ui.ComboBox.setDefaultInitializer("cb_init")
    ...
    OPEN FORM f1 FROM "customers"
    DISPLAY FORM f1 -- initialization function is called
    ...
END MAIN

FUNCTION cb_init(cb)
  DEFINE cb ui.ComboBox
  CALL cb.clear()
  CALL cb.addItem(0,"Undefined")
  ...
END FUNCTION
```

ui.ComboBox.forName
Search for a combobox in the current form.

Syntax

```
ui.ComboBox.forName(
   name STRING )
  RETURNING result ui.ComboBox
```

1. *name* is the name of `COMBOBOX` form item.

Usage

The `ui.ComboBox.forName()` class method searches for a `ui.ComboBox` object by form field name in the current form.

> **Important:** The form field name must be in lower-case letters: The language syntax allows case-insensitive form field names, and the runtime system must reference fields in lowercase letters internally. Since the form compiler converts field names to lowercase in the 42f file, the name must be lowercase in this method call.

After loading a form with `OPEN WINDOW WITH FORM`, use the class method to retrieve a `ui.ComboBox` object into a variable defined as a `ui.ComboBox`.

```
DEFINE cb ui.ComboBox
LET cb = ui.ComboBox.forName("formonly.airport")
```

Verify the function has returned an object, as the form field may not exist.

```
IF cb IS NULL THEN
  ERROR "Form field not found in current form"
  EXIT PROGRAM
END IF
```

Once instantiated, the `ui.ComboBox` object can be used, for example to fill the items of the drop down list.

```
CALL cb.clear()
CALL cb.addItem(1,"Paris")
CALL cb.addItem(2,"London")
CALL cb.addItem(3,"Madrid")
```

ui.ComboBox.addItem
Add an element to the item list.

Syntax

```
addItem(
   value STRING,
   label STRING )
```

1. *value* is the unique key that identifies the item.
2. *label* is the text to be displayed in the drop down list.

Usage

The `addItem()` method adds an item to the end of the drop down list of the `COMBOBOX`.

The first parameter is the value that can be set in the form field. The second parameter is the label to be displayed in the drop down list. If the second parameter is `NULL`,the runtime system automatically uses the first parameter as the display value.

Uniqueness is not checked by the runtime system. Make sure that the items created are unique, regarding the value key and the display label.

Trailing spaces should be avoided when populating the first parameter because values get truncated when field validation occurs, and the resulting value (without trailing spaces) will no longer match the `COMBOBOX` item name. Additionally, trailing spaces in the second parameter may cause the `COMBOBOX` to be much wider than expected. To avoid such problems, use `VARCHAR` or `STRING` variables, or use the `CLIPPED` operator with `CHAR` variables.

ui.ComboBox.clear
Clear the item list of a combobox.

Syntax

```
clear()
```

Usage

The `clear()` method clears the item list of the combobox.

If the item list is empty, the `COMBOBOX` drop-down button shows an empty list on the client side.

ui.ComboBox.getColumnName
Get the column name of the form field.

Syntax

```
getColumnName()
  RETURNING result STRING
```

Usage

The `getColumnName()` method returns the form field column name. The form field column name can be `NULL` if not defined at the form field level.

Use the `getTableName()` and `getColumnName()` methods together in order to identify the form field associated with the `COMBOBOX`. This allows to identify the combobox field in your program, for example to fill the drop down list with the appropriate items.

```
IF cb.getTableName()||"."||cb.getColumnName()
   == "customer.cust_city" THEN
      CALL cb.clear()
      CALL cb.addItem(1, "Paris" )
      CALL cb.addItem(2, "London" )
      CALL cb.addItem(3, "Madrid" )
END IF
```

ui.ComboBox.getIndexOf
Get an item position by name.

Syntax

```
getIndexOf(
   name STRING )
  RETURNING result INTEGER
```

1. *name* is the name of a combobox item.

Usage

The `getIndexOf()` method takes an item name as parameter and returns the position of the item in the drop down list.

The first item is at position 1. The method returns 0 (zero) if the item name does not exist.

The next example checks for item existence, before adding the item.

```
IF cb.getIndexOf("SFO") == 0 THEN
  CALL cb.addItem("SFO", "San Francisco International Airport, CA" )
END IF
```

ui.ComboBox.getItemCount
Get the number of items.

Syntax

```
getItemCount()
  RETURNING result INTEGER
```

Usage

The `getItemCount()` method returns the current number of items defined for the `COMBOBOX` form field.

The method returns 0 (zero) if no items are defined.

ui.ComboBox.getItemName
Get an item name by position.

Syntax

```
getItemName(
   position INTEGER )
  RETURNING result STRING
```

1. *position* is the index of the combobox item.

Usage

The `getItemName()` method returns the name of an item at the give position.

The first item starts at position 1.

ui.ComboBox.getItemText
Get the item text by position.

Syntax

```
getItemText(
   position INTEGER )
  RETURNING result STRING
```

1. *position* is the index of the combobox item.

Usage

The `getItemText()` method returns the display label of an item at the give position.

The first item starts at position 1.

ui.ComboBox.getTableName
Get the table prefix of the form field.

Syntax

```
getTableName()
  RETURNING result STRING
```

Usage

The `getTableName()` method returns the name of the form field table prefix. The form field table prefix can be `NULL` if not defined at the form field level.

This allows to identify a `COMBOBOX` field in your program, for example to fill the drop down list with the appropriate items.

ui.ComboBox.getTag
Get the combobox tag value.

Syntax

```
getTag()
  RETURNING result STRING
```

Usage

The `getTag()` method returns the value define by the `TAG` attribute.

Use the tag to mark `COMBOBOX` form items with your own flags, in order to adapt the configuration of the combobox dynamically by program. For example, if `TAG` contains the token "short", fill the drop down list with short names, otherwise fill with long names. The same code can then be used for different `COMBOBOX` form fields.

ui.ComboBox.getTextOf
Get the item text by name.

Syntax

```
getTextOf(
   name STRING )
  RETURNING result STRING
```

1. *name* is the name of a combobox item.

Usage

The `getTextOf()` method returns the display label of the item identified by the name passed as parameter.

The method returns `NULL` if the item name does not exist.

ui.ComboBox.removeItem
Remove an item by name.

Syntax

```
removeItem(
   name STRING )
```

1. *name* is the name of a combobox item.

Usage

The `removeItem()` method deletes an item from the list. The item to be removed is identified by the name passed as a parameter. If the item does not exist, the method returns without error.

Examples
Example Get a ComboBox form field view and fill the item list

Form Specification File:

```
DATABASE FORMONLY
LAYOUT
GRID
{
  Airport:  [cb01                ]
}
END
END
ATTRIBUTES
COMBOBOX cb01 = FORMONLY.airport TYPE CHAR;
END
```

Program File:

```
MAIN
  DEFINE cb ui.ComboBox
  DEFINE airport CHAR(3)
  OPEN FORM f1 FROM "combobox"
  DISPLAY FORM f1
  LET cb = ui.ComboBox.forName("formonly.airport")
  IF cb IS NULL THEN
     ERROR "Form field not found in current form"
     EXIT PROGRAM
```

```
  END IF
  CALL cb.clear()
  CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
  CALL cb.addItem("LCY", "London-City Airport, UK")
  CALL cb.addItem("LHR", "London-Heathrow, UK")
  CALL cb.addItem("FRA", "Frankfurt Airport, Germany")
  IF cb.getIndexOf("SFO") == 0 THEN
     CALL cb.addItem("SFO", "San Francisco International Airport, CA" )
  END IF
  INPUT BY NAME airport
END MAIN
```

Example Using the INITIALIZER attribute in the form file

Form Specification File:

```
DATABASE FORMONLY
LAYOUT
GRID
{
  Airport:  [cb01                ]
}
END
END
ATTRIBUTES
COMBOBOX cb01 = FORMONLY.airport TYPE CHAR, INITIALIZER=initcombobox;
END
```

Initialization function:

```
FUNCTION initcombobox(cb)
  DEFINE cb ui.ComboBox
  CALL cb.clear()
  CALL cb.addItem("CDG", "Paris-Charles de Gaulle, France")
  CALL cb.addItem("LCY", "London-City Airport, UK")
  CALL cb.addItem("LHR", "London-Heathrow, UK")
  CALL cb.addItem("FRA", "Frankfurt Airport, Germany")
  CALL cb.addItem("SFO", "San Francisco International Airport, CA" )
END FUNCTION
```

The DragDrop class

The `ui.DragDrop` class is used to control the events related to drag & drop events.

When implementing drag & drop in a dialog, the `ON DRAG*` / `ON DROP` dialog control blocks take a `ui.DragDrop` variable as a parameter to let you configure and control the drag & drop events. The `ui.DragDrop` variable must be declared in the scope of the dialog implementing drag & drop.

ui.DragDrop methods

Methods of the `ui.DragDrop` class.

Table 394: Object methods

Name	Description
`addPossibleOperation(`	Add a possible operation.

Name	Description
oper STRING)	
dropInternal()	Perform built-in row drop in trees.
getBuffer() RETURNING *result* STRING	Get drag & drop data from the buffer.
getLocationParent() RETURNING *result* INTEGER	Get the index of the parent node where the object was dropped.
getLocationRow() RETURNING *result* INTEGER	Get the index of the target row where the object was dropped.
getOperation() RETURNING *result* STRING	Identify the type of operation on drop.
getSelectedMimeType() RETURNING *result* STRING	Get the previously selected MIME type.
selectMimeType(*type* STRING)	Select the MIME type before getting the data.
setBuffer(*data* STRING)	Set the text data of the dragged object.
setFeedback(*type* STRING)	Define the appearance of the target during Drag & Drop.
setMimeType(*type* STRING)	Define the MIME type of the dragged object.
setOperation(*oper* STRING)	Define the type of Drag & Drop operation.

ui.DragDrop.addPossibleOperation
Add a possible operation.

Syntax

```
addPossibleOperation(
   oper STRING )
```

1. *oper* is the name of a drag & drop operation.

Usage

Drag & drop actions can be of different kinds; you can do a copy of the dragged object, or move the dragged object from the source to the destination.

The default drag & drop operation is defined by a call to `setOperation()` method in `ON DRAG_START`. Use the `addPossibleOperation()` method to define additional operations that are allowed.

See setOperation() for possible values.

ui.DragDrop.dropInternal
Perform built-in row drop in trees.

Syntax

```
dropInternal()
```

Usage

In order to simplify drag & drop programming in the same list, the `ui.DragDrop` class provides the `dropInternal()` utility method, to be called in the `ON DROP` block. This method will perform all the row changes in the array and move row selection as well as cell attributes.

When implementing drag & drop on a tree-view, dropping an element on the tree requires complex code in order to handle parent-child relationships. Nodes can be inserted under a parent between two children, appended at the end of the children list, and at different levels in the tree hierarchy. However, the `dropInternal()` method can also be used simple lists displayed in a regular `TABLE`.

A call to `dropInternal()` will silently be ignored, if the drag source is not the drop target, or if the method is called in a different context as `ON DROP`.

For more details about dropping elements in tree-views, see Drag & drop on page 1411.

ui.DragDrop.getBuffer
Get drag & drop data from the buffer.

Syntax

```
getBuffer()
  RETURNING result STRING
```

Usage

After identifying the MIME type of a dropped object with `getSelectedMimeType()`, you can call the `getBuffer()` method to get text data from the drag & drop buffer.

Drag & drop data is only available at `ON DROP` time, therefore, the `getBuffer()` method must be called in `ON DROP` only.

ui.DragDrop.getLocationParent
Get the index of the parent node where the object was dropped.

Syntax

```
getLocationParent()
  RETURNING result INTEGER
```

Usage

When using a tree view, a node can be dropped as a sibling or as a child node to another node. In order to distinguish between the cases, you must use the `getLocationParent()` method, which returns the index of the parent node of the drop target node returned by `getLocationRow()`.

If both methods return the same row index, you must append the dropped row as a child of the target node. Otherwise, `getLocationParent()` identifies the parent node where the dropped row has to be added as a child, and `getLocationRow()` is the index of a sibling node. In the last case the dropped node must be inserted before the node identified by `getLocationRow()`.

These methods are typically used in the `ON DROP` block, but can also be used in `ON DRAG_OVER` to deny the drop according to the indexes returned; for example, the program might only allow the drop of objects as new children for a given parent node.

ui.DragDrop.getLocationRow
Get the index of the target row where the object was dropped.

Syntax

```
getLocationRow()
  RETURNING result INTEGER
```

Usage

The `getLocationRow()` method returns the index of the row in the drop target list pointed to by the mouse cursor.

This method is typically used in the `ON DROP` block to get the index of the target row to be modified or replaced by the dragged object.

The method can also be used in `ON DRAG_OVER` to deny the drop according to the current target row returned by `getLocationRow()`

ui.DragDrop.getSelectedMimeType
Get the previously selected MIME type.

Syntax

```
getSelectedMimeType()
  RETURNING result STRING
```

Usage

Before retrieving data from the drag & drop buffer with `getBuffer()`, first call the `getSelectedMimeType()` method to identify the data format that was previously selected by a `selectMimeType()` call.

The `getSelectedMimeType()` method is typically called in `ON DROP` to identity the format of the dropped object.

ui.DragDrop.getOperation
Identify the type of operation on drop.

Syntax

```
getOperation()
  RETURNING result STRING
```

Usage

The `getOperation()` method returns the type of the current drag & drop operation ("`copy`", "`move`", or "`none`").

According to the value returned by this method, the program can make the appropriate changes in the data model. For example, after a row has been dropped into another list, the source list can remove the original row if the operation was a "`move`", but keeps the original row if the operation was a "`copy`".

The `getOperation()` method is typically called in the `ON DRAG_FINISHED` block.

ui.DragDrop.selectMimeType
Select the MIME type before getting the data.

Syntax

```
selectMimeType(
   type STRING )
```

1. *type* defines the MIME type for dragged objects.

Usage

Call the `selectMimeType()` method to check that data is available in a format identified by the MIME type passed as parameter.

If this type of data is available in the buffer, the method returns `TRUE` and you can later get the data with `getBuffer()`.

The `selectMimeType()` method is typically used in `ON DRAG_ENTER`, `ON DRAG_OVER` to deny the drag & drop operation if none of the supported MIME types is available in the buffer.

ui.DragDrop.setBuffer
Set the text data of the dragged object.

Syntax

```
setBuffer(
   data STRING )
```

1. *data* is a string expression containing drag & drop data.

Usage

Use the `setBuffer()` method to provide the text data of objects dragged from the program to an external application.

The `setBuffer()` method is typically used in an `ON DRAG_START` block in conjunction with `setMimeType()`.

By default, the dialog will serialize the data of the selected rows as a tab-separated list of values.

The text/plain MIME type is the default.

ui.DragDrop.setFeedback
Define the appearance of the target during Drag & Drop.

Syntax

```
setFeedback(
   type STRING )
```

1. *type* is the type of feedback to display during the drag & drop operation.

Usage

The `setFeedback()` method defines the appearance the target object must have during the drag & drop process.

For example, in a table or tree view, when the mouse is flying over rows in the drop target, a different visual indicator will appear according to the value that was passed to `setFeedback()`.

Possible values for the `setFeedback()` method are:

Table 395: Values for the `setFeedback()` method

Parameter Values	Description
`all`	Dragged object will be dropped somewhere on the target widget, the exact location does not matter.
`insert`	In lists, dragged object will be inserted in between existing rows.
`select`	In lists, dragged object will replace the current row under the mouse.

ui.DragDrop.setMimeType
Define the MIME type of the dragged object.

Syntax

```
setMimeType(
   type STRING )
```

1. *type* defines the MIME type for the drag & drop buffer.

Usage

Objects dragged from the program to an external application need to be identified with a MIME type and the program must provide the data. The MIME type can be specified with the `setMimeType()` method.

The `setMimeType()` method is typically used in an `ON DRAG_START` block in conjunction with `setBuffer()`.

By default, the source target will use the text/plain MIME type and copy the data of the selected rows into the Drag & Drop buffer.

ui.DragDrop.setOperation
Define the type of Drag & Drop operation.

Syntax

```
setOperation(
   oper STRING )
```

1. *oper* is the name of a drag & drop operation.

Usage

Drag & drop actions can be of different kinds; you can do a copy of the dragged object, or move the dragged object from the source to the destination.

Use the `setOperation()` method to define/force the type of drag & drop operation or to deny/cancel the drag & drop process.

Table 396: Parameters for the `setOperation()` method

Parameter Value	Description
`NULL`	To deny/cancel the drag & drop process.
`copy`	To allow drag & drop as a copy of the source object.
`move`	To allow drag & drop as a move of the source object.

The `setOperation()` method can be called in different drag & drop triggers.

A common usage is to deny drag & drop by passing `NULL` in the `ON DRAG_ENTER` and/or `ON DRAG_OVER` blocks because the dragged object does not correspond to the type of objects the target can receive.

This method is also used in `ON DRAG_START` to force a specific type of drag & drop operation (copy or move), or to deny drag start if the context does not allow a drag & drop action.

When called in the `ON DRAG_ENTER` block, the method forces a specific drag & drop operation.

The om package

These topics cover the built-in classes for the om class

- The DomDocument class on page 1833
- The DomNode class on page 1839
- The NodeList class on page 1858
- The SaxAttributes class on page 1860
- The SaxDocumentHandler class on page 1865
- The XmlReader class on page 1871
- The XmlWriter class on page 1876

The DomDocument class

The `om.DomDocument` class provides methods to manipulate a data tree, following the DOM standards.

A `om.DomDocument` object holds a DOM tree of `om.DomNode` objects.

A unique root `om.DomNode` object is owned by a `om.DomDocument` object.

om.DomDocument methods

Methods of the `om.DomDocument` class.

Table 397: Class methods

Name	Description

`om.DomDocument.create(`
`  tag STRING )`
`  RETURNING result om.DomDocument`
	Create a new empty `om.DomDocument` object.

`om.DomDocument.createFromString(` `  string STRING )`	Create a new `om.DomDocument` object from an XML string.

Name	Description
`RETURNING result om.DomDocument`	
`om.DomDocument.createFromXmlFile( filename STRING ) RETURNING result om.DomDocument`	Create a new `om.DomDocument` object from an XML file.

Table 398: Object methods

Name	Description
`createChars( string STRING ) RETURNING result om.DomNode`	Create a new text node in the DOM document.
`createElement( tag STRING ) RETURNING result om.DomNode`	Create a new element node in the DOM document.
`createEntity( name STRING ) RETURNING result om.DomNode`	Create a new entity node in the DOM document.
`copy( source om.DomNode, deep INTEGER ) RETURNING result om.DomNode`	Create a new element node by copying an existing node.
`getDocumentById( id INTEGER ) RETURNING result om.DomNode`	Returns a node element according to the internal AUI tree id.
`getDocumentElement() RETURNING result om.DomNode`	Returns the root node element of the DOM document.
`removeElement( element om.DomNode )`	Remove a DomNode object and all its descendants.

om.DomDocument.create
Create a new empty `om.DomDocument` object.

Syntax

```
om.DomDocument.create(
   tag STRING )
  RETURNING result om.DomDocument
```

1. *tag* defines the tag name of the root element.

Usage

Use the class method `om.DomDocument.create()` to instantiate a new, empty DOM document object.

To hold the reference to a DOM document object, define a variable with the type `om.DomDocument` type.

Example

```
DEFINE d  om.DomDocument
LET d = om.DomDocument.create("Vehicles")
```

om.DomDocument.createFromString
Create a new `om.DomDocument` object from an XML string.

Syntax

```
om.DomDocument.createFromString(
   string STRING )
  RETURNING result om.DomDocument
```

1. *string* is the string expression containing XML data.

Usage

Use the class method `om.DomDocument.createFromString()` to instantiate a new DomDocument object that is filled with the content of the specified XML formatted string.

To hold the reference to a DOM document object, define a variable with the type `om.DomDocument` type.

Example

```
DEFINE d  om.DomDocument
LET d = om.DomDocument.createFromString("<Vehicles/>")
```

om.DomDocument.createFromXmlFile
Create a new `om.DomDocument` object from an XML file.

Syntax

```
om.DomDocument.createFromXmlFile(
   filename STRING )
  RETURNING result om.DomDocument
```

1. *filename* is the path to the file containing XML data.

Usage

Use the class method `om.DomDocument.createFromXmlFile()` to instantiate a new DomDocument object that is filled with the content of the specified XML file.

To hold the reference to a DOM document object, define a variable with the type `om.DomDocument` type.

Example

```
DEFINE d  om.DomDocument
LET d = om.DomDocument.createFromXmlFile("vehicles.xml")
```

om.DomDocument.getDocumentElement
Returns the root node element of the DOM document.

Syntax

```
getDocumentElement()
  RETURNING result om.DomNode
```

Usage

The method `getDocumentElement()` returns the root `om.DomNode` element node of the DOM document.

To hold the reference to the root node, define a variable with the type `om.DomNode` type.

Example

```
DEFINE n om.DomNode
LET n = mydoc.getDocumentElement()
```

om.DomDocument.getElementById
Returns a node element according to the internal AUI tree id.

Syntax

```
getDocumentById(
   id INTEGER )
  RETURNING result om.DomNode
```

Usage

The method `getElementById()` returns the `om.DomNode` element of the DOM document according to the internal id number passed as parameter.

Each DOM node gets an internal integer id when it is created in the abstract user interface tree, and can be referenced by this unique id. The node id is typically used in other nodes, to reference a node in the DOM document.

To hold the reference to the root node, define a variable with the type `om.DomNode` type.

Example

```
DEFINE n om.DomNode
LET n = mydoc.getElementById()
```

om.DomDocument.createChars
Create a new text node in the DOM document.

Syntax

```
createChars(
   string STRING )
  RETURNING result om.DomNode
```

1. *string* defines the content of the text node.

Usage

Use the method `createChars()` to create a new `om.DomNode` text node. The content of the text node must be passed as parameter.

The new created node will have the reserved tagName "`@chars`", and a single attribute named "`@chars`" storing the character data.

To hold the reference to the new node, define a variable with the type `om.DomNode` type.

Example

```
MAIN
  DEFINE mydoc  om.DomDocument
  DEFINE root, text om.DomNode
  LET mydoc = om.DomDocument.create("Test")
  LET root = mydoc.getDocumentElement()
  LET text = mydoc.createChars("Hello, world!")
  DISPLAY text.getAttribute("@chars")
  CALL root.appendChild(text)
  CALL root.writeXML("output.xml")
END MAIN
```

om.DomDocument.createElement
Create a new element node in the DOM document.

Syntax

```
createElement(
   tag STRING )
  RETURNING result om.DomNode
```

1. *tag* defines the tag name of the node.

Usage

Use the method `createElement()` to create a new `om.DomNode` element node. The tag name of the element must be passed as parameter.

To hold the reference to the new node, define a variable with the type `om.DomNode` type.

Example

```
DEFINE n om.DomNode
LET n = mydoc.createElement("Car")
```

om.DomDocument.createEntity
Create a new entity node in the DOM document.

Syntax

```
createEntity(
   name STRING )
  RETURNING result om.DomNode
```

1. *name* defines the name of the entity node.

Usage

Use the method `createEntity()` to create a new `om.DomNode` entity node. The entity name must be passed as parameter.

The text representation of a entity node is *&name;*.

The new created node will have the reserved tagName "`@entity`" , with a single attribute named "`@entity`" containing the text of the entity.

To hold the reference to the new node, define a variable with the type `om.DomNode` type.

Example

```
DEFINE n om.DomNode
LET n = mydoc.createEntity("quot")
```

om.DomDocument.copy
Create a new element node by copying an existing node.

Syntax

```
copy(
   source om.DomNode,
   deep INTEGER )
  RETURNING result om.DomNode
```

1. *source* references the source node to copy.
2. *deep* is a boolean to control the recursive node copy.

Usage

Use the method `copy()` to create a new `om.DomNode` element node from an existing node.

Pass `TRUE` as second parameter to clone a complete tree of nodes.

To hold the reference to the new node, define a variable with the type `om.DomNode` type.

Example

```
DEFINE n, s om.DomNode
LET s = mydoc.createElement("Car")
LET n = mydoc.copy(s, TRUE)
```

om.DomDocument.removeElement
Remove a DomNode object and all its descendants.

Syntax

```
removeElement(
   element om.DomNode )
```

1. *element* is the DOM node to be removed.

Usage

Use the `removeElement()` method to remove an element and all its descendants from DOM document.

Any reference to the removed `om.DomNode` objects becomes invalid.

Examples
Example 1: Creating a DOM document

```
MAIN
  DEFINE d  om.DomDocument
  DEFINE r om.DomNode
  LET d = om.DomDocument.create("MyDocument")
  LET r = d.getDocumentElement()
END MAIN
```

The DomNode class
The `om.DomNode` class provides methods to manipulate a DOM node of a data tree.

This class follows the DOM standards.

A DomNode object is a node (or element) of a DomDocument.

Tag and attribute names of DOM nodes are case sensitive; "Wheel" is not the same as "wheel".

Text nodes cannot have attributes, but they have plain text. In text nodes, the characters can be accessed with the `@chars` attribute name. In XML representation, a text node is the text itself. Do not confuse it with the parent node. For example, `<Item id="32">Red shoes</Item>` represents 2 nodes: The parent 'Item' node and a text node with string 'Red shoes'.

If you need to identify an element, use a common attribute like "name". If you need to label an element, use a common attribute like "text".

om.DomNode methods
Methods of the `om.DomNode` class.

Table 399: Object methods: Node creation

Name	Description
`appendChild( node om.DomNode )`	Adds an existing node at the end of the list of children in the current node.
`createChild( tag STRING ) RETURNING result om.DomNode`	Creates and adds an node at the end of the list of children in the current node.
`insertBefore( new om.DomNode, existing om.DomNode)`	Inserts an existing node before the existing node specified.
`removeChild( node om.DomNode )`	Deletes the specified child node from the current node.
`replaceChild( new om.DomNode, old om.DomNode)`	Replaces a node by another in the children nodes of the current node.

Table 400: Object methods: In/Out

Name	Description
`loadXml( filename STRING ) RETURNING result om.DomNode`	Load an XML file into the current node.
`parse( string STRING ) RETURNING result om.DomNode`	Parses an XML formatted string and creates the DOM structure in the current node.
`toString() RETURNING result STRING`	Serializes the current node into an XML formatted string.
`write( sdh om.SaxDocumentHandler )`	Processes a DOM document with a SAX document handler.
`writeXml( filename STRING )`	Creates an XML file from the current DOM node.

Table 401: Object methods: Node identification

Name	Description
`getId() RETURNING result INTEGER`	Returns the internal AUI tree id of a DOM node.
`getTagName() RETURNING result STRING`	Returns the XML tag name of a DOM node.

Table 402: Object methods: Attributes management

Name	Description
`getAttribute( name STRING ) RETURNING result STRING`	Returns the value of a DOM node attribute.
`getAttributesCount() RETURNING result INTEGER`	Returns the number of attributes in the DOM node.
`getAttributeInteger( name STRING, def STRING )`	Returns the value of a DOM node attribute, with default integer value.

Name	Description
`RETURNING result INTEGER`	
`getAttributeString( name STRING, def STRING ) RETURNING result STRING`	Returns the value of a DOM node attribute, with default string value.
`getAttributeName( index INTEGER ) RETURNING result STRING`	Returns the name of a DOM node attribute by position.
`getAttributeValue( index INTEGER ) RETURNING result STRING`	Returns the value of a DOM node attribute by position.
`setAttribute( name STRING, value STRING )`	Sets the value of a DOM node attribute.
`removeAttribute( name STRING ) RETURNING result STRING`	Delete the specified attribute from the DOM node.

Table 403: Object methods: Tree navigation

Name	Description
`getChildByIndex( index INTEGER ) RETURNING result om.DomNode`	Returns a child DOM node by position.
`getChildCount() RETURNING result INTEGER`	Returns the number of children nodes.
`getFirstChild() RETURNING result om.DomNode`	Returns the first child DOM node.
`getLastChild() RETURNING result om.DomNode`	Returns the last child DOM node.
`getNext() RETURNING result om.DomNode`	Returns the next sibling DOM node of this node.
`getParent()`	Returns the parent DOM node.

Name	Description
RETURNING *result* om.DomNode	
getPrevious() RETURNING *result* om.DomNode	Returns previous sibling DOM node of this node.
selectByPath(*xpath* STRING) RETURNING *result* om.NodeList	Finds descendant DOM nodes according to an XPath-like pattern.
selectByTagName(*tagname* STRING) RETURNING *result* om.NodeList	Finds descendant DOM nodes according to a tag name.

om.DomNode.appendChild
Adds an existing node at the end of the list of children in the current node.

Syntax

```
appendChild(
   node om.DomNode )
```

1. *node* is a reference to a node.

Usage

The `appendChild()` method takes an existing `om.DomNode` element node and adds it at the end of the children of the object node calling the method.

The child node passed to the `appendChild()` method must have been created from the same DOM document object, for example with the `om.DomDocument.createElement()` method.

If the node passed to the `appendChild()` method is already attached to another parent node, it will be detached from that parent node before being attached to the new parent node.

Example

```
MAIN
    DEFINE doc  om.DomDocument,
           r om.DomNode,
           p1, p2 om.DomNode,
           c1, c2 om.DomNode

    LET doc = om.DomDocument.create("Items")

    LET r = doc.createElement("Zoo")

    LET p1 = doc.createElement("DodoList")

    -- appends p1 under r
    CALL r.appendChild(p1)

    LET c1 = doc.createElement("Dodo")
    CALL c1.setAttribute("name", "momo")
    CALL c1.setAttribute("gender", "male")
    CALL p1.appendChild(c1)
```

```
    LET p2 = doc.createElement("DodoList")
    CALL r.appendChild(p2)
    LET c2 = doc.createElement("Dodo")
    CALL c2.setAttribute("name", "kiki")
    CALL c2.setAttribute("gender", "female")
    CALL p2.appendChild(c2)

    CALL r.writeXml("file1.xml")

    -- moves c1 under p2
    CALL p2.appendChild(c1)

    CALL r.writeXml("file2.xml")
END MAIN
```

The above program will produce the following XML files:

`file.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList>
    <Dodo name="momo" gender="male"/>
  </DodoList>
  <DodoList>
    <Dodo name="kiki" gender="female"/>
  </DodoList>
</Zoo>
```

`file2.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList/>
  <DodoList>
    <Dodo name="kiki" gender="female"/>
    <Dodo name="momo" gender="male"/>
  </DodoList>
</Zoo>
```

om.DomNode.createChild
Creates and adds an node at the end of the list of children in the current node.

Syntax

```
createChild(
   tag STRING )
  RETURNING result om.DomNode
```

1. *tag* is the tag name of the new node.

Usage

The `createChild()` method creates a new `om.DomNode` element with the tag name passed as parameter, and adds it at the end of the children of the object node calling the method.

The method returns the reference to the new created object.

Example

```
DEFINE parent, child om.DomNode
...
LET child = parent.createChild("Item")
```

om.DomNode.insertBefore
Inserts an existing node before the existing node specified.

Syntax

```
insertBefore(
   new om.DomNode,
   existing om.DomNode)
```

1. *new* is a reference to a new created node.
2. *existing* is a reference to a child node existing in the current node.

Usage

The `insertBefore()` method takes an existing `om.DomNode` element node and adds it before the child node passed as second parameter, in the list of children of the object node calling the method.

The child node passed to the `insertChild()` method must have been created from the same DOM document object, for example with the `om.DomDocument.createElement()` method.

Example

```
DEFINE parent, other, child om.DomNode
...
LET child = mydoc.createElement("Item")
CALL parent.insertBefore(child, other)
```

om.DomNode.loadXml
Load an XML file into the current node.

Syntax

```
loadXml(
   filename STRING )
  RETURNING result om.DomNode
```

1. *filename* is the path to the XML file.

Usage

The `loadXml()` method takes a file path as parameter and loads the XML content into the current node, by creating a new DOM structure in memory. The method then returns the new created child DOM node.

To hold the reference to the new node, define a variable with the type `om.DomNode` type.

Example

```
DEFINE parent, new om.DomNode
...
LET new = parent.loadXml("myfile.xml")
```

om.DomNode.parse
Parses an XML formatted string and creates the DOM structure in the current node.

Syntax

```
parse(
   string STRING )
  RETURNING result om.DomNode
```

1. *string* is an XML formatted string.

Usage

The `parse()` method scans the XML formatted string passed as parameter and creates the corresponding DOM nodes into the current node. The method then returns the new created child DOM node.

The node must be created before it is passed as parameter to this method, typically, with `om.DomDocument.createElement()`.

Example

```
DEFINE parent, child om.DomNode
...
LET child = parent.parse("<Item/>")
```

om.DomNode.getAttribute
Returns the value of a DOM node attribute.

Syntax

```
getAttribute(
   name STRING )
  RETURNING result STRING
```

1. *name* is the name of the attribute.

Usage

The `getAttribute()` method returns the value of the attribute passed as parameter, as defined in the current node.

DOM node attribute names are case-sensitive.

If the attribute does not exist for this node type, or if the attribute is not set, the method returns `NULL`.

For character nodes (created for example with the `createChars()` of a `DomDocument` object), you can get the text value by passing the `@chars` attribute name to the method.

Example

```
DEFINE node om.DomNode
...
DISPLAY node.getAttribute("color")
```

om.DomNode.getAttributeInteger
Returns the value of a DOM node attribute, with default integer value.

Syntax

```
getAttributeInteger(
   name STRING,
   def STRING )
  RETURNING result INTEGER
```

1. *name* is the name of the attribute.
2. *def* is the default value.

Usage

The `getAttributeInteger()` method returns the value of the attribute passed as parameter, as defined in the current node.

DOM node attribute names are case-sensitive.

If the attribute is not defined, the method returns the default value passed as second parameter.

om.DomNode.getAttributesCount
Returns the number of attributes in the DOM node.

Syntax

```
getAttributesCount()
  RETURNING result INTEGER
```

Usage

The `getAttributesCount()` method returns the number of attributes defined in the current node.

This method is typically used to scan all the attributes of a node by position, with the `getAttributeName()` and `getAttributeValue()` methods.

Example

```
DEFINE node om.DomNode,
       index INTEGER
...
FOR index = 1 TO node.getAttributesCount()
    DISPLAY node.getAttributeName(index)
END FOR
```

om.DomNode.getAttributeString
Returns the value of a DOM node attribute, with default string value.

Syntax

```
getAttributeString(
   name STRING,
   def STRING )
  RETURNING result STRING
```

1. *name* is the name of the attribute.
2. *def* is the default value.

Usage

The `getAttributeString()` method returns the value of the attribute passed as parameter, as defined in the current node.

DOM node attribute names are case-sensitive.

If the attribute is not defined, the method returns the default value passed as second parameter.

om.DomNode.getAttributeName
Returns the name of a DOM node attribute by position.

Syntax

```
getAttributeName(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the index of the attribute, starts at 1.

Usage

The `getAttributeName()` method returns the name of an attribute by position in the current node.

DOM node attribute names are case-sensitive.

If the attribute does not exist at the given position, the method returns `NULL`.

Example

```
DEFINE node om.DomNode
...
DISPLAY node.getAttributeName(12)
```

om.DomNode.getAttributeValue
Returns the value of a DOM node attribute by position.

Syntax

```
getAttributeValue(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the index of the attribute, starts at 1.

Usage

The `getAttributeValue()` method returns the value of an attribute by position in the current node.

DOM node attribute names are case-sensitive.

If the attribute does not exist at the given position, the method returns `NULL`.

Example

```
DEFINE node om.DomNode
...
DISPLAY node.getAttributeValue(12)
```

om.DomNode.getChildByIndex
Returns a child DOM node by position.

Syntax

```
getChildByIndex(
   index INTEGER )
  RETURNING result om.DomNode
```

1. *index* is the index of the child node, starts at 1.

Usage

The `getChildByIndex()` method returns the child DOM node by position in the current node.

If there is no child node at the give position, the method returns `NULL`.

om.DomNode.getChildCount
Returns the number of children nodes.

Syntax

```
getChildCount()
  RETURNING result INTEGER
```

Usage

The `getChildCount()` method returns the number of children nodes in the current node.

This method is typically used to scan the children nodes of a DOM node, with the `getChildByIndex()` method.

Example

```
DEFINE parent, child om.DomNode,
       index INTEGER
...
FOR index=1 TO node.getChildCount()
    LET child = parent.getChildByIndex(index)
    ...
END FOR
```

om.DomNode.getFirstChild
Returns the first child DOM node.

Syntax

```
getFirstChild()
  RETURNING result om.DomNode
```

Usage

The `getFirstChild()` method returns the first child DOM node in the current node.

This method is typically used to scan children nodes with the `getNext()` method, until `getNext()` returns `NULL`.

Example

```
DEFINE parent, child om.DomNode
...
LET child = parent.getFirstChild()
WHILE child IS NOT NULL
    ...
    LET child = child.getNext()
END WHILE
```

om.DomNode.getId
Returns the internal AUI tree id of a DOM node.

Syntax

```
getId()
  RETURNING result INTEGER
```

Usage

The `getId()` method returns to internal integer identifier generated automatically for any `om.DomNode` object created in the abstract user interface tree.

The internal id is typically used to reference a DOM node in an attribute of another node, to link logically nodes together.

If the DOM node does not belong to the AUI tree, the method returns zero.

om.DomNode.getLastChild
Returns the last child DOM node.

Syntax

```
getLastChild()
  RETURNING result om.DomNode
```

Usage

The `getLastChild()` method returns the last child DOM node in the current node.

This method is typically used to scan children nodes with the `getPrevious()` method, until `getPrevious()` returns `NULL`.

Example

```
DEFINE parent, child om.DomNode
...
LET child = parent.getLastChild()
WHILE child IS NOT NULL
    ...
    LET child = child.getPrevious()
END WHILE
```

om.DomNode.getNext
Returns the next sibling DOM node of this node.

Syntax

```
getNext()
  RETURNING result om.DomNode
```

Usage

The `getNext()` method returns the next sibling DOM node following the current node, withing the children list of the parent node.

om.DomNode.getParent
Returns the parent DOM node.

Syntax

```
getParent()
  RETURNING result om.DomNode
```

Usage

The `getParent()` method returns the parent DOM node of the current node.

If the current node is the root node, the method returns `NULL`.

Example

```
DEFINE parent, current om.DomNode
...
LET parent = current.getParent()
```

om.DomNode.getPrevious
Returns previous sibling DOM node of this node.

Syntax

```
getPrevious()
  RETURNING result om.DomNode
```

Usage

The `getPrevious()` method returns the previous sibling DOM node preceding the current node, withing the children list of the parent node.

om.DomNode.getTagName
Returns the XML tag name of a DOM node.

Syntax

```
getTagName()
  RETURNING result STRING
```

Usage

The `getTagName()` method returns the XML tag name of the node.

Use this method to identity the type of the node.

om.DomNode.removeAttribute
Delete the specified attribute from the DOM node.

Syntax

```
removeAttribute(
   name STRING )
  RETURNING result STRING
```

1. *name* is the name of the attribute.

Usage

The `removeAttribute()` method deletes the attribute identified by the name passed as parameter.

DOM node attribute names are case-sensitive.

If the attribute does not exist for this node the method returns silently.

Example

```
DEFINE node om.DomNode
...
CALL node.removeAttribute("comments")
```

om.DomNode.removeChild
Deletes the specified child node from the current node.

Syntax

```
removeChild(
   node om.DomNode )
```

1. *node* is a reference to a node.

Usage

The `removeChild()` method detaches a `om.DomNode` element node from the current node.

The removed node is not destroyed, if it is still referenced by a variable: The `removeChild()` method will only break the link between the parent node and the child node. The child node still exists in the DOM document, but it is an orphan node, that can be attached to another parent node in the document.

Example

```
MAIN
    DEFINE doc  om.DomDocument,
           r om.DomNode,
           p om.DomNode,
           c om.DomNode

    LET doc = om.DomDocument.create("Items")

    LET r = doc.createElement("Zoo")
```

```
    LET p = doc.createElement("DodoList")
    CALL r.appendChild(p)

    LET c = doc.createElement("Dodo")
    CALL c.setAttribute("name", "momo")
    CALL c.setAttribute("gender", "male")
    CALL p.appendChild(c)

    CALL r.writeXml("file1.xml")

    CALL p.removeChild(c)

    -- c is orphan but still exists
    CALL c.writeXml("file2.xml")
    LET c = NULL -- unref/destroy the node

    CALL r.writeXml("file3.xml")
END MAIN
```

The above program will produce the following files:

`file1.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList>
    <Dodo name="momo" gender="male"/>
  </DodoList>
</Zoo>
```

`file2.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Dodo name="momo" gender="male"/>
```

`file3.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList/>
</Zoo>
```

om.DomNode.replaceChild
Replaces a node by another in the children nodes of the current node.

Syntax

```
replaceChild(
   new om.DomNode,
   old om.DomNode)
```

1. *new* is a reference to the new node.
2. *old* is the the node to be replaced.

Usage

The `replaceChild()` nethod puts the `om.DomNode` element passed as first parameter at the place of the node referenced by the second parameter, in the children list of the object node calling the method.

The new child node passed to the `replaceChild()` method must have been created from the same DOM document object, for example with the `om.DomDocument.createElement()` method.

The old node is not destroyed, if it is still referenced by a variable. The old node still exists in the DOM document, but it is an orphan node, that can be attached to another parent node in the document.

Example

```
MAIN
    DEFINE doc  om.DomDocument,
           r om.DomNode,
           p om.DomNode,
           o om.DomNode,
           n om.DomNode

    LET doc = om.DomDocument.create("Items")

    LET r = doc.createElement("Zoo")

    LET p = doc.createElement("DodoList")
    CALL r.appendChild(p)

    LET o = doc.createElement("Dodo")
    CALL o.setAttribute("name", "momo")
    CALL o.setAttribute("gender", "male")
    CALL p.appendChild(o)

    CALL r.writeXml("file1.xml")

    LET n = doc.createElement("Dodo")
    CALL n.setAttribute("name", "kiki")
    CALL n.setAttribute("gender", "female")

    CALL p.replaceChild(n, o)

    -- o is orphan but still exists
    CALL o.writeXml("file2.xml")
    LET o = NULL -- unref/destroy the node

    CALL r.writeXml("file3.xml")
END MAIN
```

The above program will produce following files:

`file1.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
  <DodoList>
    <Dodo name="momo" gender="male"/>
  </DodoList>
</Zoo>
```

`file2.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Dodo name="momo" gender="male"/>
```

`file3.xml`

```
<?xml version='1.0' encoding='ASCII'?>
<Zoo>
```

```
  <DodoList>
    <Dodo name="kiki" gender="female"/>
  </DodoList>
</Zoo>
```

om.DomNode.setAttribute
Sets the value of a DOM node attribute.

Syntax

```
setAttribute(
   name STRING,
   value STRING )
```

1. *name* is the name of the attribute.
2. *value* is the attribute value.

Usage

The `setAttribute()` method sets the value of an attribute in the current node.

DOM node attribute names are case-sensitive.

> **Note:** Make sure that the strings passed to the method do not contain illegal XML characters: Illegal XML characters will be silently ignored. Illegal XML characters are any character below space (ASCII 32), except \r (ASCII 13), \n (ASCII 10) and \t (ASCII 9).

Example

```
DEFINE node om.DomNode
...
CALL node.setAttribute("name", "tiger")
```

om.DomNode.toString
Serializes the current node into an XML formatted string.

Syntax

```
toString()
  RETURNING result STRING
```

Usage

The `toString()` method builds an XML formatted string with the DOM structure of the current node and returns the string.

Example

```
DEFINE node om.DomNode, s STRING
...
LET s = node.toString()
```

om.DomNode.write
Processes a DOM document with a SAX document handler.

Syntax

```
write(
   sdh om.SaxDocumentHandler )
```

1. *sdh* references a SAX document handler.

Usage

The `write()` method processes the current DOM node content with the SAX document handler passed as parameter.

See the SAX document handler class for more details.

om.DomNode.writeXml
Creates an XML file from the current DOM node.

Syntax

```
writeXml(
   filename STRING )
```

1. *filename* is the path to the XML file.

Usage

The `writeXml()` method writes the content of the current DOM node to the file passed as parameter.

Example

```
DEFINE node om.DomNode
...
CALL noe.writeXml("output.xml")
```

om.DomNode.selectByPath
Finds descendant DOM nodes according to an XPath-like pattern.

Syntax

```
selectByPath(
   xpath STRING )
  RETURNING result om.NodeList
```

1. *xpath* is an XPath-like pattern, using .

Usage

The `selectByPath()` method scans the DOM tree for descendant nodes according to the specified XPath-like pattern.

The pattern supported is limited to the following syntax:

```
{ / | // } TagName [ [@AttributeName="Value"] ] [...]
```

DOM node tag names and attributes names are case-sensitive.

The method creates a list of nodes as a `om.NodeList` object. This list object is then used to process the nodes found.

Example

```
DEFINE node om.DomNode,
       nodelist om.NodeList
...
LET nodelist = node.selectByPath("//Grid/Table[@tabName=
\"t1\"]")
```

om.DomNode.selectByTagName
Finds descendant DOM nodes according to a tag name.

Syntax

```
selectByTagName(
   tagname STRING )
  RETURNING result om.NodeList
```

1. *tagname* is a tag name for the search.

Usage

The `selectByTagName()` method scans the DOM tree for descendant nodes defined with the tag name specified as parameter.

DOM node tag names are case-sensitive.

The method creates a list of nodes as a `om.NodeList` object. This list object is then used to process the nodes found.

Example

```
DEFINE node om.DomNode,
       nodelist om.NodeList
...
LET nodelist = node.selectByTagName("Car")
```

Examples
Example 1: Creating a DOM tree

To create a DOM tree with the following structure (represented in XML format):

```
<Vehicles>
   <Car name="Corolla" color="Blue" weight="1546">Nice car!</Car>
   <Bus name="Maxibus" color="Yellow" weight="5278">
      <Wheel width="315" diameter="925" />
      <Wheel width="315" diameter="925" />
      <Wheel width="315" diameter="925" />
      <Wheel width="315" diameter="925" />
   </Bus>
</Vehicles>
```

You write the following:

```
MAIN
  DEFINE d  om.DomDocument
  DEFINE r, n, t, w om.DomNode
```

```
  DEFINE i INTEGER

  LET d = om.DomDocument.create("Vehicles")
  LET r = d.getDocumentElement()

  LET n = r.createChild("Car")
  CALL n.setAttribute("name","Corolla")
  CALL n.setAttribute("color","Blue")
  CALL n.setAttribute("weight","1546")

  LET t = d.createChars("Nice car!")
  CALL n.appendChild(t)
  LET t = d.createEntity("nbsp")
  CALL n.appendChild(t)
  LET t = d.createChars("Yes, very nice!")
  CALL n.appendChild(t)

  LET n = r.createChild("Bus")
  CALL n.setAttribute("name","Maxibus")
  CALL n.setAttribute("color","yellow")
  CALL n.setAttribute("weight","5278")
  FOR i=1 TO 4
    LET w = n.createChild("Wheel")
    CALL w.setAttribute("width","315")
    CALL w.setAttribute("diameter","925")
  END FOR

  CALL r.writeXml("Vehicles.xml")

END MAIN
```

Example 2: Displaying a DOM tree recursively

The following example displays a DOM tree content recursively:

```
FUNCTION displayDomNode(n,e)
  DEFINE n om.DomNode
  DEFINE e, i, s INTEGER

  LET s = e*2
  DISPLAY s SPACES || "Tag: " || n.getTagName()

  DISPLAY s SPACES || "Attributes:"
  FOR i=1 TO n.getAttributesCount()
     DISPLAY s SPACES || "  " || n.getAttributeName(i) ||
                     "=[" || n.getAttributeValue(i) ||"]"
  END FOR
  LET n = n.getFirstChild()

  DISPLAY s SPACES || "Child Nodes:"
  WHILE n IS NOT NULL
    CALL displayDomNode(n,e+1)
    LET n = n.getNext()
  END WHILE

END FUNCTION
```

Example 3: Writing a DOM tree to a SAX handler

The following example outputs a Dom tree without indentation.

```
MAIN
  DEFINE d  om.DomDocument
```

```
  DEFINE r, n, t, w om.DomNode
  DEFINE dh om.SaxDocumentHandler

  DEFINE i INTEGER

  LET dh = om.XmlWriter.createPipeWriter("cat")
  CALL dh.setIndent(FALSE)

  LET d = om.DomDocument.create("Vehicles")
  LET r = d.getDocumentElement()

  LET n = r.createChild("Car")
  CALL n.setAttribute("name","Corolla")
  CALL n.setAttribute("color","Blue")
  CALL n.setAttribute("weight","1546")

  LET t = d.createChars("Nice car!")
  CALL n.appendChild(t)

  LET n = r.createChild("Bus")
  CALL n.setAttribute("name","Maxibus")
  CALL n.setAttribute("color","yellow")
  CALL n.setAttribute("weight","5278")
  FOR i=1 TO 4
    LET w = n.createChild("Wheel")
    CALL w.setAttribute("width","315")
    CALL w.setAttribute("diameter","925")
  END FOR

  CALL r.write(dh)

END MAIN
```

The NodeList class
A `om.NodeList` object hold a list of DOM nodes.

The list is created from a `om.DomNode.selectByTagName()` or `om.DomNode.selectByPath()` method.

After creating the node list, you can process the nodes withe the `getLength()` and `item()` methods of the om.NodeList object.

om.NodeList methods
Methods of the `om.NodeList` class.

Table 404: Object methods

Name	Description
`getLength()` `  RETURNING` *`result`* `INTEGER`	Returns the number of elements in the node list.
`item(` *`index`* `INTEGER )` `  RETURNING` *`result`* `om.DomNode`	Returns a DOM node element by position in the node list.

om.NodeList.getLength
Returns the number of elements in the node list.

Syntax

```
getLength()
  RETURNING result INTEGER
```

1. *node* is a reference to a node.

Usage

The `getLength()` method returns the size of the node list.

Query for node list for elements with the `item()` method, in the range 1 to `getLength()`.

Example

```
DEFINE list om.NodeList
...
DISPLAY list.getLength()
```

om.NodeList.item
Returns a DOM node element by position in the node list.

Syntax

```
item( index INTEGER )
  RETURNING result om.DomNode
```

1. *index* is the ordinal position of the node in the list.

Usage

The `item()` method returns the `om.DomNode` object at the position specified.

First element is at position 1.

If there is no element at the specified index, the method returns `NULL`.

Example

```
DEFINE list om.NodeList,
       node om.DomNode
...
LET node = list.item(12)
```

Examples
Example 1: Search for child nodes by tag name

```
MAIN
  DEFINE nl om.NodeList
  DEFINE r, n om.DomNode
  DEFINE i INTEGER

  LET r = ui.Interface.getRootNode()
  LET nl = r.selectByTagName("Form")

  FOR i=1 to nl.getLength()
```

```
    LET n = nl.item(i)
    DISPLAY n.getAttribute("name")
  END FOR

END MAIN
```

Example 2: Search for child nodes by XPath

```
MAIN
  DEFINE nl om.NodeList
  DEFINE r, n om.DomNode
  DEFINE i INTEGER

  LET r = ui.Interface.getRootNode()
  LET nl = r.selectByPath("//Window[@name=\"screen\"]")

  FOR i=1 to nl.getLength()
    LET n = nl.item(i)
    DISPLAY n.getAttribute("name")
  END FOR

END MAIN
```

The SaxAttributes class
The `om.SaxAttributes` class holds a set of attributes to process with a SAX reader or writer.

To process SAX attributes, create a `om.SaxAttributes` object with a SAX reader or SAX writer SAX writer object.

Get an instance of SaxAttributes with the `om.XmlReader.getAttributes()` method.

om.SaxAttributes methods
Methods of the `om.SaxAttributes` class.

Table 405: Class methods

Name	Description
`copy( attrs om.SaxAttributes ) RETURNING result om.SaxAttributes`	Clones an existing SAX attributes object.
`create() RETURNING result om.SaxAttributes`	Create a new SAX attributes object.

Table 406: Object methods

Name	Description
`addAttribute( name STRING, value STRING )`	Appends a new attribute to the end of the list.
`clear()`	Clears the SAX attribute list.
`getLength()`	Returns the number of attributes in the list.

Name	Description
`RETURNING result INTEGER`	
`getName( index INTEGER ) RETURNING result STRING`	Returns the name of an attribute by position.
`getValue( name STRING ) RETURNING result STRING`	Returns the value of an attribute by name.
`getValueByIndex( index INTEGER ) RETURNING result STRING`	Returns an attribute value by position.
`removeAttribute( index INTEGER )`	Delete an attribute by position.
`setAttributes( attrs om.SaxAttributes )`	Clears the list and copies the attributes passed.

om.SaxAttributes.addAttribute
Appends a new attribute to the end of the list.

Syntax

```
addAttribute(
   name STRING,
   value STRING )
```

1. *name* is the name of the attribute.
2. *value* is the value of the attribute.

Usage

The `addAttribute()` method appends a new attribute with name and value at the end of the list.

Attribute names are case-sensitive.

> **Note:** Make sure that the strings passed to the method do not contain illegal XML characters: Illegal XML characters will be silently ignored. Illegal XML characters are any character below space (ASCII 32), except \r (ASCII 13), \n (ASCII 10) and \t (ASCII 9).

Example

```
DEFINE attrs om.SaxAttributes
...
CALL attrs.addAttribute("name","jo")
```

om.SaxAttributes.copy
Clones an existing SAX attributes object.

Syntax

```
copy(
   attrs om.SaxAttributes )
  RETURNING result om.SaxAttributes
```

1. *attrs* is a set of SAX attributes to clone.

Usage

The `om.SaxAttributes.copy()` class method makes a clone of the `om.SaxAttributes` object passed as reference and returns the new created object.

Example

```
DEFINE copy, orig om.SaxAttributes
...
LET copy = om.SaxAttributes.copy(orig)
```

om.SaxAttributes.create
Create a new SAX attributes object.

Syntax

```
create()
  RETURNING result om.SaxAttributes
```

Usage

The `om.SaxAttributes.create()` class method create a new `om.SaxAttributes` object returns it.

To hold the reference to a SAX attributes object, define a variable with the type `om.SaxAttributes` type.

Example

```
DEFINE attrs om.SaxAttributes
...
LET attrs = om.SaxAttributes.create()
```

om.SaxAttributes.clear
Clears the SAX attribute list.

Syntax

```
clear()
```

Usage

Use the `clear()` method the clean the SAX attribute list.

om.SaxAttributes.getLength
Returns the number of attributes in the list.

Syntax

```
getLength()
  RETURNING result INTEGER
```

Usage

The `getLength()` method returns the number of attributes in the current SAX attribute list.

Use this method with `getName()` and `getValueByIndex()` to retrieve attributes by position.

Example

```
DEFINE attrs om.SaxAttributes,
       index INTEGER
...
FOR index = 1 TO attrs.getLength()
    DISPLAY attrs.getName(index), " = ",
 attrs.getValueByIndex(index)
END FOR
```

om.SaxAttributes.getName
Returns the name of an attribute by position.

Syntax

```
getName(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the position of the attribute in the list.

Usage

The `getName()` method returns the name of the attribute at the specified ordinal position in the list.

If the attribute does not exist at the given position, the method returns `NULL`.

om.SaxAttributes.getValue
Returns the value of an attribute by name.

Syntax

```
getValue(
   name STRING )
  RETURNING result STRING
```

1. *name* is the name of an attribute.

Usage

The `getValue()` method returns the value of the attribute identified by the name passed as parameter.

If the attribute does not exist, the method returns `NULL`.

Example

```
DEFINE attrs om.SaxAttributes
...
DISPLAY attrs.getValue("name")
```

om.SaxAttributes.getValueByIndex
Returns an attribute value by position.

Syntax

```
getValueByIndex(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the position of the attribute in the list.

Usage

The `getValueByIndex()` method returns the value of the attribute at the specified ordinal position in the list.

If the attribute does not exist at the given position, the method returns `NULL`.

om.SaxAttributes.removeAttribute
Delete an attribute by position.

Syntax

```
removeAttribute(
   index INTEGER )
```

1. *index* is the position of the attribute in the list.

Usage

The `removeAttribute()` method removes the attribute at the given ordinal position.

If the attribute does not exist at the given position, the method returns silently.

Example

```
DEFINE attrs om.SaxAttributes
...
CALL attrs.removeAttribute( attrs.getLength() )
```

om.SaxAttributes.setAttributes
Clears the list and copies the attributes passed.

Syntax

```
setAttributes(
   attrs om.SaxAttributes )
```

1. *attrs* is a reference to list of attributes.

Usage

The `setAttributes()` method takes an existing `om.SaxAttributes` object reference and makes a copy of all attributes into the current attribute list.

Example

```
DEFINE curr, orig om.SaxAttributes
...
CALL curr.setAttributes(orig)
```

Examples

Example 1: Displaying SAX attributes of an XML node

```
FUNCTION displayAttributes( a )
   DEFINE a om.SaxAttributes
   DEFINE i INTEGER
   FOR i=1 to a.getLength()
      DISPLAY a.getName(i) || "=[" || a.getValueByIndex(i) || "]"
   END FOR
END FUNCTION
```

The SaxDocumentHandler class

The `om.SaxDocumentHandler` class provides an interface to write an XML filter with events.

This class follows the SAX standards.

A `om.SaxDocumentHandler` object can be used in two different ways:

1. To implement an XML SAX filter, based of functions defined in a `.4gl` module, by using the `createForName()` class method.
2. To write an XML document to a file, process or socket output, by using `om.XmlWriter` creation methods, and the `om.SaxDocumentHandler` processing methods.

The `om.SaxDocumentHandler` class also provides methods to process all SAX events by hand. This is useful if you want to chain SAX filters.

om.SaxDocumentHandler methods

Methods of the `om.SaxDocumentHandler` class.

Table 407: Class methods

Name	Description

`om.SaxDocumentHandler.createForName( module STRING )`
 	
`RETURNING result om.SaxDocumentHandler`
	Creates a new SAX document handler object for the given .4gl module.

Table 408: Object methods

Name	Description
`characters(`	Processes a text node.

Name	Description
data STRING)	
endDocument()	Processes the end of the document.
endElement(*tagname* STRING)	Processes the end of an element.
processingInstruction(*name* STRING, *data* STRING)	Processes a processing instruction.
readXmlFile(*filename* STRING)	Reads and processes an XML file with the SAX document handler.
setIndent(*on* BOOLEAN)	Controls indentation in XML output.
startDocument()	Processes the beginning of the document.
startElement(*tagname* STRING, *attrs* om.SaxAttributes)	Processes the beginning of an element.
skippedEntity(*name* STRING)	Processes an unresolved entity.

om.SaxDocumentHandler.createForName
Creates a new SAX document handler object for the given .4gl module.

Syntax

```
om.SaxDocumentHandler.createForName(
   module STRING )
  RETURNING result om.SaxDocumentHandler
```

1. *module* is the name of the .4gl module defining the document handler events.

Usage

The `om.SaxDocumentHandler.createForName()` method creates a `om.SaxDocumentHandler` instance and binds the .4gl module passed as argument to the object.

To hold the reference to a SAX document handler object, define a variable with the type `om.SaxDocumentHandler` type.

The .4gl module must be available as a compiled 42m file, loadable according to environment settings (FGLLDPATH).

The .4gl module must implement the following functions to process the SAX filter events:

Table 409: Functions of the SAX document handler module

Function	Description
`startDocument()`	Called once at the beginning of the document processing.
`endDocument()`	Called once at the end of the document processing.

`startElement(`
` tagname STRING,`
` attrs om.SaxAttributes )`
1. *tagname* is the tag name of element.
2. *attrs* is list of attributes.
	Called when an XML element is reached. Use the `om.SaxAttributes` methods to handle the attributes of the processed element.

`endElement( tagname STRING )` **1.** *tagname* is the tag name of element.	Called when the end of an XML element is reached.

`processingInstruction(`
` piname STRING,`
` data STRING )`
1. *piname* is the name of the processing instruction.
2. *data* is the content of the processing instruction.
	Called when a processing instruction is reached.

`characters( data STRING )` **1.** *data* is the text data.	Called when a text node is reached.
`skippedEntity( name STRING )` **1.** *name* is the name of the unknown entity.	Called when an unknown entity node is reached (like &xxx; for example).

Example

```
DEFINE f om.SaxDocumentHandler
LET f = om.SaxDocumentHandler.createForName("mysaxmod")
```

om.SaxDocumentHandler.characters
Processes a text node.

Syntax

```
characters(
   data STRING )
```

1. *data* is the content of the text node.

Usage

The `characters()` method processes a text node with the SAX interface.

> **Note:** Make sure that the strings passed to the method do not contain illegal XML characters: Illegal XML characters will be silently ignored. Illegal XML characters are any character below space (ASCII 32), except \r (ASCII 13), \n (ASCII 10) and \t (ASCII 9).

om.SaxDocumentHandler.endDocument
Processes the end of the document.

Syntax

```
endDocument()
```

Usage

The `endDocument()` method ends the document processing with the SAX interface.

om.SaxDocumentHandler.endElement
Processes the end of an element.

Syntax

```
endElement(
   tagname STRING )
```

1. *tagname* is the tag name of element.

Usage

The `endElement()` method processes the end of an element with the SAX interface.

om.SaxDocumentHandler.processingInstruction
Processes a processing instruction.

Syntax

```
processingInstruction(
   name STRING,
   data STRING )
```

1. *name* is the name of the processing instruction (token after <?).
2. *data* is the string in the processing instruction tag.

Usage

The `processingInstruction()` method processes a processing instruction with the SAX interface.

A processing instruction appears in an XML formatted text as:

```
<?name data ?>
```

om.SaxDocumentHandler.readXmlFile
Reads and processes an XML file with the SAX document handler.

Syntax

```
readXmlFile(
```

```
    filename STRING )
```

1. *filename* is the path to an XML formatted file.

Usage

Use the `readXmlFile()` method after creating the `om.SaxDocumentHandler` object, to process the XML data from a file input stream.

Example

```
DEFINE f om.SaxDocumentHandler
LET f = om.SaxDocumentHandler.createForName("mysaxmod")
CALL f.readXmlFile("cars.xml")
```

om.SaxDocumentHandler.setIndent
Controls indentation in XML output.

Syntax

```
setIndent(
   on BOOLEAN )
```

1. *on* is a boolean: `TRUE` enables identation; `FALSE` disables indentation.

Usage

By default, the `om.SaxDocumentHandler` object outputs XML with indentation.

In order to disable indentation, use the `setIndent(FALSE)` method.

om.SaxDocumentHandler.startDocument
Processes the beginning of the document.

Syntax

```
startDocument()
```

Usage

The `startDocument()` method begins the document processing with the SAX interface.

om.SaxDocumentHandler.startElement
Processes the beginning of an element.

Syntax

```
startElement(
   tagname STRING,
   attrs om.SaxAttributes )
```

1. *tagname* is the tag name of element.
2. *attrs* is the list of attributes of the element.

Usage

The `startElement()` method processes the beginning of an element with the SAX interface.

Use the `om.SaxAttributes` methods to handle the attributes of an element.

Example

```
DEFINE out om.SaxDocumentHandler
       attrs om.SaxAttributes,
       node om.DomNode,
       x INTEGER
...
CALL attrs.clear()
FOR x=1 TO r.getAttributesCount()
    CALL attrs.addAttribute( node.getAttributeName(x),
 node.getAttributeValue(x) )
END FOR
CALL out.startElement( node.getTagName(), attrs )
```

om.SaxDocumentHandler.skippedEntity
Processes an unresolved entity.

Syntax

```
skippedEntity(
   name STRING )
```

1. *name* is the name of the unknown entity.

Usage

The `skippedEntity()` method processes an entity that is not known by the XML parser.

Examples
Example 1: Extracting phone numbers from a directory.

This example shows how to write a SAX filter to extract phone numbers from a directory file written in XML.

```
MAIN
  DEFINE f om.SaxDocumentHandler
  LET f = om.SaxDocumentHandler.createForName("module1")
  CALL f.readXmlFile("customers")
END MAIN
```

Note:

1. The parameter of the createForName() method specifies the name of a source file that has been compiled into a .42m file ("module1.42m" in our example).

The module "module1.4gl":

```
FUNCTION startDocument()
END FUNCTION

FUNCTION processingInstruction(name,data)
  DEFINE name,data STRING
END FUNCTION

FUNCTION startElement(name,attr)
  DEFINE name STRING
  DEFINE attr om.SaxAttributes
  DEFINE i INTEGER
  IF name="Customer" THEN
     DISPLAY attr.getValue("lname")," ",
```

```
            attr.getValue("fname"),":",
            COLUMN 60, attr.getValue("phone")
  END IF
END FUNCTION

FUNCTION endElement(name)
  DEFINE name STRING
END FUNCTION

FUNCTION endDocument()
END FUNCTION

FUNCTION characters(chars)
  DEFINE chars STRING
END FUNCTION

FUNCTION skippedEntity(chars)
  DEFINE chars STRING
END FUNCTION
```

The XML file "customers":

```
<Customers>
  <Customer customer_num="101" fname="Ludwig" lname="Pauli"
    company="All Sports Supplies" address1="213 Erstwild Court"
    address2="" city="Sunnyvale" state="CA" zip-code="94086"
    phone="408-789-8075" />
  <Customer customer_num="102" fname="Carole" lname="Sadler"
    company="Sports Spot" address1="785 Geary St"
    address2="" city="San Francisco" state="CA" zip-code="94117"
    phone="415-822-1289" />
  <Customer customer_num="103" fname="Philip" lname="Currie"
    company="Phil&apos;s Sports" address1="654 Poplar"
    address2="P. O. Box 3498" city="Palo Alto" state="CA"
    zip-code="94303" phone="415-328-4543" />
</Customers>
```

The XmlReader class

The `om.XmlReader` class provides methods to read and process a file written in XML format.

The processing of the XML file is streamed-data based; the file is loaded and processed sequentially with events. To process XML element attributes, an `om.XmlReader` object must cooperate with a `om.SaxAttributes` object. The XmlReader class can only read from a file. To write to a file, use the `om.XmlWriter` class.

Steps to use a XML reader:

1. Declare a variable with the `om.XmlReader` type.
2. Create the reader object with the `createFileReader()` method and assign the reference to the variable.
3. Process SAX events in a `WHILE` loop, by reading document fragments with the `read()` method.
4. Inside the loop, according to the SAX event, process element attributes with getAttributes() or get the element data with the getCharacters() methods.

om.XmlReader methods

Methods of the `om.XmlReader` class.

Table 410: Class methods

Name	Description
`createFileReader( filename STRING ) RETURNING result om.XmlReader`	Creates an XML reader object from a file.

Table 411: Object methods

Name	Description
`getCharacters() RETURNING result STRING`	Returns the character data of the current processed element.
`getAttributes() RETURNING result om.SaxAttributes`	Builds an attribute list for the current processed element.
`getTagName() RETURNING result STRING`	Returns the tag name of the current processed element.
`read() RETURNING result STRING`	Reads the next SAX event to process.
`skippedEntity() RETURNING result STRING`	Returns the name of an unresolved entity.

om.XmlReader.createFileReader

Creates an XML reader object from a file.

Syntax

```
createFileReader(
   filename STRING )
  RETURNING result om.XmlReader
```

1. *filename* is the path to an XML formatted file.

Usage

Use the `om.XmlReader.createFileReader()` method to create a new `om.XmlReader` object, to process the XML data from a file input stream.

To hold the reference to an XmlReader object, define a variable with the type `om.XmlReader` type.

Example

```
DEFINE r om.XmlReader
```

```
LET r = om.XmlReader.createFileReader("cars.xml")
```

om.XmlReader.getAttributes
Builds an attribute list for the current processed element.

Syntax

```
getAttributes()
  RETURNING result om.SaxAttributes
```

Usage

Use the `getAttributes()` method create a list of attributes as a `om.SaxAttributes` object, from the current processed element, in the `StartElement` or `EndElement` event context.

Declare a variable with the `om.SaxAttributes` type to reference the attribute list.

Note that once created with the `getAttributes()` method, the `om.SaxAttributes` object is automatically updated based on the element currently processed by the `om.XmlReader`.

Example

```
DEFINE r om.XmlReader,
       e STRING, i INT
       a om.SaxAttributes
   ...
   LET e = r.read()
   WHILE e IS NOT NULL
     CASE e
       ...
       WHEN "StartElement"
         LET a = r.getAttributes()
         FOR i=1 to a.getLength()
             ...
```

om.XmlReader.getCharacters
Returns the character data of the current processed element.

Syntax

```
getCharacters()
  RETURNING result STRING
```

Usage

Use the `getCharacters()` method to get the character data of the current processed element, in the `Characters` event context.

Example

```
DEFINE r om.XmlReader,
       e STRING
   ...
   LET e = r.read()
   WHILE e IS NOT NULL
     CASE e
       ...
       WHEN "Characters"
```

```
          DISPLAY "Characters:'",r.getCharacters(),"'"
      ...
```

om.XmlReader.getTagName
Returns the tag name of the current processed element.

Syntax

```
getTagName()
  RETURNING result STRING
```

Usage

Use the `readXmlFile()` method to get the tag name of the current processed element, in the `StartElement` or `EndElement` event context.

Example

```
DEFINE r om.XmlReader,
       e STRING
   ...
   LET e = r.read()
   WHILE e IS NOT NULL
     CASE e
       ...
       WHEN "StartElement"
         DISPLAY "TagName = ", r.getTagName()
             ...
```

om.XmlReader.read
Reads the next SAX event to process.

Syntax

```
read()
  RETURNING result STRING
```

Usage

The `read()` method reads the next XML fragment and returns the name of the SAX event to process.

Table 412: Events that can be returned by the `read()`method

Event name	Description	Action
`StartDocument`	Beginning of the document	Prepare processing (allocate resources)
`StartElement`	Beginning of a node	Get current element's tag name or attributes with `getTagName()` `getAttributes()`
`Characters`	Value of the current element	Get current text element's value with `getCharacters()`
`SkippedEntity`	Reached skipped entity	Get current skipped entity element's value with `skippedEntity()`
`EndElement`	Ending of a node	Get current element's tagname with `getTagName()`
`EndDocument`	Ending of the document	Finish processing (release resources)

Example

```
DEFINE r om.XmlReader,
       e STRING
   ...
   LET e = r.read()
   WHILE e IS NOT NULL
     CASE e
       ...
     END CASE
     LET e = r.read()
   END WHILE
```

om.XmlReader.skippedEntity
Returns the name of an unresolved entity.

Syntax

```
skippedEntity()
  RETURNING result STRING
```

Usage

The `skippedEntity()` method returns the name of the unresolved entity, in the `SkippedEntity` event context.

The parser identifies well know character entities such as `&` / `&apos;` / `<` / `>` / `&quot;`, other character entities are treated as skipped entities and can be processed in the `SkippedEntity` event.

Example

```
DEFINE r om.XmlReader,
       e STRING
   ...
```

```
    LET e = r.read()
    WHILE e IS NOT NULL
      CASE e
        ...
        WHEN "SkippedEntity"
          DISPLAY "Entity:'",r.skippedEntity(),"'"
      ...
```

Examples

Example 1: Parsing an XML file

```
MAIN
   DEFINE i, l INTEGER
   DEFINE r om.XmlReader
   DEFINE e String
   DEFINE a om.SaxAttributes
   LET r = om.XmlReader.createFileReader("myfile.xml")
   LET l = 0
   LET e = r.read()
   WHILE e IS NOT NULL
     CASE e
       WHEN "StartDocument"
         DISPLAY "StartDocument:"
       WHEN "StartElement"
         LET l=l+1
         DISPLAY l SPACES, "StartElement:", r.getTagName()
         LET a = r.getAttributes()
         FOR i=1 to a.getLength()
           DISPLAY l SPACES,"  ",
             a.getName(i)," = ",
             a.getValueByIndex(i)
         END FOR
       WHEN "Characters"
         DISPLAY l SPACES, "  Characters:'",r.getCharacters(),"'"
       WHEN "EndElement"
         DISPLAY l SPACES, "EndElement:", r.getTagName()
         LET l=l-1
       WHEN "EndDocument"
         DISPLAY "EndDocument:"
       OTHERWISE
         DISPLAY "Invalid event: ",e
     END CASE
     LET e=r.read()
   END WHILE
END MAIN
```

The XmlWriter class

The `om.XmlWriter` class implements methods to write XML to a stream.

The `om.XmlWriter` class implements methods to create a `om.SaxDocumentHandler` object.

Steps to use a XML writer:

1. Declare a variable with the `om.SaxDocumentHandler` type.
2. Create the writer object with one of the class methods of `om.XmlWriter` and assign the reference to the variable.
   - `om.XmlWriter.createFileWriter(`*`filename`*`)` creates an object writing to a file.
   - `om.XmlWriter.createPipeWriter(`*`command`*`)` creates an object writing to a pipe opened by a sub-process.
   - `om.XmlWriter.createSocketWriter(`*`hostname`*`,`*`portnum`*`)` creates an object writing to the TCP socket.

3. Output XML data with the methods of the `om.SaxDocumentHandler` object:
   a. Use the method `startDocument()` to start writing to the output.
   b. From this point, the order of method calls defines the structure of the XML document. To write an element, fill an `om.SaxAttributes` object with attributes.
   c. Then, initiate the element output with the method `startElement()`.
   d. Write element data with the `characters()` method.
   e. Entity nodes are created with the `skippedEntity()` method.
   f. Finish element output with a call to the `endElement()` method.
   g. Repeat these steps as many times as you have elements to write.
   h. Instead of using the `startElement()` method, you can generate processing instruction elements with `processingInstruction()`.
   i. Finally, you must finish the document output with a `endDocument()` call.

om.XmlWriter methods
Methods of the `om.XmlWriter` class.

Table 413: Class methods

Name	Description
`om.XmlWriter.createChannelWriter( channel base.Channel ) RETURNING result om.SaxDocumentHandler`	Creates an `om.SaxDocumentHandler` object writing to a channel object.
`om.XmlWriter.createFileWriter( filename STRING ) RETURNING result om.SaxDocumentHandler`	Creates an `om.SaxDocumentHandler` object writing to a file.
`om.XmlWriter.createPipeWriter( command STRING ) RETURNING result om.SaxDocumentHandler`	Creates an `om.SaxDocumentHandler` object writing to a pipe created for a process.
`om.XmlWriter.createSocketWriter( host STRING, port INTEGER ) RETURNING result om.SaxDocumentHandler`	Creates an `om.SaxDocumentHandler` object writing to a socket.

om.XmlWriter.createChannelWriter
Creates an `om.SaxDocumentHandler` object writing to a channel object.

Syntax

```
om.XmlWriter.createChannelWriter(
   channel base.Channel )
  RETURNING result om.SaxDocumentHandler
```

1. *channel* is a `base.Channel` object reference.

Usage

The `om.XmlWriter.createChannelWriter()` class method creates an `om.SaxDocumentHandler` object that will write to the specified channel object, when using the `om.SaxDocumentHandler` methods.

The `base.Channel` object must exist and be open to receive data from the SAX document handler.

Example

The next example uses the channel to write to stdout, passing `NULL` as file name to the `base.Channel.openFile()` method:

```
DEFINE w om.SaxDocumentHandler
DEFINE ch base.Channel
...
LET ch = base.Channel.create()
CALL ch.openFile(NULL,"w")
LET w = om.XmlWriter.createChannelWriter(ch)
...
```

om.XmlWriter.createFileWriter
Creates an `om.SaxDocumentHandler` object writing to a file.

Syntax

```
om.XmlWriter.createFileWriter(
   filename STRING )
  RETURNING result om.SaxDocumentHandler
```

1. *filename* is the path to the file.

Usage

The `om.XmlWriter.createFileWriter()` class method creates an `om.SaxDocumentHandler` object that will write to the specified file when using the `om.SaxDocumentHandler` methods.

The file is created if it does not exist. If the file cannot be created, the method returns `NULL`.

When passing `NULL` as file name, the XmlWriter can be used to write to stdout.

Example

```
DEFINE w om.SaxDocumentHandler
...
LET w = om.XmlWriter.createFileWriter("mydata.xml")
IF w IS NULL THEN
   ERROR "Could not create file."
   EXIT PROGRAM 1
END IF
...

-- Create an XmlWriter object to write to stdout:
LET w = om.XmlWriter.createFileWriter(NULL)
...
```

om.XmlWriter.createPipeWriter
Creates an `om.SaxDocumentHandler` object writing to a pipe created for a process.

Syntax

```
om.XmlWriter.createPipeWriter(
   command STRING )
  RETURNING result om.SaxDocumentHandler
```

1. *command* is the command to be executed.

Usage

The `om.XmlWriter.createPipeWriter()` class method creates an `om.SaxDocumentHandler` object that will write to a pipe created for the specified command. XML data will be send through the pipe when using the `om.SaxDocumentHandler` methods.

If the process or pipe cannot be created, the method returns `NULL`.

Example

```
DEFINE w om.SaxDocumentHandler
...
LET w = om.XmlWriter.createPipeWriter("sort -u")
IF w IS NULL THEN
   ERROR "Could not create process."
   EXIT PROGRAM 1
END IF
```

om.XmlWriter.createSocketWriter
Creates an `om.SaxDocumentHandler` object writing to a socket.

Syntax

```
om.XmlWriter.createSocketWriter(
   host STRING,
   port INTEGER )
  RETURNING result om.SaxDocumentHandler
```

1. *host* is the name of the host computer listening to the TCP port.
2. *port* is the port number to connect to.

Usage

The `om.XmlWriter.createSocketWriter()` class method creates an `om.SaxDocumentHandler` object that will write to a socket identified by the host and port number passed as parameters. XML data will be send through the socket when using the `om.SaxDocumentHandler` methods.

If the socket cannot be opened, the method returns `NULL`. No timeout is used.

Example

```
DEFINE w om.SaxDocumentHandler
...
LET w = om.XmlWriter.createSocketWriter("myhost",8012)
IF w IS NULL THEN
   ERROR "Could not open socket."
   EXIT PROGRAM 1
```

```
  END IF
```

Examples

Example 1: Writing XML to a file

```
MAIN
  DEFINE w om.SaxDocumentHandler
  DEFINE a,n om.SaxAttributes

  LET w = om.XmlWriter.createFileWriter("sample.html")
  LET a = om.SaxAttributes.create()
  LET n = om.SaxAttributes.create()

  CALL n.clear()

  CALL w.startDocument()

    CALL w.startElement("HTML",n)

      CALL w.startElement("HEAD",n)

        CALL w.startElement("TITLE",n)
        CALL w.characters("HTML page generated with XmlWriter")
        CALL w.endElement("TITLE")

        CALL a.clear()
        CALL a.addAttribute("type","text/css")
        CALL w.startElement("STYLE",a)
        CALL w.characters("\nBODY { background-color:#c0c0c0; }\n")
        CALL w.endElement("STYLE")

      CALL w.endElement("HEAD")

      CALL w.startElement("BODY",n)

        CALL addHLine(w)
        CALL addTitle(w,"What is XML?",1,"55ff55")
        CALL addParagraph(w,"XML = eXtensible Markup Language ...")

        CALL addHLine(w)
        CALL addTitle(w,"What is SAX?",1,"55ff55")
        CALL addParagraph(w,"SAX = Simple Api for XML ...")

      CALL w.endElement("BODY")

    CALL w.endElement("HTML")

  CALL w.endDocument()

END MAIN

FUNCTION addHLine(w)
  DEFINE w om.SaxDocumentHandler
  DEFINE a om.SaxAttributes
  LET a = om.SaxAttributes.create()
  CALL a.clear()
  CALL a.addAttribute("width","100%")
  CALL w.startElement("HR",a)
  CALL w.endElement("HR")
END FUNCTION

FUNCTION addTitle(w,t,x,c)
```

```
  DEFINE w om.SaxDocumentHandler
  DEFINE t VARCHAR(100)
  DEFINE x INTEGER DEFINE c VARCHAR(20)
  DEFINE a om.SaxAttributes
  DEFINE n varchar(10)
  LET a = om.SaxAttributes.create()
  LET n = "h" || x
  CALL a.clear()
  CALL w.startElement(n,a)
  IF c IS NOT NULL THEN CALL a.addAttribute("color",c)
  END IF CALL w.startElement("FONT",a)
  CALL w.characters(t)
  CALL w.endElement("FONT")
  CALL w.endElement(n)
END FUNCTION

FUNCTION addParagraph(w,t)
  DEFINE w om.SaxDocumentHandler
  DEFINE t VARCHAR(2000)
  DEFINE a om.SaxAttributes
  LET a = om.SaxAttributes.create()
  CALL a.clear()
  CALL w.startElement("P",a)
  CALL w.characters("Text is:")
  CALL w.skippedEntity("nbsp") # Add a non breaking space:  
  CALL w.characters("is")
  CALL w.characters(t)
  CALL w.endElement("P")
END FUNCTION
```

Built-in front calls

This section contains the description of all built-in front calls.

- Built-in front calls on page 1881
  - Standard front calls on page 1889
  - Webcomponent front calls on page 1902
  - Genero Desktop Client front calls on page 1904
  - Genero Application Server front calls on page 1924
  - Genero Mobile common front calls on page 1925
  - Genero Mobile Android front calls on page 1940
  - Genero Mobile iOS front calls on page 1945

Built-in front calls

Various front-end functions are implemented within Genero front-ends.

This section describes the front-end functions available for all type of front-ends. Note that several front-end functions are specific to the type of front-end.

Table 414: Standard front-end functions

Function Name	Description	GDC	GWC-JS	GMA	GMI
`ui.Interface.frontCall("standard", "cbAdd",`	Adds to the content of the clipboard.	Yes	No	No	No

Function Name	Description	GDC	GWC-JS	GMA	GMI
`[text], [result])`					
`ui.Interface.frontCall("standard", "cbClear", [], [result])`	Clears the content of the clipboard.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbGet", [], [text])`	Gets the content of the clipboard.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbPaste", [], [result])`	Pastes the content of the clipboard to the current field.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbSet", [text], [result])`	Set the content of the clipboard.	Yes	No	No	No
`ui.Interface.frontCall("standard", "execute", [cmd,wait], [result])`	Executes a command on the front-end platform, with or without waiting.	Yes	No	No	No
`ui.Interface.frontCall("standard", "feInfo", [name], [result])`	Queries general front-end properties.	Yes	Yes	Yes	Yes
`ui.Interface.frontCall("standard", "getEnv", [name], [value])`	Returns an environment variable set in the user session on the front end platform.	Yes	No	No	No
`ui.Interface.frontCall("standard", "getWindowId", [aui-win-id], [loc-win-id])`	Returns the local window manager identifier of the window corresponding to the AUI window id passed as parameter.	Yes	No	No	No
`ui.Interface.frontCall("standard", "hardCopy", [pgsize], [result])`	Prints a screen shot of the current window	Yes	No	No	No
`ui.Interface.frontCall("standard", "launchURL", [ url [, mode ] ], [] )`	Opens an URL with the default URL handler of the front-end.	Yes	Yes	Yes	Yes
`ui.Interface.frontCall("standard", "mdClose",`	Unloads a DLL or shared library front call module.	Yes	No	No	No

Function Name	Description	GDC	GWC-JS	GMA	GMI
[*name*], [*result*])					
ui.Interface.frontCall("standard", "openDir", [*path*,*caption*], [*result*])	Displays a file dialog window to get a directory path on the local file system.	Yes	No	No	No
ui.Interface.frontCall("standard", "openFile", [*path*,*name*,*wildcards*,*caption*], [*result*])	Displays a file dialog window to get a path to open a file on the local file system.	Yes	Yes	No	No
ui.Interface.frontCall("standard", "playSound", [*filename*], [])	Plays the sound file passed as parameter on the front-end platform.	Yes	Yes	Yes	Yes
ui.Interface.frontCall("standard", "saveFile", [*path*,*name*,*filetype*,*caption*], [*result*])	Displays a file dialog window to get a path to save a file on the local file system.	Yes	No	No	No
ui.Interface.frontCall("standard", "setReportFont", [*font*], [*result*])	Override the font used for report generation for the current application.	Yes	No	No	No
ui.Interface.frontCall("standard", "setReportPrinter", [*printer*], [*result*])	Override the printer configuration used for report generation for the current application.	Yes	No	No	No
ui.Interface.frontCall("standard", "setWebComponentPath", [*path*], [])	Defines the base path where web components are located.	Yes	N/A	Yes	Yes
ui.Interface.frontCall("standard", "shellExec", [*document*, *action*], [*result*])	Opens a file on the front-end platform with the program associated to the file extension.	Yes	No	No	No

Table 415: Webcomponent module front-end functions

Function Name	Description
ui.Interface.frontCall("webcomponent", "call", [*aui-name*, *function-name*, [*param1*, *param2*, ...]], [*result*]	Calls a JavaScript function through the web component.

Function Name	Description
)	
ui.Interface.frontCall("webcomponent", "frontCallAPIVersion", [],[*result*])	Returns the API version of web component front-end calls.
ui.Interface.frontCall("webcomponent", "getTitle", [*aui-name*], [*result*])	Returns the title of the HTML doc rendered by a web component.

Table 416: Windows DDE front-end functions

Function name	Description
CALL ui.Interface.frontCall("WINDDE","DDEConnect", [*program*, *document*, *encoding*], [*result*])	DDEConnect opens a DDE connection.
CALL ui.Interface.frontCall("WINDDE","DDEExecute", [*program*, *document*, *command*, *encoding*], [*result*])	DDEExecute executes a DDE command.
CALL ui.Interface.frontCall("WINDDE","DDEFinish", [*program*, *document*], [*result*])	DDEFinish closes a DDE connection.
CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [*result*])	DDEFinishAll closes all DDE connections.
CALL ui.Interface.frontCall("WINDDE","DDEError", [], [*errmsg*])	DDEError returns error information about the last DDE operation.
CALL ui.Interface.frontCall("WINDDE","DDEPeek", [*program*, *container*, *cells*, *encoding*], [*result*, *value*])	DDEPeek retrieves data from the specified program and document using the DDE channel.
CALL ui.Interface.frontCall("WINDDE","DDEPoke", [*program*, *container*, *cells*, *values*, *encoding*], [*result*])	DDEPoke sends data to the specified program and document using the DDE channel.

Table 417: Windows COM front-end functions

Function name	Description
`CALL ui.Interface.frontCall("WinCOM","CreateInstance", [ program ], [handle] )`	The `CreateInstance` function creates an instance of a registered COM object.
`CALL ui.Interface.frontCall("WINCOM","CallMethod", [ handle, method, arg1, ... ], [result] )` `CALL ui.Interface.frontCall("WINCOM","CallMethod", [ handle, method(arg1, ...) ], [result] )`	The `CallMethod` function calls a method on a specified object.
`CALL ui.Interface.frontCall("WINCOM","GetProperty", [ handle, member ], [result] )`	The `GetProperty` function gets a property of an object.
`CALL ui.Interface.frontCall("WINCOM","SetProperty", [handle, member, value], [result] )`	The `SetProperty` function sets a property of an object.
`CALL ui.Interface.frontCall("WINCOM","GetError", [], [result] )`	The `GetError` function gets a description of the last error which occurred.
`CALL ui.Interface.frontCall("WINCOM","ReleaseInstance", [handle], [result] )`	The `ReleaseInstance` function releases an instance of a COM object.

Table 418: WinMail front-end functions: General

Function name	Description
`CALL ui.Interface.frontCall("WinMail","Init", [], [id ] )`	The `Init` function initializes the module.
`CALL ui.Interface.frontCall("WinMail","Close", [id], [ result ] )`	The `Close` function clears all information corresponding to a message, and frees the memory occupied by the message.
`CALL ui.Interface.frontCall("WinMail","SetBody",`	The `SetBody` function sets the body of the mail.

Function name	Description
[*id*, *body*], [*result*])	
CALL ui.Interface.frontCall("WinMail","SetSubject", [*id*, *subject*], [*result*])	The SetSubject function sets the subject of the mail.
CALL ui.Interface.frontCall("WinMail","AddTo", [id, *name*, *address*], [*result*])	The AddTo function adds a "To" addressee to the mail.
CALL ui.Interface.frontCall("WinMail","AddCC", [id, *name*, *address*], [*result*])	The AddCC function adds a "CC" addressee to the mail.
CALL ui.Interface.frontCall("WinMail","AddBCC", [*id*, *name*, *address*], [*result*])	The AddBCC function adds a "BCC" addressee to the mail.
CALL ui.Interface.frontCall("WinMail","AddAttachment", [*id*, *fileName*], [*result*])	The AddAttachment function adds a file as an attachment to the mail.
CALL ui.Interface.frontCall("WinMail","SendMailSMTP", [*id*], [*result*])	The SendMailSMTP function sends the mail with the SMTP protocol.
CALL ui.Interface.frontCall("WinMail","SendMailMAPI", [*id*], [*result*])	The SendMailMAPI function sends the mail with the MAPI protocol.
CALL ui.Interface.frontCall("WinMail","GetError", [*id*], [*result*])	The GetError function gets a description of the last error that occurred.

Table 419: WinMail front-end functions: SMTP-specific

Function name	Description
CALL ui.Interface.frontCall("WinMail","SetSmtp", [*id*, *smtp:port*], [*result*])	The SetSmtp function sets the SMTP server to be used.
CALL ui.Interface.frontCall("WinMail","SetFrom",	The SetFrom function sets sender information.

Function name	Description
`[ id, name, address ], [ result ] )`	

Table 420: Session module front-end functions

Function name	Description
`ui.Interface.frontCall("session", "getVar", [name], [result])`	Returns the value of a session variable.
`ui.Interface.frontCall("session", "setVar", [name,value], [result])`	Sets a value of a session variable.

Table 421: Common mobile module front-end functions

Function Name	Description
`ui.Interface.frontCall("mobile", "chooseContact", [], [result])`	Lets the user choose a contact from the mobile device contact list and returns the vCard.
`ui.Interface.frontCall("mobile", "choosePhoto", [], [path])`	Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.
`ui.Interface.frontCall("mobile", "chooseVideo", [], [path])`	Lets the user select a video from the mobile device's video gallery and returns a video identifier.
`ui.Interface.frontCall("mobile", "composeMail", [to, subject, content, cc, bcc, attachments ...], [result])`	Invokes the user's default mail application for a new mail to send.
`ui.Interface.frontCall("mobile", "composeSMS", [ recipients, content ], [ result ] )`	Sends an SMS text to one or more phone numbers.
`ui.Interface.frontCall("mobile", "connectivity", [], [result] )`	Returns the type of network available for the mobile device.
`ui.Interface.frontCall("mobile", "getGeolocation", [], [status, latitude, longitude] )`	Returns the Global Positioning System (GPS) location of a mobile device.
`ui.Interface.frontCall("mobile","getRemoteNotifications",`	This front call retrieves push notification messages.

Function Name	Description
`[sender_id], [data] )`	
`ui.Interface.frontCall("mobile", "importContact", [vcard], [result] )`	Creates a new, or merges to an existing entry, the contact details passed in vCard string.
`ui.Interface.frontCall("mobile","registerForRemoteNotifications", [sender_id], [registration_token] )`	This front call registers a mobile device for push notifications.
`ui.Interface.frontCall("mobile", "runOnServer", [ appurl, timeout ], [] )`	Run an application from the Genero Application Server according to the specified URL.
`ui.Interface.frontCall("mobile", "scanBarCode", [], [code, type] )`	Allow the user to scan a barcode with a mobile device
`ui.Interface.frontCall("mobile", "takePhoto", [], [path] )`	Lets the user take a picture with the mobile device and returns the corresponding picture identifier.
`ui.Interface.frontCall("mobile", "takeVideo", [], [path])`	Lets the user take a video with the mobile device and returns the corresponding video identifier.
`ui.Interface.frontCall("mobile","unregisterFromRemoteNotifications", [ sender_id ], [] )`	This front call unregisters the mobile device from push notifications.

Table 422: Android module front-end functions

Function Name	Description
`ui.Interface.frontCall("android", "askForPermission", [permission], [result])`	Ask the user to enable a dangerous feature on the Android device.
`ui.Interface.frontCall("android", "showAbout", [],[])`	Shows the GMA about box displaying version information.
`ui.Interface.frontCall("android", "showSettings", [], [])`	Shows the GMA settings box controlling debug options.
`ui.Interface.frontCall("android", "startActivity", [action, data, category, type, component, extras],`	Starts an external Android application (activity), and returns to the GMA application immediately.

Function Name	Description
`[])`	
`ui.Interface.frontCall("android", "startActivityForResult", [action, data, category, type, component, extras], [outdata, outextras])`	Starts an external application (Android activity) and waits until the activity is closed.

Table 423: iOS module front-end functions

Function Name	Description
`ui.Interface.frontCall("ios", "getBadgeNumber", [],[value])`	Returns the current badge number associated to the app.
`ui.Interface.frontCall("ios", "newContact", [defaults],[vcard])`	Lets the user input contact information to create a new entry in the contact database of the mobile device.
`ui.Interface.frontCall("ios", "setBadgeNumber", [value], [])`	Sets the current badge number associated to the app.

Standard front calls

Standard front call functions provide common utility APIs to control the front-end.

This table shows the functions implemented by the front-ends in the "`standard`" module, available on all front-ends.

Table 424: Standard front-end functions

Function Name	Description	GDC	GWC-JS	GMA	GMI
`ui.Interface.frontCall("standard", "cbAdd", [text], [result])`	Adds to the content of the clipboard.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbClear", [], [result])`	Clears the content of the clipboard.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbGet", [], [text])`	Gets the content of the clipboard.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbPaste", [], [result])`	Pastes the content of the clipboard to the current field.	Yes	No	No	No
`ui.Interface.frontCall("standard", "cbSet",`	Set the content of the clipboard.	Yes	No	No	No

Function Name	Description	GDC	GWC-JS	GMA	GMI
`[text], [result])`					
`ui.Interface.frontCall("standard", "execute", [cmd,wait], [result])`	Executes a command on the front-end platform, with or without waiting.	Yes	No	No	No
`ui.Interface.frontCall("standard", "feInfo", [name], [result])`	Queries general front-end properties.	Yes	Yes	Yes	Yes
`ui.Interface.frontCall("standard", "getEnv", [name], [value])`	Returns an environment variable set in the user session on the front end platform.	Yes	No	No	No
`ui.Interface.frontCall("standard", "getWindowId", [aui-win-id], [loc-win-id])`	Returns the local window manager identifier of the window corresponding to the AUI window id passed as parameter.	Yes	No	No	No
`ui.Interface.frontCall("standard", "hardCopy", [pgsize], [result])`	Prints a screen shot of the current window	Yes	No	No	No
`ui.Interface.frontCall("standard", "launchURL", [ url [, mode ] ], [] )`	Opens an URL with the default URL handler of the front-end.	Yes	Yes	Yes	Yes
`ui.Interface.frontCall("standard", "mdClose", [name], [result])`	Unloads a DLL or shared library front call module.	Yes	No	No	No
`ui.Interface.frontCall("standard", "openDir", [path,caption], [result])`	Displays a file dialog window to get a directory path on the local file system.	Yes	No	No	No
`ui.Interface.frontCall("standard", "openFile", [path,name,wildcards,caption], [result])`	Displays a file dialog window to get a path to open a file on the local file system.	Yes	Yes	No	No
`ui.Interface.frontCall("standard", "playSound", [filename], [])`	Plays the sound file passed as parameter on the front-end platform.	Yes	Yes	Yes	Yes
`ui.Interface.frontCall("standard", "saveFile",`	Displays a file dialog window to get a path to save a file on the local file system.	Yes	No	No	No

Function Name	Description	GDC	GWC-JS	GMA	GMI
`[path,name,filetype,caption], [result])`					
`ui.Interface.frontCall("standard", "setReportFont", [font], [result])`	Override the font used for report generation for the current application.	Yes	No	No	No
`ui.Interface.frontCall("standard", "setReportPrinter", [printer], [result])`	Override the printer configuration used for report generation for the current application.	Yes	No	No	No
`ui.Interface.frontCall("standard", "setWebComponentPath", [path], [])`	Defines the base path where web components are located.	Yes	N/A	Yes	Yes
`ui.Interface.frontCall("standard", "shellExec", [document, action], [result])`	Opens a file on the front-end platform with the program associated to the file extension.	Yes	No	No	No

cbAdd
Adds to the content of the clipboard.

Syntax

```
ui.Interface.frontCall("standard", "cbAdd",
  [text], [result])
```

1. *text* - The text to be added.
2. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage

The "cbAdd" front call adds the text passed as parameter to the content of the clipboard of the front-end platform.

cbClear
Clears the content of the clipboard.

Syntax

```
ui.Interface.frontCall("standard", "cbClear",
  [], [result])
```

1. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage

The "cbClear" front call clears the content of the clipboard. This front call takes no input parameters.

cbGet
Gets the content of the clipboard.

Syntax

```
ui.Interface.frontCall("standard", "cbGet",
   [], [text])
```

1. *text* - Holds the text found in the clipboard.

Usage

The "cbGet" front call returns the current content of the clipboard.

This front call takes no input parameters.

cbPaste
Pastes the content of the clipboard to the current field.

Syntax

```
ui.Interface.frontCall("standard", "cbPaste",
   [], [result])
```

1. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage

The "cbPaste" front call pastes the content of the clipboard to the current field.

This front call takes no input parameters.

cbSet
Set the content of the clipboard.

Syntax

```
ui.Interface.frontCall("standard", "cbSet",
 [text], [result])
```

1. *text* - The text to be set.
2. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage

The "cbSet" front call sets the content of the clipboard with the text passed as parameter.

execute
Executes a command on the front-end platform, with or without waiting.

Syntax

```
ui.Interface.frontCall("standard", "execute",
 [cmd,wait], [result])
```

1. *cmd* - The command to be executed.
2. *wait* - The wait option (TRUE=wait, FALSE=do not wait).
3. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage

The "`execute`" front call runs a command on the front-end platform, with or without waiting option.

If the second parameter is set to 1 (`TRUE`), the runtime system will wait until the front-end gives the control back after the local command was executed.

feInfo
Queries general front-end properties.

Syntax

```
ui.Interface.frontCall("standard", "feInfo",
 [name], [result])
```

1. *name* - The name of the property.
2. *result* - Holds the value of the property.

Usage

The `feInfo` front call returns a front-end property value according to the property name passed in as the parameter.

Some feInfo options take an optional parameter, such as screenresolution:

```
CALL ui.Interface.frontCall("standard", "feInfo", ["screenResolution", 2],
 [resolution])
```

Table 425: Property names and descriptions for the standard.feInfo front call

feInfo property values

Property name	Description	GDC	GWC-JS	GMA	GMI
`dataDirectory`	Returns the directory name that can be used for temporary files on the front-end side. This directory is cleaned at front-end startup and end, and is common to all front-end instances, except GWC. The possible values returned are: • With Genero Web Client, this is not applicable. • With Genero Desktop Client, the local cache directory. For example, "`/home/username/.cache/Four Js/Genero Desktop`". • With Genero Mobile for Android™, this is the GMA application cache directory. Content may be erased, once the app is closed. • With Genero Mobile for iOS, this is the temporary directory in the application sandbox (iOS `NSTemporaryDirectory()` system call). Content may be erased, once the app is closed.	Yes	No	Yes	Yes
`dictionariesDirectory`	Returns the directory name where spell checking dictionary files are located. This parameter is only supported by GDC, for the `spellCheck` style attribute of TextEdit elements.	Yes	N/A	N/A	N/A

Property name	Description	GDC	GWC-JS	GMA	GMI
	A program can query the `dictionariesDirectory` info in order to send dictionary files to the GDC with an `fgl_putfile()` call.				
`deviceModel`	Returns the name of the device, e.g. "iPad4,5".	Yes	No	Yes	Yes
`deviceId`	• With Genero Mobile for iOS, returns the `identifierForVendor`. • With Genero Mobile for Android, returns this IMEI, otherwise the Android id (but may change after device reinstallation)	Yes	No	Yes	Yes
`feName`	The code identifying the type of front-end component. The possible values returned are: • "`Genero Desktop Client`" for Genero Desktop Client. • "`GBC`" for Genero Web Client - JavaScript (GWC-JS). • "`GWC` for Genero Web Client - HTML5 theme. • "`GMA`" for Genero Mobile for Android. • "`GMI`" for Genero Mobile for iOS.	Yes	Yes	Yes	Yes
`fePath`	The installation directory of the front-end executable. • With Genero Desktop Client, it returns the path to the installation directory of the GDC. • When Genero Web Client, it returns the path to the installation directory of the GAS. • With Genero Mobile for Android, it returns the installation directory. For example, "`/data/data/com.fourjs.gma/fgl`". • With Genero Mobile for iOS, it returns the installation directory. For example: "`/private/var/mobile/Applications/B3E6-C48A-ED4EFA`". Below the installation directory are the "`Documents`" (which is by default pwd), "`GMI.app`" (deployed p-code resides in `GMI.app/app/`) and "`tmp`" directories. **Important:** The installation path returned by this front call may change in future versions, do not based application code on this. On mobile devices, consider using the os.Path.pwd on page 2004 utility function to get the application working directory when executing programs.	Yes	No	Yes	Yes
`freeStorageSpace`	Returns the number of bytes available on the device.	Yes	No	Yes	Yes

Property name	Description	GDC	GWC-JS	GMA	GMI
`iccid`	• With Genero Mobile for iOS, returns an error (not allowed). • With Genero Mobile for Android, returns the ICCID if available, otherwise raise an error.	N/A	N/A	Yes	Yes
`imei`	• With Genero Mobile for iOS, returns an error (not allowed). • With Genero Mobile for Android, returns the IMEI if available, otherwise raises an error.	N/A	N/A	Yes	Yes
`ip`	Returns the IP address of the network interface used for the GUI connection. For mobile platforms, this is the preferred IP address of the device: if there is WIFI, either the IPv4 address is given back (for example: `192.168.0.12`) or if there is no IPv4 address, the IPv6 address is given back (for example: `2a02:810a:82c0:478:d462:e334:6a1d:fb78`). If there is no WIFI, either the cellular IPv4 or IPv6 address is given back. If there is no network, `NULL` is returned.	Yes	No	Yes	Yes
`isActiveX`	Returns "`1`" if the front-end runs in Active X mode (GDC specific). For Genero Mobile clients, the return value will always be "`0`"	Yes	No	Yes	Yes
`numScreens`	Number of screens available on the front-end platform. On typical front-end platforms and devices, the number of screens is 1. In some rare cases, a desktop computer can be configured with more than one screen.	Yes	No	No	No
`osType`	The operating system type where the front-end is running. Possible return values include "`WINDOWS`", "`LINUX`", "`OSX`", "`ANDROID`", "`IOS`".	Yes	Yes	Yes	Yes
`osVersion`	The version of the operating system. Example of returned values: "`4.3`", "`5.10.15`".	Yes	No	Yes	Yes
`outputMap`	Returns the GWC application output map of the current application. This option is only supported with a GAS >= 2.22.00. Example of returned value: "`DUA_HTML5`", ...	No	No	No	No
`ppi`	Returns the screen pixel density of the front-end platform (Pixels Per Inch). This front call takes an	Yes	No	Yes	Yes

Property name	Description	GDC	GWC-JS	GMA	GMI
	optional screen number as parameter (1 is the default). • With Genero Mobile for iOS, returns the `PixelsPerInch` of a iOS device. • With Genero Mobile for Android, returns the DPI (ppi == dpi)				
`screenResolution`	Returns the screen resolution of the front-end platform. This front call takes an optional screen number as parameter (1 is the default). Example of returned values: "`1200x1824`", "`1920x1104`". **Note:** For mobile devices, the value can change depending on the device orientation.	Yes	Yes	Yes	Yes
`target`	Returns the build platform target code name, identifying the operating system the front-end binary was compiled. This front call is provided for debugging purpose, do not base code on the returned value, it can change if the target OS version is upgraded for example. Use the `ostype` property instead. Example of returned values: • "`w32v100`" = Windows 32 bits, Visual C++ 10. • "`w64v110`" = Windows 64 bits, Visual C++ 11. • "`d32a040`" = Android 4.0 ARM 32 bits. • "`d32x040`" = Android 4.0 x86 32 bits. • "`i32a070`" = iOS 7.0 ARM 32 bits. • "`i32x070`" = iOS 7.0 x86 32 bits. **Note:** For GWC-JS, will return the same value as osType.	Yes	Yes	Yes	Yes
`windowSize`	Returns the current size of the front-end view-port. • For mobile front-ends, this is the size of the mobile screen. • For Genero Desktop Client, this is the size of the current window. • For Genero Web Client, this is the size of the browser webview. Example of returned values: "`1200x1824`", "`1920x1104`".	Yes	Yes	Yes	Yes

getEnv
Returns an environment variable set in the user session on the front end platform.

Syntax

```
ui.Interface.frontCall("standard", "getEnv",
 [name], [value])
```

1. *name* - The name of the environment variable.
2. *value* - Holds the value of the environment variable.

Usage

The "`getEnv`" front call returns an environment variable set in the user session on the front-end platform.

getWindowId
Returns the local window manager identifier of the window corresponding to the AUI window id passed as parameter.

Syntax

```
ui.Interface.frontCall("standard", "getWindowId",
 [aui-win-id], [loc-win-id])
```

1. *aui-win-id* - The id of the window node in the AUI tree.
2. *loc-win-id* - The id of the window in the window manager where the front-end is running.

Usage

Returns the local identifier that corresponds to the AUI window id passed as parameter, in the window manager where the front-end is displaying the application forms.

The node id must reference a `Window` node, otherwise "0" is returned ; in traditional mode, window widgets are simple frames ; Use "0" as `aui-win-id` parameter to get the top level window id in the local windowing system.

hardCopy
Prints a screen shot of the current window

Syntax

```
ui.Interface.frontCall("standard", "hardCopy",
 [pgsize], [result])
```

1. *pgsize* - Pass "1" to adapt the screen shot to the page size.
2. *result* - Holds the execution result (`TRUE`=success, `FALSE`=error).

Usage

The "`hardCopy`" front call allows you to print a screen shot of the current window.

The *pgsize* parameter is optional; Either leave out, or enter "1" to indicate that the screen shot must be adapted to the page size.

launchURL
Opens an URL with the default URL handler of the front-end.

Syntax

```
ui.Interface.frontCall("standard", "launchURL",
  [ url [, mode ] ], [] )
```

1. *url* - The URL to invoke.
2. *mode* (optional) - front-end specific meaning (see below).

Usage (General)

The "launchURL" front call opens an URL with the default URL handler available on the front-end platform; This is typically the web browser for "HTTP:" URLs, or the mailer for "mailto:" URLs, but the corresponding application may also be dedicated to the type of object specified by the URL (for example, a mapping service or to initiate a phone call).

This front call is a powerful feature: Front-end applications can register themselves as URL handlers, so you can start applications on the front-end through the lauchurl front call.

Supported schemes depend on your system configuration.

> **Important:** Some type of URLs are not supported by all front-end platforms. Make sure that you test all target front-ends when using a lauchurl front call.

The *mode* parameter is optional and is interpreted differently according to the front-end type:

- With Genero Web Client (GWC), use "replace" for the *mode* parameter, if you want the current application in the browser window or tab to be replaced with the new URL, instead of launching a new browser window or tab. If it is not present, or if a value other than "replace" is specified, the Genero Web Client behaves like the Genero Desktop Client, opening the URL in a new browser window.
- With Genero Mobile and Genero Desktop Client (GDC) front-ends, the *mode* parameter is ignored if specified.

Example

To invoke Google Play Store:

```
CALL ui.Interface.frontCall("standard", "launchURL",
 ["market://details?id=com.google.android.apps.currents"], [])
```

```
CALL ui.Interface.frontCall("standard", "launchURL",
 ["market://details?id=com.google.zxing.client.android"], [])
```

To open Google Maps:

```
CALL ui.Interface.frontCall("standard", "launchURL",
 ["geo:48.613363,7.711083?z=17"], [])
```

To open Google Street View:

```
CALL ui.Interface.frontCall("standard", "launchURL",
 ["google.streetview:cbll=48.613363,7.711083&cbp=1,0,,0,1.0&mz=17"],
 [])
```

To initiate a phone call:

```
CALL ui.Interface.frontCall("standard", "launchURL", ["tel:
+336717623"], [])
```

mdClose
Unloads a DLL or shared library front call module.

Syntax

```
ui.Interface.frontCall("standard", "mdClose",
 [name], [result])
```

1. *name* - The name of the module to be closed.
2. *result* - Holds the result (0 = success, -1 = module not found, -2 = cannot unload (busy)).

Usage
Front call modules are loaded on demand. After calling a function of a specific module, you can use the "`mdClose`" front call to unload the shared library and save resources.

openDir
Displays a file dialog window to get a directory path on the local file system.

Syntax

```
ui.Interface.frontCall("standard", "openDir",
 [path,caption], [result])
```

1. *path* - The default path.
2. *caption* - The caption to be displayed.
3. *result* - Holds the name of the selected directory (or `NULL` if canceled).

Usage
When invoking the "`openDir`" front call, the front-end displays the typical file dialog window on the local file system, to let the end user enter a directory path.

If the user cancels the dialog, the front call returns `NULL` in the result variable.

openFile
Displays a file dialog window to get a path to open a file on the local file system.

Syntax

```
ui.Interface.frontCall("standard", "openFile",
 [path,name,wildcards,caption],
 [result])
```

1. *path* - The default path.
2. *name* - The name to be displayed for the file type.
3. *wildcards* - A blank separated list of wildcards (for ex: "*.pdf" or "README* test*.txt")
4. *caption* - The caption to be displayed.
5. *result* - Holds the name of the selected file (or `NULL` if canceled).

Usage

When invoking the "`openFile`" front call, the front-end displays a file dialog window using the local file system, to let the end user enter a file path, to select an existing file.

If the user cancels the dialog, the front call returns `NULL` in the result variable.

> **Note:** With the GWS-JS front-end, the *path* parameter is ignored, and *wildcards* can only be hold one type of file extension.

playSound
Plays the sound file passed as parameter on the front-end platform.

Syntax

```
ui.Interface.frontCall("standard", "playSound",
  [filename], [])
```

1. *filename* - The sound file to play.

Usage

The "`playSound`" front call opens the sound file passed as parameter and plays the sound on the front-end.

If the file is not located on the front-end, it will automatically be transfered to the front-end through the file-transfer facility.

Supported sound file format depends on the front-end infastructure (platform, technology, web browser, ...)

Example

```
CALL ui.Interface.frontCall("standard", "playSound",
    ["/opt/var/sounds/beep.mp3"], [])
```

saveFile
Displays a file dialog window to get a path to save a file on the local file system.

Syntax

```
ui.Interface.frontCall("standard", "saveFile",
  [path,name,filetype,caption],
  [result])
```

1. *path* - The default path.
2. *name* - The name to be displayed for the file type.
3. *filetype* - The file types (as a blank separated list of extensions).
4. *caption* - The caption to be displayed.
5. *result* - Holds the name of the selected file (or `NULL` if canceled).

Usage
When invoking the "`saveFile`" front call, the front-end displays the typical file dialog window on the local file system, to let the end user enter a file path, to save data to a new file.

If the user cancels the dialog, the front call returns `NULL` in the result variable.

setReportFont
Override the font used for report generation for the current application.

Syntax

```
ui.Interface.frontCall("standard", "setReportFont",
  [font], [result])
```

1. *font* - A string describing the font to use for report generation (see Usage for details).
2. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage
The "setReportFont" front call allows you to override the font used for report generation for the current application. You can simply copy/paste the font string from the "Report To Printer" font panel from GDC Monitor. An empty or null string reset to the default behavior.

The *font* parameter is a string that describe the font to use for report generation. For example: "Helvetica, Bold, Italic, 13". Alternatively, you can specify "<ASK_ONCE>" , "<ASK_ALWAYS>" , "<USER_DEFINED>" or "<USE_DEFAULT>" " which will perform the corresponding actions.

setReportPrinter
Override the printer configuration used for report generation for the current application.

Syntax

```
ui.Interface.frontCall("standard", "setReportPrinter",
  [printer], [result])
```

1. *printer* - A string describing the printer to use for report generation (see Usage for details).
2. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage
The "setReportPrinter" front call allows you to override the printer configuration used for report generation for the current application. You can simply copy/paste the printer string from the "Report To Printer" printer panel from GDC Monitor. An empty or null string reset to the default behavior.

The *printer* parameter is a string that describe the printer to use for report generation. For example: "moliere, Portrait, A4, 96 dpi, 1 copy, Ascendent, Color, Auto". Alternatively, you can specify "<ASK_ONCE>" , "<ASK_ALWAYS>" , "<USER_DEFINED>" or "<USE_DEFAULT>" " which will perform the corresponding actions.

setWebComponentPath
Defines the base path where web components are located.

Syntax

```
ui.Interface.frontCall("standard", "setWebComponentPath",
  [path], [])
```

1. *path* - The base url. For example, "http://myserver/components" or "file:///c:/components".

Usage

This front call defines the base path to find gICAPI web components files.

For the Genero Desktop Client, it defines the base path where web components are located, when GDC is directly connected to the runtime system. This is ignored when GDC is connected to the GAS.

For Genero Mobile, it sets the main web component lookup path. An URI is expected. For example, "file:///data/data/com.fourjs.gma/cache/appdata/mywebcomponents"or "http://mygas/mywebcomponents/".

shellExec
Opens a file on the front-end platform with the program associated to the file extension.

Syntax

```
ui.Interface.frontCall("standard", "shellExec",
  [document, action], [result])
```

1. *document* - The document file to be opened.
2. *action* - (optional, Windows™ Only!) The action to perform, related to the way the file type is registered in Windows™ Registry.
3. *result* - Holds the execution result (TRUE=success, FALSE=error).

Usage
The "shellExec" front call opens a file on the front-end platform with the program associated to the file extension.

This front call is mainly designed for the Genero Desktop Client on Windows™ platforms.

> **Important:** Under X11 Systems, this uses xdg-open, which needs to be installed and configured on your system. Kfmclient will be used as a workaround when xdg-open is not available.
>
> **Tip:** In order to view a document (like a PDF for example), if that document can be displayed by web browsers, use the launchURL on page 1898 front call instead, especially if you want to use both the Genero Desktop Client (GDC) and the Genero Web Client (GWC) front-ends.

Webcomponent front calls
This section describes Webcomponent specific front calls.

This table shows the functions provided by the "webcomponent" module on the front-ends supporting web components.

Table 426: Webcomponent module front-end functions

Function Name	Description
`ui.Interface.frontCall("webcomponent", "call", [aui-name, function-name, [ param1, param2, ... ] ], [result] )`	Calls a JavaScript function through the web component.
`ui.Interface.frontCall("webcomponent", "frontCallAPIVersion", [],[result])`	Returns the API version of web component front-end calls.
`ui.Interface.frontCall("webcomponent", "getTitle", [aui-name], [result] )`	Returns the title of the HTML doc rendered by a web component.

call
Calls a JavaScript function through the web component.

Syntax

```
ui.Interface.frontCall("webcomponent", "call",
```

```
  [aui-name, function-name, [ param1, param2, ... ] ],
  [result]
)
```

1. *aui-name* - This is the name of the web component name in the AUI tree.
2. *function-name* - This is the name of the web component JavaScript function to be called.
3. *param1*, *param2*, ... - Optional parameters to be passed to the web component JavaScript function.
4. *result* - Holds the JavaScript function return value.

Usage

Calls a JavaScript function through the web component. The JavaScript function must be implemented in the HTML content pointed by the URL-based web component, or in the user-defined JavaScript of a gICAPI-based web component.

The *aui-name* and *function-name* arguments are mandatory.

The arguments following the *function-name* argument will be passed to the JavaScript function.

That the *result* variable will contain the value returned by the JavaScript function.

Example

```
DEFINE result STRING
CALL ui.Interface.frontCall("webcomponent","call",
        ["formonly.data","echoString","abcdef"],[result])
```

For a complete example, see Example 2: Calling a JavaScript function of a gICAPI web component on page 1432.

frontCallAPIVersion
Returns the API version of web component front-end calls.

Syntax

```
ui.Interface.frontCall("webcomponent", "frontCallAPIVersion",
  [],[result])
```

1. *result* - Holds the API version for web component front calls.

Usage

This front call can be used to check the API version for the web component front calls.

If the API version changes, you must adapt the code to the expected front call API implemented for the web components.

The value returned by this front call is a typicaly version number such as `1.0`, `1.1`, etc.

Example

```
FUNCTION wc_api_version()
    DEFINE vers STRING
    TRY
       CALL
 ui.Interface.frontCall("webcomponent","frontCallAPIVersion",[],
[vers])
       -- we can safely call "webcomponent" "call" in the code
       RETURN vers
    CATCH
       -- we can't call the "webcomponent" functions...
```

```
        RETURN 0
    END TRY
END FUNCTION
```

getTitle
Returns the title of the HTML doc rendered by a web component.

Syntax

```
ui.Interface.frontCall("webcomponent", "getTitle",
  [aui-name], [result] )
```

1. *aui-name* - This is the name of the web component name in the AUI tree.
2. *result* - Holds the title of of the HTML document.

Usage

This front call can be used to get the title of the HTML document that is rendered by the web component identified by the *aui-name*. For more details refer to http://www.w3schools.com/tags/tag_title.asp.

A typical usage of this front call is when implementing a web component based on the O-Auth mechanism to identify the current user: For example, with the Google accounts authentication service, after the login and password were validated by Google, the authentication token will be returned in the title of the HTML document. This token is typically used by the application to identify the user in distant API calls.

Genero Desktop Client front calls
This section describes GDC specific front calls.

The GDC front-end implements the following front call modules:

- Windows DDE Support on page 1904
- Windows COM Support on page 1910
- Windows Mail extension on page 1917

> **Important:** These front call modules are only available on Windows™ platforms.

Windows™ DDE Support
Description of Windows™ DDE support.

> **Important:** The Win DDE front call library is deprecated.

Dynamic Data Exchange (DDE) is a form of inter-process communication implemented by Microsoft™ for Windows™ platforms. DDE uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in applications that send updates to one another as new data becomes available.

Please refer to your Microsoft™ documentation for DDE compatibility between existing versions. As an example, DDE commands were changed between Office 97 and Office 98.

We provide a DDE interface as a Front-End Extension: WinDDE.DLL

- Using the WinDDE API on page 1905
- The DDE API function list on page 1905
- WinDDE example on page 1909

Using the WinDDE API

With WinDDE Support, you can invoke a Windows™ application and send or receive data to or from it. To use this functionality, the program must use the Windows™ Front End.

Before using the DDE functions, the TCP communication channel between the application and the front end must be established with a display (OPEN WINDOW, MENU, DISPLAY TO).

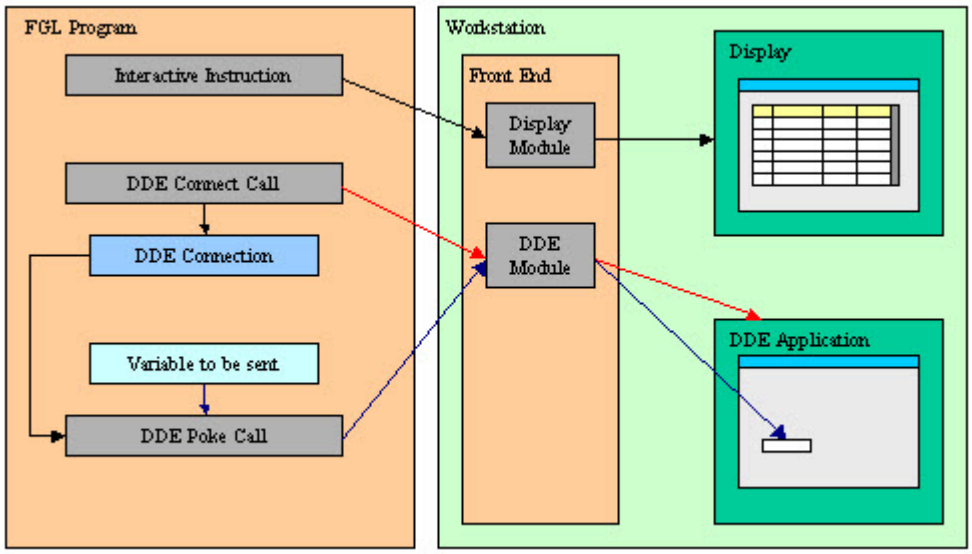


Figure 103: The four-part procedure of the DDE API

The DDE API is used in a four-part procedure, as described in the following steps:

1. The application sends to the Front End the DDE order using the TCP/IP channel.
2. The Front End executes the DDE order and sends the data to the Windows™ application through the DDE API.
3. The Windows™ application executes the command and sends the result, which can be data or an error code, to the Front End.
4. The Windows™ Front End sends back the result to the application using the TCP/IP channel.

A DDE connection is uniquely identified by two values: The name of the DDE Application and the document. Most DDE functions require these two values to identify the DDE source or target.

The DDE API function list

The DDE API is based on the front call technique.

The DDE API is based on the front call technology. All DDE functions are grouped in the **WINDDE** front end function module.

Table 427: Windows DDE front-end functions

Function name	Description
`CALL ui.Interface.frontCall("WINDDE","DDEConnect", [ program, document, encoding ], [result] )`	`DDEConnect` opens a DDE connection.
`CALL ui.Interface.frontCall("WINDDE","DDEExecute",`	`DDEExecute` executes a DDE command.

Function name	Description
`[ program, document, command, encoding ], [result] )`	
`CALL ui.Interface.frontCall("WINDDE","DDEFinish", [ program, document ], [result] )`	`DDEFinish` closes a DDE connection.
`CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [result] )`	`DDEFinishAll` closes all DDE connections.
`CALL ui.Interface.frontCall("WINDDE","DDEError", [], [errmsg] )`	`DDEError` returns error information about the last DDE operation.
`CALL ui.Interface.frontCall("WINDDE","DDEPeek", [ program, container, cells, encoding ], [ result, value ] )`	`DDEPeek` retrieves data from the specified program and document using the DDE channel.
`CALL ui.Interface.frontCall("WINDDE","DDEPoke", [ program, container, cells, values, encoding ], [result] )`	`DDEPoke` sends data to the specified program and document using the DDE channel.

DDEConnect

`DDEConnect` opens a DDE connection.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEConnect",
 [ program, document, encoding ], [result] )
```

- *program* is the name of the DDE application.
- *document* is the document that is to be opened.
- *encoding* is an optional parameter. It allows to force the encoding to use between ASCII and wide char/ unicode. When not specified, WinDDE will try to retrieve the correct encoding by itself. Possible values are:
  - UNICODE
  - ASCII
- *result* is an integer variable receiving the status.
- *result* is TRUE if the function succeeded, FALSE otherwise.
- If the function failed, use `DDEError` to get the description of the error.

Warnings

- If the function failed with DMLERR_NO_CONV_ESTABLISHED, then the DDE application was probably not running. Use the execute or shellexec front call to start the DDE application.
- In Microsoft™ Office 2010, the use of DDE is disabled by default. You need to uncheck **Ignore other applications that use Dynamic Data Exchange(DDE)** in advanced options, otherwise DDEConnect will fail.

DDEExecute

`DDEExecute` executes a DDE command.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEExecute",
  [ program, document, command, encoding ], [result] )
```

- *program* is the name of the DDE application.
- *document* is the document that is to be used.
- *command* is the command that needs to be executed.
- *encoding* is an optional parameter. It allows to force the encoding to use between ASCII and wide char/unicode. When not specified, WinDDE will try to retrieve the correct encoding by itself. Possible values are: "UNICODE", "ASCII"
- Refer to the *program* documentation to know the syntax of *command*.
- *result* is an integer variable receiving the status.
- *result* is TRUE if the function succeeded, FALSE otherwise.
- If the function failed, use DDEError to get the description of the error.

Warnings

- The DDE connection must be opened see DDEConnect.

DDEFinish

`DDEFinish` closes a DDE connection.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEFinish",
  [ program, document ], [result] )
```

- *program* is the name of the DDE application.
- *document* is the document that is to be closed.
- *result* is an integer variable receiving the status.
- *result* is TRUE if the function succeeded, FALSE otherwise.
- If the function failed, use `DDEError` to get the description of the error.

Warnings

- The DDE connection must be opened, see `DDEConnect`.

DDEFinishAll

`DDEFinishAll` closes all DDE connections.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEFinishAll",
  [], [result] )
```

- *result* is TRUE if the function succeeded, FALSE otherwise.

Usage

This function closes all DDE connections, as well as the DDE server program.

DDEError

`DDEError` returns error information about the last DDE operation.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEError",
  [], [errmsg] )
```

- *errmsg* is the error message. It is set to NULL if no error occurred.

DDEPeek

`DDEPeek` retrieves data from the specified program and document using the DDE channel.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEPeek",
  [ program, container, cells, encoding ], [ result, value ] )
```

- *program* is the name of the DDE application.
- *container* is the document or sub-document that is to be used. A sub-document can, for example, be a sheet in Microsoft™ Excel.
- *cells* represents the working items; see the *program* documentation to know the format of *cells*.
- *encoding* is an optional parameter. It allows to force the encoding to use between ASCII and wide char/unicode. When not specified, WinDDE will try to retrieve the correct encoding by itself. Possible values are: "UNICODE", "ASCII"
- *value* represents the data to be retrieved; see the *program* documentation to know the format of*values*.
- *result* is an integer variable receiving the status.
- *result* is TRUE if the function succeeded, FALSE otherwise.
- If the function failed, use `DDEError` to get the description of the error.
- *value* is a variable receiving the cells values.

Warnings

- The DDE connection must be opened; see `DDEConnect`.
- `DDEError` can only be called once to check if an error occurred.

DDEPoke

`DDEPoke` sends data to the specified program and document using the DDE channel.

Syntax

```
CALL ui.Interface.frontCall("WINDDE","DDEPoke",
  [ program, container, cells, values, encoding ], [result] )
```

- *program* is the name of the DDE application.
- *container* is the document or sub-document that is to be used. A sub-document can, for example, be a sheet in Microsoft™ Excel.
- *cells* represents the working items; see the *program* documentation to know the format of *cells*.
- *values* represents the data to be sent; see the *program* documentation to know the format of *values*.

- *encoding* is an optional parameter. It allows to force the encoding to use between ASCII and wide char/ unicode. When not specified, WinDDE will try to retrieve the correct encoding by itself. Possible values are: "UNICODE", "ASCII"
- *result* is an integer variable receiving the status.
- *result* is TRUE if the function succeeded, FALSE otherwise.
- If the function failed, use `DDEError` to get the description of the error.

Warnings

- The DDE connection must be opened; see `DDEConnect`.
- An error may occur if you try to set many (thousands of) cells in a single operation.

WinDDE example
This section provides a WinDDE example.

`dde_example.per`

```
DATABASE formonly
SCREEN
{
Value to be given to top-left corner :
[f00                                ]
Value found on top-left corner :
[f01                                ]
}
ATTRIBUTES
  f00 = formonly.val;
  f01 = formonly.rval, NOENTRY;
```

`dde_example.4gl`

```
MAIN
  -- Excel must be open beforehand
  CONSTANT file = "Sheet1"
  CONSTANT prog = "EXCEL"
  DEFINE val, rval STRING
  DEFINE res INTEGER
  OPEN WINDOW w1 AT 1,1 WITH FORM "dde_example.per"
  INPUT BY NAME val
  CALL ui.Interface.frontCall("WINDDE","DDEConnect", [prog,file], [res] )
  CALL checkError(res)
  CALL ui.Interface.frontCall("WINDDE","DDEPoke", [prog,file,"R1C1",val],
 [res] );
  CALL checkError(res)
  CALL ui.Interface.frontCall("WINDDE","DDEPeek", [prog,file,"R1C1"],
 [res,rval] );
  CALL checkError(res)
  DISPLAY BY NAME rval
  INPUT BY NAME val WITHOUT DEFAULTS
  CALL ui.Interface.frontCall("WINDDE","DDEExecute", [prog,file,"[save]"],
 [res] );
  CALL checkError(res)
  CALL ui.Interface.frontCall("WINDDE","DDEFinish", [prog,file], [res] );
  CALL checkError(res)
  CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [res] );
  CALL checkError(res)
  CLOSE WINDOW w1
END MAIN

FUNCTION checkError(res)
  DEFINE res INTEGER
  DEFINE mess STRING
```

```
  IF res THEN RETURN END IF
  DISPLAY "DDE Error:"
  CALL ui.Interface.frontCall("WINDDE","DDEError",[],[mess]);
  DISPLAY mess
  CALL ui.Interface.frontCall("WINDDE","DDEFinishAll", [], [res] );
  DISPLAY "Exit with DDE Error."
  EXIT PROGRAM (-1)
END FUNCTION
```

Windows™ COM Support
"COM" stands for Component Object Model. It allows anyone to directly access Windows™ Applications Objects. You can create instances of those objects, call methods on them, and get or set their properties.

Important: The WinCOM front call library is deprecated.

- Using the WinCOM API on page 1910
- The WinCOM API function list on page 1910
- WinCOM examples on page 1913

Using the WinCOM API
With WinCOM Support, you can invoke a Windows™ application and send or receive data to or from it.

To use this functionality, the program must use the Windows™ Front End.

The WinCOM API function list
The WinCOM API is based on the front call technique as described in Front End Functions. All WinCOM functions are grouped in the WinCOM front end function module.

Table 428: Windows COM front-end functions

Function name	Description
`CALL ui.Interface.frontCall("WinCOM","CreateInstance", [ program ], [handle] )`	The `CreateInstance` function creates an instance of a registered COM object.
`CALL ui.Interface.frontCall("WINCOM","CallMethod", [ handle, method, arg1, ... ], [result] )` `CALL ui.Interface.frontCall("WINCOM","CallMethod", [ handle, method(arg1, ...) ], [result] )`	The `CallMethod` function calls a method on a specified object.
`CALL ui.Interface.frontCall("WINCOM","GetProperty", [ handle, member ], [result] )`	The `GetProperty` function gets a property of an object.
`CALL ui.Interface.frontCall("WINCOM","SetProperty",`	The `SetProperty` function sets a property of an object.

Function name	Description
[*handle*, *member*, *value*], [*result*])	
CALL ui.Interface.frontCall("WINCOM","GetError", [], [*result*])	The GetError function gets a description of the last error which occurred.
CALL ui.Interface.frontCall("WINCOM","ReleaseInstance", [*handle*], [*result*])	The ReleaseInstance function releases an instance of a COM object.

Supported syntax

COM language syntax is very flexible and allows lots of notation. Genero WinCOM API is slightly more strict:

- := notation **is allowed** in version 2.00.1e (or later) **only**; for instance: `myFunction(SourceType:=3)`
- "no parenthesis" notation **is not allowed**; for instance: `myFunction 3` must be written`myFunction(3)`
- numeric constants **are allowed** in version 2.00.1e (or later) **only**. The constant list depends on the application used via WinCOM, therefore the list is configurable: a file named etc/WinCOM.cst gathers all the constants provided today by Microsoft™ for Office XP. It can be modified to add user-defined constants. Example with Word: `CALL ui.Interface.frontCall("WINCOM","SetProperty", [wdapp,"Selection.Font.Bold","9999998"],[wddoc])` Here, "9999998" stands for the constant **wdToggle** (see etc/WinCOM.cst).
- There is no way to handle an array as a method argument. This is also due to BDL limitation: you can't pass BDL Arrays to frontcalls.

CreateInstance

The `CreateInstance` function creates an instance of a registered COM object.

Syntax

```
CALL ui.Interface.frontCall("WinCOM","CreateInstance",
  [ program ], [handle] )
```

- *program* is the classname of the registered COM object.
- *handle* is an integer variable receiving the status.
- *handle* is -1 if there as an error, otherwise an integer value that can be used for a later call to the API.
- If the function failed, use `GetError` to get the description of the error.

CallMethod

The `CallMethod` function calls a method on a specified object.

Syntax

```
CALL ui.Interface.frontCall("WINCOM","CallMethod",
  [ handle, method, arg1, ... ], [result] )
```

OR

```
CALL ui.Interface.frontCall("WINCOM","CallMethod",
  [ handle, method(arg1, ...) ], [result] )
```

- *handle* is the handle returned by another front call (`CreateInstance`, `CallMethod`, `GetProperty`).
- *method* is the member name to call.
- *arg1 (and ...)* are the arguments to pass to the method call. Depending on the syntax allowed by the version of the program you're interacting with, arguments might be used inside brackets or outside. The best way for Microsoft™ applications (such as Excel or Word) is to initially test your code with a macro of the manipulation you're expecting to do. According to the method which is used, arguments may or may not be optional.
- *result* is either a handle or a value of a predefined type.
- *result* is -1 in case of error (use `GetError` to get the description of the error).

GetProperty

The `GetProperty` function gets a property of an object.

Syntax

```
CALL ui.Interface.frontCall("WINCOM","GetProperty",
  [ handle, member ], [result] )
```

- *handle* is the handle returned by another front call (`CreateInstance`, `CallMethod`, `GetProperty`).
- *member* is the member property name to get.
- *result* is either a handle or a value of a predefined type.
- *result* is -1 in case of error (use `GetError` to get the description of the error).

SetProperty

The `SetProperty` function sets a property of an object.

Syntax

```
CALL ui.Interface.frontCall("WINCOM","SetProperty",
  [handle, member, value], [result] )
```

- *handle* is the handle returned by another front call (`CreateInstance`, `CallMethod`, `GetProperty`).
- *member* is the member property name to set.
- *value* is the value to which the property will be set.
- *result* is -1 in case of error (use `GetError` to get the description of the error), otherwise it is 0.

GetError

The `GetError` function gets a description of the last error which occurred.

Syntax

```
CALL ui.Interface.frontCall("WINCOM","GetError",
  [], [result] )
```

- *result* is the description of the last error.
- the returned value is NULL if there was no error.

ReleaseInstance

The `ReleaseInstance` function releases an Instance of a COM object.

Syntax

```
CALL ui.Interface.frontCall("WINCOM","ReleaseInstance",
  [handle], [result] )
```

- *handle* is the handle returned by another front call (`CreateInstance`, `CallMethod`, `GetProperty`).

- *result* is -1 in case of error (use `GetError` to get the description of the error), otherwise it is 0.

WinCOM examples
Various WinCOM examples.

- Wincom and Excel example on page 1913
- Wincom and Word example on page 1914
- Wincom and Outlook example on page 1915
- Wincom and Internet Explorer example on page 1916

Wincom and Excel example

This section provides a Wincom and Excel example.

This example puts "foo" in the first row of the 1st column of an Excel Sheet.

```
DEFINE xlapp INTEGER
DEFINE xlwb INTEGER
MAIN
  DEFINE result INTEGER
  DEFINE str STRING
--initialization of global variables
  LET xlapp = -1
  LET xlwb = -1
--first, we must create an Instance of an Excel Application
  CALL ui.Interface.frontCall("WinCOM", "CreateInstance",
  ["Excel.Application"], [xlapp])
  CALL CheckError(xlapp, __LINE__)
--then adding a Workbook to the current document
  CALL ui.interface.frontCall("WinCOM", "CallMethod",
  [xlapp, "WorkBooks.Add"], [xlwb])
  CALL CheckError(xlwb, __LINE__)
--then, setting it to be visible
  CALL ui.interface.frontCall("WinCOM", "SetProperty",
  [xlapp, "Visible", true], [result])
  CALL CheckError(result, __LINE__)
--then CALL SetProperty to set the value of the cell
  CALL ui.Interface.frontCall("WinCOM", "SetProperty",
  [xlwb, 'activesheet.Range("A1").Value', "foo"],[result])
  CALL CheckError(result, __LINE__)
--then CALL GetProperty to check the value again
  CALL ui.Interface.frontCall("WinCOM", "GetProperty",
  [xlwb, 'activesheet.Range("A1").Value'], [str])
  CALL CheckError(str, __LINE__)
  DISPLAY "content of the cell is: " || str

 --then Free the memory on the client side
  CALL freeMemory()
END MAIN

FUNCTION freeMemory()
 DEFINE res INTEGER
  IF xlwb != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [xlwb], [res] )
  END IF
  IF xlapp != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [xlapp], [res] )
  END IF
END FUNCTION

FUNCTION checkError(res, lin)
  DEFINE res INTEGER
  DEFINE lin INTEGER
  DEFINE mess STRING
```

```
  IF res = -1 THEN
    DISPLAY "COM Error for call at line:", lin
    CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])
    DISPLAY mess
--let's release the memory on the GDC side
    CALL freeMemory()
    DISPLAY "Exit with COM Error."
    EXIT PROGRAM (-1)
  END IF
END FUNCTION
```

Wincom and Word example

This section provides a Wincom and Word example.

This example puts "This is a title" centered on the page, underlined, and in bold.

```
DEFINE wdapp INTEGER
DEFINE wddoc INTEGER
MAIN
  DEFINE result INTEGER
--initialization of global variables
  LET wdapp = -1
  LET wddoc = -1
--first, we must create an Instance of a Word Application
  CALL ui.Interface.frontCall("WINCOM","CreateInstance",
   ["Word.Application"],[wdapp])
  CALL CheckError(wdapp, __LINE__)
--then adding a document
  CALL ui.Interface.frontCall("WINCOM","CallMethod",
   [wdapp,"Documents.Add"],[wddoc])
  CALL CheckError(wddoc, __LINE__)
--then, setting it to be visible
  CALL ui.Interface.frontCall("WINCOM","SetProperty",
   [wdapp,"Visible",true],[result])
  CALL CheckError(result, __LINE__)
--Centering the cursor for the title
  CALL ui.Interface.frontCall("WINCOM","SetProperty",
   [wdapp,"Selection.ParagraphFormat.Alignment","1"],[wddoc])
  CALL CheckError(wddoc, __LINE__)
--Underlining the title
  CALL ui.Interface.frontCall("WINCOM","SetProperty",
   [wdapp,"Selection.Font.Underline","1"],[wddoc])
  CALL CheckError(wddoc, __LINE__)
--Putting the title in bold
  CALL ui.Interface.frontCall("WINCOM","SetProperty",
   [wdapp,"Selection.Font.Bold","9999998"],[wddoc])
  CALL CheckError(wddoc, __LINE__)
--Typing the title's text
  CALL ui.Interface.frontCall("WINCOM","CallMethod",
   [wdapp,'Selection.TypeText("This is a title")'],[wddoc])
  CALL CheckError(wddoc, __LINE__)
--then Free the memory on the client side
  CALL freeMemory()
END MAIN

FUNCTION freeMemory()
  DEFINE res INTEGER
  IF wddoc != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [wddoc], [res] )
  END IF
  IF wdapp != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [wdapp], [res] )
  END IF
```

```
END FUNCTION

FUNCTION checkError(res, lin)
  DEFINE res INTEGER
  DEFINE lin INTEGER
  DEFINE mess STRING
  IF res = -1 THEN
    DISPLAY "COM Error for call at line:", lin
    CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])
    DISPLAY mess
--let's release the memory on the GDC side
    CALL freeMemory()
    DISPLAY "Exit with COM Error."
    EXIT PROGRAM (-1)
  END IF
END FUNCTION
```

Wincom and Outlook example

This section provides a Wincom and Outlook example.

This example executes Outlook, creates a new contact, and saves it in your contact list.

```
DEFINE outapp INTEGER
DEFINE outit INTEGER
DEFINE outcon INTEGER
DEFINE outsav INTEGER
MAIN
  DEFINE result INTEGER
  DEFINE str STRING
--initialization of global variables
  LET outapp = -1
  LET outit = -1
  LET outcon = -1
  LET outsav = -1
--first, we must create an Instance of an Outlook Application
  CALL ui.interface.frontcall("WinCOM", "CreateInstance",
   ["Outlook.Application"], [outapp])
  CALL CheckError(outapp, __LINE__)
--then, creating a contact object
  CALL ui.interface.frontcall("WinCOM", "CallMethod",
   [outapp, "CreateItem(olContactItem)"], [outit])
  CALL CheckError(outit, __LINE__)
--then, displaying the contact form
  CALL ui.interface.frontCall("WinCOM", "CallMethod",
   [outit, "Display"], [outcon])
  CALL CheckError(outcon, __LINE__)
--CALL SetProperty to fill the various fields with the values you expect
#First Name
  CALL ui.interface.frontCall("WinCOM", "SetProperty",
   [outit, "FirstName", "Lionel"], [result])
  CALL CheckError(result, __LINE__)
#1st email address
  CALL ui.interface.frontCall("WinCOM", "SetProperty",
   [outit, "Email1Address", "lif@4js.com"], [result])
  CALL CheckError(result, __LINE__)
#Business address
  CALL ui.interface.frontCall("WinCOM", "SetProperty",
   [outit, "BusinessAddress", "1 rue de Berne"], [result])
  CALL CheckError(result, __LINE__)
--then, CALL GetProperty to check the values again
  CALL ui.Interface.frontCall("WinCOM", "GetProperty",
   [outit, "FirstName"], [str])
  CALL CheckError(str, __LINE__)
```

```
  DISPLAY "First Name of the new contact is " || str
  CALL ui.Interface.frontCall("WinCOM", "GetProperty",
   [outit, "Email1Address"], [str])
  CALL CheckError(str, __LINE__)
  DISPLAY "1st email of the new contact is " || str
  CALL ui.Interface.frontCall("WinCOM", "GetProperty",
   [outit, "BusinessAddress"], [str])
  CALL CheckError(str, __LINE__)
  DISPLAY "Business Address of the new contact is " || str
--at the end, saving the contact
  CALL ui.interface.frontCall("WinCOM", "CallMethod", [outit, "Save"],
 [outsav])
  CALL CheckError(outsav, __LINE__)
--then Free the memory on the client side
  CALL freeMemory()
END MAIN

FUNCTION freeMemory()
  DEFINE res INTEGER
  IF outit != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [outit], [res] )
  END IF
  IF outapp != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [outapp],
 [res] )
  END IF
END FUNCTION

FUNCTION checkError(res, lin)
  DEFINE res INTEGER
  DEFINE lin INTEGER
  DEFINE mess STRING
  IF res = -1 THEN
    DISPLAY "COM Error for call at line:", lin
    CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])
    DISPLAY mess
--let's release the memory on the GDC side
    CALL freeMemory()
    DISPLAY "Exit with COM Error."
    EXIT PROGRAM (-1)
  END IF
END FUNCTION
```

Tip: You may find the various Outlook objects (such as *ContactItem* object), methods (such as the*CreateItem* method), and properties (such as the *FirstName* or *BusinessAddress* properties) on the Microsoft™ Developer Network.

Wincom and Internet Explorer example

This section provides a Wincom and Internet Explorer example.

This example executes Internet Explorer on a defined URL with the address bar masked.

```
DEFINE ieapp INTEGER
DEFINE ienav INTEGER
MAIN
  DEFINE result INTEGER
--initialization of global variables
  LET ieapp = -1
  LET ienav = -1
--first, we must create an Instance of Internet Explorer application
  CALL ui.Interface.frontCall("WinCOM", "CreateInstance",
   ["InternetExplorer.Application"], [ieapp])
  CALL CheckError(ieapp, __LINE__)
```

```
--then, specifying the URL you want to load
  CALL call ui.interface.frontCall("WinCOM", "CallMethod",
   [ie_app, "Navigate", "www.4js.com"], [ienav])
  CALL CheckError(ienav, __LINE__)
--then, masking the address bar
  CALL ui.interface.frontCall("WinCOM", "SetProperty",
   [ieapp, "AddressBar", false], [result])
  CALL CheckError(result, __LINE__)
--then, setting it to visible
  CALL ui.interface.frontCall("WinCOM", "SetProperty", [ieapp, "Visible",
 true],
   [result])
  CALL CheckError(result, __LINE__)
--then Free the memory on the client side
  CALL freeMemory()
END MAIN

FUNCTION freeMemory()
  DEFINE res INTEGER
  IF ienav != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [ienav], [res] )
  END IF
  IF ieapp != -1 THEN
    CALL ui.Interface.frontCall("WinCOM","ReleaseInstance", [ieapp], [res] )
  END IF
END FUNCTION

FUNCTION checkError(res, lin)
  DEFINE res INTEGER
  DEFINE lin INTEGER
  DEFINE mess STRING
  IF res = -1 THEN
    DISPLAY "COM Error for call at line:", lin
    CALL ui.Interface.frontCall("WinCOM","GetError",[],[mess])
    DISPLAY mess
--let's release the memory on the GDC side
    CALL freeMemory()
    DISPLAY "Exit with COM Error."
    EXIT PROGRAM (-1)
  END IF
END FUNCTION
```

Windows™ Mail extension

Description of the Windows™ Mail extension.

> **Important:** The WinMAIL front call library is deprecated. If used, the GDC executable format must match the WinAPI executable format: The 32-bit WinAPI can only be used with a 32-bit GDC.

Send mail using MAPI

MAPI is an acronym for Messaging Application Programming Interface. The **MAPI** extension will create a new mail in the default mailer software, which needs to be "MAPI" compatible, and ask the user to send the mail. The mail sent using MAPI will be stored by the default mailer software in the same way as any other mail created by the user.

Send mail using an SMTP server

Another method of sending mail is to connect directly to an **SMTP** server (Simple Mail Transfer Protocol is the de facto standard for email transmission across the Internet). The extension will connect to a given **SMTP** server and send the mail through this server. The mail is not kept on the client side.

- The WinMail API on page 1918

- WinMail examples on page 1923

The WinMail API

The WinMail API is based on the front call technique as described in Front End Functions. All WinMail functions are grouped in the WinMail front end function module.

Table 429: WinMail front-end functions: General

Function name	Description
`CALL ui.Interface.frontCall("WinMail","Init", [], [id ] )`	The `Init` function initializes the module.
`CALL ui.Interface.frontCall("WinMail","Close", [id], [ result ] )`	The `Close` function clears all information corresponding to a message, and frees the memory occupied by the message.
`CALL ui.Interface.frontCall("WinMail","SetBody", [ id, body ], [ result ] )`	The `SetBody` function sets the body of the mail.
`CALL ui.Interface.frontCall("WinMail","SetSubject", [ id, subject ], [ result ] )`	The `SetSubject` function sets the subject of the mail.
`CALL ui.Interface.frontCall("WinMail","AddTo", [ id, name, address ], [ result ] )`	The `AddTo` function adds a "To" addressee to the mail.
`CALL ui.Interface.frontCall("WinMail","AddCC", [ id, name, address ], [ result ] )`	The `AddCC` function adds a "CC" addressee to the mail.
`CALL ui.Interface.frontCall("WinMail","AddBCC", [ id, name, address ], [ result ] )`	The `AddBCC` function adds a "BCC" addressee to the mail.
`CALL ui.Interface.frontCall("WinMail","AddAttachment", [ id, fileName], [ result ] )`	The `AddAttachment` function adds a file as an attachment to the mail.
`CALL ui.Interface.frontCall("WinMail","SendMailSMTP",`	The `SendMailSMTP` function sends the mail with the SMTP protocol.

Function name	Description
[*id*], [*result*])	
CALL ui.Interface.frontCall("WinMail","SendMailMAPI", [*id*], [*result*])	The `SendMailMAPI` function sends the mail with the MAPI protocol.
CALL ui.Interface.frontCall("WinMail","GetError", [*id*], [*result*])	The `GetError` function gets a description of the last error that occurred.

The following functions are needed when you use SMTP server connections:

Table 430: WinMail front-end functions: SMTP-specific

Function name	Description
CALL ui.Interface.frontCall("WinMail","SetSmtp", [*id*, *smtp:port*], [*result*])	The `SetSmtp` function sets the SMTP server to be used.
CALL ui.Interface.frontCall("WinMail","SetFrom", [*id*, *name*, *address*], [*result*])	The `SetFrom` function sets sender information.

Init

The `Init` function initializes the module.

Syntax

```
CALL ui.Interface.frontCall("WinMail","Init",
  [], [id ] )
```

- *ret* is the identifier of the message initialized.
- For each Init function, a `Close` must be called.

Usage

This function initializes the module. It returns the identifier for the message, which will be used in other functions.

Close

The `Close` function clears all information corresponding to a message, and frees the memory occupied by the message.

Syntax

```
CALL ui.Interface.frontCall("WinMail","Close",
  [id], [ result ] )
```

- *id* is the message identifier.

- *result* is the status of the function.

SetBody

The `SetBody` function sets the body of the mail.

Syntax

```
CALL ui.Interface.frontCall("WinMail","SetBody",
  [ id, body ], [ result ] )
```

- *id* is the message identifier.
- *body* is the string text containing the body of the mail.
- *result* is the status of the function.

SetSubject

The `SetSubject` function sets the subject of the mail.

Syntax

```
CALL ui.Interface.frontCall("WinMail","SetSubject",
  [ id, subject ], [ result ] )
```

- *id* is the message identifier.
- *subject* is the string text containing the subject of the mail.
- *result* is the status of the function.

AddTo

The `AddTo` function adds a "To" addressee to the mail.

Syntax

```
CALL ui.Interface.frontCall("WinMail","AddTo",
  [ id, name, address ], [ result ] )
```

- *id* is the message identifier.
- *name* is the name to be displayed in the mail.
- *address* is the mail address to be used for this addressee.
- *result* is the status of the function.

Usage

This function adds a "To" Addressee to the mail. The Addressee has a name and a mail address.

AddCC

The `AddCC` function adds a "CC" addressee to the mail.

Syntax

```
CALL ui.Interface.frontCall("WinMail","AddCC",
  [ id, name, address ], [ result ] )
```

- *id* is the message identifier.
- *name* is the name to be displayed in the mail.
- *address* is the mail address to be used for this addressee.
- *result* is the status of the function.

Usage

This function adds a "CC" Addressee to the mail. The Addressee has a name and a mail address.

AddBCC

The `AddBCC` function adds a "BCC" addressee to the mail.

Syntax

```
CALL ui.Interface.frontCall("WinMail","AddBCC",
  [ id, name, address ], [ result ] )
```

- *id* is the message identifier.
- *name* is the name to be displayed in the mail.
- *address* is the mail address to be used for this addressee.
- *result* is the status of the function.

Usage

This function adds a "BCC" Addressee to the mail. The Addressee has a name and a mail address.

AddAttachment

The `AddAttachment` function adds a file as an attachment to the mail.

Syntax

```
CALL ui.Interface.frontCall("WinMail","AddAttachment",
  [ id, fileName], [ result ] )
```

- *id* is the message identifier.
- *fileName* is the path of the attachment; the path can be relative to the directory from which GDC is run, or absolute.
- *result* is the status of the function.

Usage

This function adds a file as an attachment to the mail. The file must be located on the front-end.

SendMailSMTP

The `SendMailSMTP` function sends the mail with the SMTP protocol.

Syntax

```
CALL ui.Interface.frontCall("WinMail","SendMailSMTP",
  [ id ], [result] )
```

- *id* is the message identifier.
- *result* is TRUE in case of success; use `GetError` to get the description of the error when needed.

Usage

This function sends the mail by using the SMTP protocol. default mailer software is called to create the mail. The user must press the "send" button to send the mail.

SendMailMAPI

The `SendMailMAPI` function sends the mail with the MAPI protocol.

Syntax

```
CALL ui.Interface.frontCall("WinMail","SendMailMAPI",
  [ id ], [result] )
```

- *id* is the message identifier.
- *result* is TRUE in case of success; use `GetError` to get the description of the error when needed.

Important:

- MAPI needs to log in to the mailer software. The first login could take time, depending on the mailer software. Your Genero application will be blocked until MAPI returns.
- MAPI depends on the mailer software for error management. For instance, Mozilla Thunderbird returns "success" when the mail is created, but Outlook 2002 only returns "success" when the mail is sent.

Usage

This function sends the mail by using the MAPI protocol. With MAPI, the default mailer software is called to create the mail. The user must press the "send" button to send the mail.

GetError

The `GetError` function gets a description of the last error that occurred.

Syntax

```
CALL ui.Interface.frontCall("WinMail","GetError",
  [ id ], [ result ] )
```

- *id* is the message identifier.
- *result* is the description of the last error.
- the returned value is NULL if there was no error.

SetSmtp

The `SetSmtp` function sets the SMTP server to be used.

Syntax

```
CALL ui.Interface.frontCall("WinMail","SetSmtp",
 [ id, smtp:port ], [ result ] )
```

- *id* is the message identifier.
- *smtp* is the string text containing the SMTP server to be used.
- *port* is optional. It allows to specify a port for your SMTP server. When not specified, the default port remains 25.
- *result* is the status of the function.

SetFrom

The `SetFrom` function sets sender information.

Syntax

```
CALL ui.Interface.frontCall("WinMail","SetFrom",
  [ id, name, address ], [ result ] )
```

- *id* is the message identifier.

- *name* is the name to be displayed in the mail.
- *address* is the mail address to be used for this addressee.
- *result* is the status of the function.

WinMail examples
Various WinMail examples.

- Mail using MAPI on page 1923
- Mail using SMTP server on page 1923

Mail using MAPI

This topic provides an example of sending mail using MAPI.

```
MAIN
  DEFINE result, id INTEGER
  DEFINE str STRING

-- first, we initialize the module
  CALL ui.Interface.frontCall("WinMail", "Init", [], [id])

-- Set the body of the mail
  CALL ui.interface.frontCall("WinMail", "SetBody",
   [id, "This is a text mail using WinMail API - MAPI"], [result])

-- Set the subject of the mail
  CALL ui.interface.frontCall("WinMail", "SetSubject",
   [id, "test mail - ignore it"], [result])

-- Add an Addressee as "TO"
  CALL ui.Interface.frontCall("WinMail", "AddTo",
   [id, "myBoss", "boss@mycompany.com"], [result])

-- Add another Addresse as "BCC"
  CALL ui.Interface.frontCall("WinMail", "AddBCC",
   [id, "my friend", "friend@mycompany.com"], [result])

-- Add Two attachments
  CALL ui.Interface.frontCall("WinMail", "AddAttachment",
   [id, "c:\\mydocs\report.doc"], [result])
  CALL ui.Interface.frontCall("WinMail", "AddAttachment",
   [id, "c:\\mydocs\demo.png"], [result])

-- Send the mail via the default mailer
  CALL ui.Interface.frontCall("WinMail", "SendMailMAPI", [id], [result])
  IF result == TRUE THEN
    DISPLAY "Message sent successfully"
  ELSE
    CALL ui.Interface.frontCall("WinMail", "GetError", [id], [str])
    DISPLAY str
  END IF

  CALL ui.Interface.frontCall("WinMail", "Close", [id], [result])
END MAIN
```

Mail using SMTP server

This topic provides an example of sending mail using an SMTP server.

```
MAIN
  DEFINE result, id INTEGER
  DEFINE str STRING

-- first, we initialize the module
  CALL ui.Interface.frontCall("WinMail", "Init", [], [id])
```

```
-- Set the body of the mail
  CALL ui.interface.frontCall("WinMail", "SetBody", [id,
   "This is a text mail using WinMail API - MAPI"], [result])

-- Set the subject of the mail
  CALL ui.interface.frontCall("WinMail", "SetSubject", [id,
   "test mail - ignore it"], [result])

-- Set the mail sender
  CALL ui.Interface.frontCall("WinMail", "SetFrom", [id, "mySelf",
   "me@mycompany.com"], [result])

-- Set the SMTP server
  CALL ui.Interface.frontCall("WinMail", "SetSmtp", [id,
 "smtp.mycompany.com"],
   [result])

-- Add an Addressee as "TO"
  CALL ui.Interface.frontCall("WinMail", "AddTo", [id, "myBoss",
   "boss@mycompany.com"], [result])

-- Add another Addressee as "BCC"
  CALL ui.Interface.frontCall("WinMail", "AddBCC", [id, "my friend",
   "friend@mycompany.com"], [result])

-- Add Two attachments
  CALL ui.Interface.frontCall("WinMail", "AddAttachment", [id,
   "c:\\mydocs\report.doc"], [result])
  CALL ui.Interface.frontCall("WinMail", "AddAttachment", [id,
   "c:\\mydocs\demo.png"], [result])

-- Send the mail via smtp
  CALL ui.Interface.frontCall("WinMail", "SendMailSMTP", [id], [result])
  IF result == TRUE THEN
    DISPLAY "Message sent successfully"
  ELSE
    CALL ui.Interface.frontCall("WinMail", "GetError", [id], [str])
    DISPLAY str
  END IF
  CALL ui.Interface.frontCall("WinMail", "Close", [id], [result])
END MAIN
```

Genero Application Server front calls

Front-end functions of the session module allow you to dynamically set and get session variables from within your Genero application.

Table 431: Session module front-end functions

Function name	Description
`ui.Interface.frontCall("session", "getVar", [name], [result])`	Returns the value of a session variable.
`ui.Interface.frontCall("session", "setVar", [name,value], [result])`	Sets a value of a session variable.

setVar
Sets a value of a session variable.

Syntax

```
ui.Interface.frontCall("session", "setVar",
   [name,value], [result])
```

- *name* is the name of the session variable.
- *value* is the value to set to the named session variable.
- *result* returns 1 if successful; 0 otherwise.

Usage

The `setVar` function sets a session variable to the value specified.

Setting a variable to an empty string is equivalent to deleting the variable.

getVar
Returns the value of a session variable.

Syntax

```
ui.Interface.frontCall("session", "getVar",
   [name], [result])
```

- *name* is the name of the session variable.
- *result* is the value of the session variable, or an empty string if the variable does not exist.

Usage

The `getVar` function retrieves the value for a session variable.

Genero Mobile common front calls
This section describes common front calls provided by all mobile front-ends.

This table shows the functions implemented by all mobile front-ends in the "`mobile`" module.

Table 432: Common mobile module front-end functions

Function Name	Description
`ui.Interface.frontCall("mobile", "chooseContact", [], [result])`	Lets the user choose a contact from the mobile device contact list and returns the vCard.
`ui.Interface.frontCall("mobile", "choosePhoto", [], [path])`	Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.
`ui.Interface.frontCall("mobile", "chooseVideo", [], [path])`	Lets the user select a video from the mobile device's video gallery and returns a video identifier.
`ui.Interface.frontCall("mobile", "composeMail", [to, subject, content, cc, bcc, attachments ...],`	Invokes the user's default mail application for a new mail to send.

Function Name	Description
`[result])`	
`ui.Interface.frontCall("mobile", "composeSMS", [ recipients, content ], [ result ] )`	Sends an SMS text to one or more phone numbers.
`ui.Interface.frontCall("mobile", "connectivity", [], [result] )`	Returns the type of network available for the mobile device.
`ui.Interface.frontCall("mobile", "getGeolocation", [], [status, latitude, longitude] )`	Returns the Global Positioning System (GPS) location of a mobile device.
`ui.Interface.frontCall("mobile","getRemoteNotifications", [sender_id], [data] )`	This front call retrieves push notification messages.
`ui.Interface.frontCall("mobile", "importContact", [vcard], [result] )`	Creates a new, or merges to an existing entry, the contact details passed in VCard string.
`ui.Interface.frontCall("mobile","registerForRemoteNotifications", [sender_id], [registration_token] )`	This front call registers a mobile device for push notifications.
`ui.Interface.frontCall("mobile", "runOnServer", [ appurl, timeout ], [] )`	Run an application from the Genero Application Server according to the specified URL.
`ui.Interface.frontCall("mobile", "scanBarCode", [], [code, type] )`	Allow the user to scan a barcode with a mobile device
`ui.Interface.frontCall("mobile", "takePhoto", [], [path] )`	Lets the user take a picture with the mobile device and returns the corresponding picture identifier.
`ui.Interface.frontCall("mobile", "takeVideo", [], [path])`	Lets the user take a video with the mobile device and returns the corresponding video identifier.
`ui.Interface.frontCall("mobile","unregisterFromRemoteNotifications", [ sender_id ], [] )`	This front call unregisters the mobile device from push notifications.

chooseContact
Lets the user choose a contact from the mobile device contact list and returns the vCard.

Syntax

```
ui.Interface.frontCall("mobile", "chooseContact",
   [], [result])
```

- *result* - The vCard string from the device's contacts database.

Usage

The "`chooseContact`" front call opens the mobile device contact chooser, lets the user select a contact and returns the contact as a vCard string.

If the user cancels the contact chooser, `NULL` is returned.

Example

```
DEFINE vcard STRING
CALL ui.Interface.frontCall("mobile", "chooseContact", [],
 [vcard] )
```

choosePhoto
Lets the user select a picture from the mobile device's photo gallery and returns a picture identifier.

Syntax

```
ui.Interface.frontCall("mobile", "choosePhoto",
   [], [path])
```

1. *path* - Holds the device opaque path to the chosen photo.

Usage

The "`choosePhoto`" front call starts the system's photo chooser (the device's photo gallery), allows the user to choose a photo, and returns the path/URL on the mobile device of the chosen photo.

If the user cancels the photo chooser, `NULL` is returned.

The value returned in the *path* variable contains a reference to the system location of the picture on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the picture.

For more details about mobile image handling, see images handling on mobile devices.

chooseVideo
Lets the user select a video from the mobile device's video gallery and returns a video identifier.

Syntax

```
ui.Interface.frontCall("mobile", "chooseVideo",
  [], [path])
```

1. *path* - Holds the device opaque path to the selected video.

Usage

The "`chooseVideo`" front call starts the system's video chooser (the device's video gallery), allows the user to choose a video, and returns the path/URL on the mobile device of the selected video.

If the user cancels the video chooser, `NULL` is returned.

The value returned in the *path* variable contains a reference to the system location of the video on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the video.

Once the video identifier/path is known, it is possible to fetch the video file from the device to the program context with the `fgl_getfile()` API. The procedure is similar to fetching photos from the device. For more details, see the section about video handling on mobile devices.

To play the video, you can perform a `"launchURL"` front call, with the opaque path returned by this front call.

composeMail
Invokes the user's default mail application for a new mail to send.

Syntax

```
ui.Interface.frontCall("mobile", "composeMail",
   [to, subject, content, cc, bcc, attachments ...],
   [result])
```

- *to* - A list of recipients, separated by commas. While the list uses commas to separate the recipients in the list, the list itself is enclosed in a single set of quotes.
- *subject* - The subject of the email.
- *content* - The body of the email.
- *cc* - (optional) A list of recipients for the carbon-copy email field, separated by commas. While the list uses commas to separate the recipients in the list, the list itself is enclosed in a single set of quotes.
- *bcc* - (optional) A list of recipients for the blind carbon-copy email field, separated by commas. While the list uses commas to separate the recipients in the list, the list itself is enclosed in a single set of quotes.
- *attachments ...* - (optional) All remaining arguments are treated as paths to attachment files. Each attachment file name is enclosed in its own set of quotes. The comma is used to separate the attachments in the list.
- *result* - Holds a status message.

Usage

The "`composeMail`" front call invokes the user's default mail application and sets up a new mail to send.

The returned result string can take one of the following values:

- "`ok`": The email was send.
- "`cancel`": The email was canceled.
- "`saved`": The email was saved.
- "`failed: reason`": The email could not be sent.

This example opens an email and populates the To, CC, and BCC fields, the Subject line, the message body, and it specifies two attachments..

```
DEFINE result STRING
CALL ui.Interface.frontCall("mobile", "composeMail",
 ["john.doe@4js.com,jane.doe@4js.com", "Hello world",
  "This is the hello world text", "john.doe@4js.com,jane.doe@4js.com",
 "hidden@4js.com",
```

```
   "/sdcard/Pictures/photo1.jpg", "/sdcard/Pictures/photo2.jpg" ], [result])
```

The next example opens an email and populates the To field, the Subject line, and the message body. No CC or BCC recipients and no attachments are specified.

```
DEFINE result STRING
CALL ui.Interface.frontCall("mobile","composeMail",
 ["huhu@haha.com","test mail","sent from my device"],[result])
```

composeSMS
Sends an SMS text to one or more phone numbers.

Syntax

```
ui.Interface.frontCall("mobile", "composeSMS",
  [ recipients, content ],
  [ result ] )
```

- *recipients* - A list of phone numbers, separated by commas. While the list uses commas to separate the phone numbers in the list, the list itself is enclosed in a single set of quotes.
- *content* - The SMS message.
- *result* - Holds a status message.

Usage

The "composeSMS" front call sends an SMS text to one or more phone numbers.

Consider using global phone numbers with a + plus sign, as described in [RFC3966].

The returned result string can take one of the following values:

- "ok": The SMS was send.
- "cancel": The SMS was canceled.
- "failed": The SMS could not be sent.

Error -6333 is raised, if there is no permission to compose an SMS on the mobile phone.

Example

```
DEFINE result STRING
CALL ui.Interface.frontCall("mobile", "composeSMS",
     ["+332781211,+339956789", "This is the SMS text"],
     [result])
```

connectivity
Returns the type of network available for the mobile device.

Syntax

```
ui.Interface.frontCall("mobile", "connectivity",
   [], [result] )
```

- *result* - Holds the type of network available.

Usage

The "connectivity" front call checks for the best available mobile network connectivity to the internet.

The returned result string can take one of the following values:

- "NONE": No connectivity is available to the internet or the specified host.
- "MobileNetwork": Connectivity is available via the mobile network (Edge, 3G, 4G).
- "WIFI": Connectivity is available via a WIFI connection.

Example

```
DEFINE network STRING
CALL ui.Interface.frontCall("mobile", "connectivity", [],
 [network] )
IF network == "WIFI" THEN
   ...
END IF
```

getGeolocation

Returns the Global Positioning System (GPS) location of a mobile device.

Syntax

```
ui.Interface.frontCall("mobile", "getGeolocation",
   [], [status, latitude, longitude] )
```

1. *status* - Holds the status of the front call execution.
2. *latitude* - Holds the current latitude.
3. *longitude* - Holds the current longitude.

Usage

The "getGeolocation" front call returns the current location of the mobile device, based on the current GPS information.

The possible values returned in the *status* parameter are:

- "ok": The mobile device location could be found.
- In case of failre, the status variable contains the error description, for example, "location services not enabled".

The returned coordinates should be stored in FLOAT variables.

If the device location cannot be found within a given period, the front call returns an error status.

Example

```
DEFINE status STRING, latitude, longitude FLOAT
CALL ui.Interface.frontCall("mobile", "getGeolocation",
     [], [status, latitude, longitude] )
MESSAGE SFMT(
  "Geo location: (status=%1) Latitude=%2 Longitude=%3",
  status, latitude, longitude )
```

getRemoteNotifications

This front call retrieves push notification messages.

Syntax

```
ui.Interface.frontCall("mobile","getRemoteNotifications",
   [sender_id], [data] )
```

1. *sender_id* - For GMA, the sender_id identifies the mobile device. It's obtained when you create a GCM project. This parameter is ignored by GMI.
2. *data* - `STRING` containing a JSON array of notifications.

Usage

After registering for push notifications with the registerForRemoteNotifications on page 1934 front call, the `getRemoteNotifications` front call can be called in the context of an `ON ACTION notificationpushed` action handler.

The GMI or GMA front-end will send the `notificationpushed` special action, when it receives notifications from the push notification server. When this action is fired, use the `getRemoteNotifications` front call to get notification data. On GMA, identify the GCM client by passing the *sender_id* obtained from the GCM project as a parameter. On GMI, the `sender_id` can be `NULL`, as it is ignored.

> **Important:**
>
> When an app restarts, if notifications are pending and the app has already registered for push notification in a previous execution, the `notificationpushed` action will be raised as soon as a dialog with the corresponding `ON ACTION` handler activates. The app should then perform a getRemoteNotifications on page 1930 front call as in the regular case, to get the pending notifications pushed to the device while the app was off.
>
> However, special consideration needs to be given to iOS devices. When push notification arrives for an iOS app that has not started, there is no mechanism to wake up the app and get the push data. Therefore, when the user starts the app from the springboard, there will never have any push data available. Depending on the context, implement the following programming patterns to solve this problem:
>
> 1. If the push notification contains a badge number, the app can verify if the badge is greater than 0 (with the `getBadgeNumber` front call) in order to perform a `getRemoteNotifications` front call. Even if there is no data available with the front call, the app should directly ask the server push provider to get last push data.
> 2. If the push notification does not contain badge numbers, the app should always perform a `getRemoteNotification` front call when it starts. If there is no push data available from the front call, the app should ask the server push provider if there is push data available. This is by the way also recommended when receiving a `notificationpushed` action during application life time.
> 3. If the user starts the app from the Notification Center, the app is launched with push data transmitted from the system, and the `notificationpushed` action is sent. The app should the perform, the `getRemoteNotifications` front call and get the push data.

The "`getRemoteNotifications`" front call returns a list of notification records as a JSON array string. Use the util.JSONArray or util.JSON class to extract notification data from the returned string. The structure of a push notification is platform specific. See below for details.

> **Important:** When an iOS app is in background, silent push notifications can occur, but notification message data (i.e. the payload) may not be available. In such case, GMI is able to detect that a notification arrived (i.e. when the app badge number is greater than zero) and raise the `notificationpushed` action, but the `getRemoteNotifications` front call will return no message data (*data* return param is `NULL`). If such case, implement a fallback mechanism (based on RESTFul web services for example), to contact the push notification provider and retrieve the message information.

Push notification records with GMA / Android™

The returned JSON string from a GCM notification server contains an array of notification records.

A notification record contains the following JSON keys:

- `"type"` - can be `"message"` or `"token"`.
- `"data"` - Contains notification data.
  - When `"type":"message"`, the notification record is a GCM application message, and the `data` attribute contains custom notification information.

    An element of "data" can be a `"genero_notification"` record, that will produce an Android graphical notification. This record must define the following attributes:

    - `"title"` - title of the graphical notification
    - `"content"` - text content of the graphical notification
    - `"icon"` - icon of the graphical notification

    The `"genero_notification"` record can be followed by custom notification data.
  - When `"type":"token""`, the notification record is a registration token update, and the `"data"` attribute contains the new registration token, which should be re-sent to the push notification server.
- `"from"` - Contains the GCM project id.

JSON push notification data example for GMA:

```
[
 {
  "type": "message",
  "data": { custom-attributes ... },
  "from": "project-id"
 },
 {
  "type": "token",
  "data": "new-registration-token",
  "from": "project-id"
 },
 ...
]
```

Note that the JSON push notification data can contain a `"data"` attribute with a `"genero_notification"` record, that will produce an Android graphical notification:

```
[
 {
  "type": "message",
  "data": {
            "genero_notification" :
            {
             "title"   : "Game Request!",
             "content" : "Bob wants to play poker...",
             "icon"    : "smiley"
            },
            custom-attributes
            ...
          },
  "from": "project-id"
 },
 ...
]
```

Push notification records with GMI / iOS

The returned JSON string from an Apple Push Notification contains an array of notification records.

A push notification record contains the following JSON attributes:

- `"aps"` (required) - key to be recognized by devices as an Apple Push Notification

- "alert" (required) - key of the push notification content. If not specified as a single value, the alert key can hold:
  - "title" - title of the alert.
  - "body" - the message to be displayed.
- "badge" (optional) - the number to display as the badge of the app icon. If this property is absent, the badge is not changed. You need to manage it through your push notification provider.
- "sound" (optional) - the sound played by the alert (aiff, wav, or caf format). default value : "default". To use a custom file you will need to use the gmi extension project and be familiar with Objective-C. The file must bundled with the app.
- "content-available" (required) - The content-available property with a value of 1 lets the remote notification act as a “silent” notification. Notifications received in background mode should be stored for delivery when the app enters foreground mode.

JSON push notification data example for GMI:

```
[
 {
  "aps" :
  {
    "alert" : "My first push",
    "badge" : 1,
    "sound" : "default",
    "content-available" : 1
  }
 },
 {
  "aps" :
  {
    "alert" :
    {
      "title" : "Push",
      "body" : "My second push"
    }
    "badge" : 2,
    "sound" : "default",
    "content-available" : 1
  },
  "new_ids" : [ "XV234", "ZF452", "RT563" ],
  "updated_ids" : [ "AC634", "HJ153" ]
 }
]
```

In the last record, custom information is provided in the "new_ids" and "updated_ids" attributes, as a JSON array of identifiers.

For more details, see Apple Push Notification Service.

Example

```
IMPORT util -- JSON API
CONSTANT GCM_SENDER_ID = "<enter your GCM Sender ID (NULL for
 APNs)>"
...
DEFINE notif_list STRING,
       sender_id STRING

LET sender_id = GCM_SENDER_ID

DIALOG ...
  ...
```

```
  ON ACTION notificationpushed

     CALL ui.Interface.frontCall(
             "mobile", "getRemoteNotifications",
             [ sender_id ], [ notif_list ] )

     -- Analyse content of notiflist
     DISPLAY util.JSON.format(notif_list)
  ...
```

importContact
Creates a new, or merges to an existing entry, the contact details passed in vCard string.

Syntax

```
ui.Interface.frontCall("mobile", "importContact",
   [vcard], [result] )
```

1. *vcard* - Holds a vCard string to be imported into the device's contacts database.
2. *result* - Holds the completed vCard string.

Usage

The "`importContact`" front call sends the vCard definition passed as parameter to the mobile device.

If the contact import is canceled, the front-end returns `NULL`. Otherwise, it returns the vCard data.

On iOS devices, the user has the choice to create a new contact, or complete an existing contact entry. When creating a new entry, the contact input form is opened on the mobile device, to let the user complete the default values passed as parameter. When merging contact information to an existing entry, the user selects an entry from the contact list. If the contact import is validated, the front call returns the completed vCard string.

On Android™ devices, this front call creates a new contact entry directly in the mobile contact list, according to the VCard definition passed as parameter, no intermediate input form is proposed to the end user. If the contact import is validated, the front call returns the original vCard string passed as parameter.

Example

```
DEFINE vcard, result STRING
LET vcard="BEGIN:VCARD\n"
      ||"VERSION:3.0\n"
      ||"N:Willi;;;;\n"
      ||"TEL;type=HOME;type=VOICE;type=pref:03812225610\n"
      ||"END:VCARD\n"
CALL ui.interface.frontcall("mobile","importContact",[vcard],
[result])
```

registerForRemoteNotifications
This front call registers a mobile device for push notifications.

Syntax

```
ui.Interface.frontCall("mobile","registerForRemoteNotifications",
   [sender_id], [registration_token] )
```

1. *sender_id* - For GMA, the sender_id identifies the mobile device. It's obtained when you create a GCM project. This parameter is ignored by GMI.

2. *registration_token* - Registration token to be sent to the push notification provider. For GMA/Android, this is the "registration token" obtained from GCM, for GMI/iOS, this is the "device token" obtained from APNs.

Usage

The "`registerForRemoteNotifications`" front call registers the mobile device for push notifications. Once the registration procedure is done (see below for platform specifics), it is possible to get notification events through the `notificationpushed` predefined action, and retrieve notification data with the getRemoteNotifications on page 1930 front call.

> **Note:** The app does not need to register for notification each time it is restarted: Even if the app is closed, the registration is still active until the `unregisterFromRemoteNotifications` front call is performed. At first execution, an app will typically ask if the user wants to get push notifications and register to the push service if needed. To disable push notification, apps usually implement an option that can be disabled (to unregister) and re-enabled (to register again) by the user. On Android, that the app must register for notification each time it is upgraded.

On Android when using GCM, you get the *sender_id* and an API key when you create a GCM project (see https://developers.google.com/cloud-messaging/android/client#get-config). The *registration_token* is the registration token returned by GCM. Once registered with the GCM service, the app must also send this registration token to the GCM application server. Registration tokens are typically sent to the GCM application server using a RESTFul HTTP POST. For more details, see GCM documentation on the Google developer web site. For more details about GCM registration, see About GCM Connection Server.

> **Note:** Android apps using push notification services need specific permissions to be defined in the manifest, such as `android.permission.GET_ACCOUNTS`, `com.google.android.c2dm.permission.RECEIVE`, and especially *`application-package-name`*`.permission.C2D_MESSAGE`. These Android permissions will be automatically set by the gmabuildtool, according to the package name specified with the `--build-app-package-name` option. For more details, see GCM documentation.

On iOS when using APNs, the *sender_id* is ignored. The *registration_token* is the device token returned by the Apple Push Notification service. Once registered with the Apple Push Notification service, the app must also send this device token to the push notification provider, typically using a RESTFul HTTP POST. For more details about Apple Push Notification Provider, see Provider Communication with Apple Push Notification Service.

Example

The following code example registers with Google Cloud Messaging or Apple Push Notification service. It then sends the registration token to the push notification provider (the get_device_type() function returns the front-end type from the feInfo/feName front call):

```
IMPORT com  -- For RESTful post
IMPORT util -- JSON API

CONSTANT GCM_SENDER_ID = "<enter your GCM Sender ID (NULL for
 APNs)>"
...

DEFINE sender_id STRING,
       registration_token STRING
DEFINE req com.HTTPRequest,
       obj util.JSONObject,
       resp com.HTTPResponse

-- First get the registration token
LET sender_id = GCM_SENDER_ID
CALL ui.Interface.frontCall(
```

```
        "mobile", "registerForRemoteNotifications",
        [ sender_id ], [ registration_token ] )

-- Then send registration token to push notification provider
TRY
    LET req = com.HTTPRequest.create("http://SERVER_IP:4930")
    CALL req.setHeader("Content-Type", "application/json")
    CALL req.setMethod("POST")
    CALL req.setTimeOut(5)
    LET obj = util.JSONObject.create()
    CALL obj.put("registration_token", registration_token)
    CALL req.doTextRequest(obj.toString())
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
       MESSAGE SFMT("HTTP Error (%1) %2",
                    resp.getStatusCode(),
                    resp.getStatusDescription())
    ELSE
       MESSAGE "Registration token sent."
    END IF
CATCH
    MESSAGE SFMT("Could not post registration token to server:
 %1", STATUS)
END TRY
...
```

runOnServer
Run an application from the Genero Application Server according to the specified URL.

Syntax

```
ui.Interface.frontCall("mobile", "runOnServer",
   [ appurl, timeout ], [] )
```

- *appurl* - The GAS URL to the Genero application (this must be a `ua/r` URL).
- *timeout* - The timeout (in seconds) to wait for the remote application.

Usage

The `runOnServer` front call allows you to start an application in the Genero Application Server (GAS), from an embedded/local application running on the mobile device. The remote application's graphical user interface displays on the mobile device.

The front call returns when the called application ends, and the control goes back to the initial application executing on the mobile device.

The applications executed on the GAS server must use the UTF-8 encoding. Mobile front-ends will reject any attempt to display forms of an application using an encoding other than UTF-8.

The remote application cannot use `RUN WITHOUT WAITING` to start child programs. Only `RUN` is supported.

The first parameter (*appurl*) identifies the remote application to be started and must contain an "`ua/r`" URL syntax (the UA protocol introduced with the GAS 3.00).

For example: `http://myappserver:6394/ua/r/myapp`.

This URL may contain a query string, with parameters for the application to be executed by the GAS.

The *timeout* parameter is optional. It can be used to give the control back to the local app, if the remote app takes too long to respond. If not specified, or when zero is passed, the timeout is infinite.

In case of failure (such as application not found, or timeout expired), the front call raises the runtime error -6333 and the HTTP status code of the request can be found in the error message details.

> **Note:** The application running on the GAS can only access the *data-directory* directory, in the sandbox of the embedded application that executes the `runOnServer` front call. File handling APIs like `fgl_getfile()` and `fgl_putfile()` can only access this directory on the mobile device. If no absolute path is specified in the file path for the mobile device, the *`data-directory`* is used.

Example

```
TRY
    CALL ui.interface.frontcall("mobile","runOnServer",["http://
santana:6394/ua/r/orders"],[])
CATCH
    ERROR err_get(STATUS)
END TRY
```

scanBarCode
Allow the user to scan a barcode with a mobile device

Syntax

```
ui.Interface.frontCall("mobile", "scanBarCode",
   [], [code, type] )
```

1. *code* - Holds a string representation of the barcode.
2. *type* - Holds the name of the barcode type.

Usage

The "`scanBarCode`" front call starts the barcode scanner to let the user scan a barcode with the device.

After reading the barcode, the front call returns the string representation of the barcode and the barcode type (i.e. symbology).

The *code* return parameter contains the barcode string.

The *type* return parameter indicates the type of barcode that was scanned.

If the barcode scan was canceled, the *code* return parameter is set to `NULL` and *type* is set to "canceled".

- On iOS devices, the barcode reader used by GMI is "ZBar". For more details, see http://zbar.sourceforge.net
- On Android™ devides, the barcode reader used by GMA is "zxing". The zxing barcode reader must be installed as a separate app. For more details, see https://play.google.com/store/apps/details?id=com.google.zxing.client.android.

Table 433: Barcode type codes returned by GMI and GMA

Barcode type name (GMI/iOS)	Barcode type name (GMA/ Android)	Description
N/A	AZTEC	Aztec barcode format
N/A	CODEBAR	CODABAR format
CODE-39	CODE_39	AKA Alpha39, Code 3 of 9 or USD-3 format
CODE-93	CODE_93	Intermec (Canada Post) format

Barcode type name (GMI/iOS)	Barcode type name (GMA/ Android)	Description
CODE-128	CODE_128	High-density barcode (128 chars) format
N/A	DATA_MATRIX	Data Matrix format
EAN-8	EAN_8	European/International Article Number (8 digits) format
EAN-13	EAN_13	European/International Article Number (13 digits) format
I2/5	ITF	Interleaved 2 of 5 format
ISBN-10	N/A	International Standard Book Number (10 digits) format
ISBN-13	N/A	International Standard Book Number (13 digits) format
N/A	MAXICODE	ISO/IEC 16023 format
PDF417	PDF_417	Portable Data File - 417 format
QR-Code	QR_CODE	Quick Response Code format
N/A	RSS_14	GS1 DataBar (Reduce Space Symbology) format
N/A	RSS_EXPANDED	GS1 DataBar Expanded (Reduce Space Symbology expanded) format
UPC-A	UPC_A	Universal Product Code (12 digits) format
UPC-E	UPC_E	Universal Product Code (6 digits) format
N/A	UPC_EAN_EXTENSION	UPC/EAN extension format

takePhoto
Lets the user take a picture with the mobile device and returns the corresponding picture identifier.

Syntax

```
ui.Interface.frontCall("mobile", "takePhoto",
   [], [path] )
```

1. *path* - Holds the device opaque path to the picture that has been taken.

Usage

The "takePhoto" front call invokes the mobile device's camera to let the user take a picture and returns the local path/URL on the mobile device to the picture.

If the photo is canceled by the user, the front call returns NULL.

The value returned in the *path* variable contains a reference to the system location of the picture on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the picture.

For more details about mobile image handling, see images handling on mobile devices.

takeVideo
Lets the user take a video with the mobile device and returns the corresponding video identifier.

Syntax

```
ui.Interface.frontCall("mobile", "takeVideo",
  [], [path])
```

1. *path* - Holds the device opaque path to the video.

Usage

The "`takeVideo`" front call invokes the mobile device's camera to let the user take a video and returns the local path/URL to the video on the mobile device.

If the photo is canceled by the user, the front call returns `NULL`.

The value returned in the *path* variable contains a reference to the system location of the video on the mobile device. This path is platform dependent, and may change in future versions. Consider the path returned by this front call as an opaque local file identifier, and do not use this path as a persistent file name for the video.

Once the video identifier/path is known, it is possible to fetch the video file from the device to the program context with the `fgl_getfile()` API. The procedure is similar to fetching photos from the device. For more details, see the section about video handling on mobile devices.

To play the video, you can perform a `"launchURL"` front call, with the opaque path returned by this front call.

unregisterFromRemoteNotifications
This front call unregisters the mobile device from push notifications.

Syntax

```
ui.Interface.frontCall("mobile","unregisterFromRemoteNotifications",
   [ sender_id ], [] )
```

1. *sender_id* - For GMA, the sender_id identifies the mobile device. It's obtained when you create a GCM project. This parameter is ignored by GMI.

Usage

The "`unregisterFromRemoteNotifications`" front call unregisters the device from push notifications after it has been registered with the registerForRemoteNotifications on page 1934 front call.

On Android with GCM, to unregister the mobile device from GCM push notifications, pass the *sender_id* used to identify the GCM client. You obtain the *sender_id* when you create the GCM project.

On iOS with APNs, provide a `NULL` as `sender_id`, to unregister the iOS mobile device from push notifications.

Example

```
DEFINE sender_id STRING
...
IF get_device_type() == "GMA" THEN
   LET sender_id = "94019931415" -- Got from GCM project
 creation
ELSE
   LET sender_id = NULL -- Ignored by GMI
END IF
```

```
CALL ui.Interface.frontCall(
        "mobile", "unregisterFromRemoteNotifications",
        [ sender_id ], [ ] )
...
```

Genero Mobile Android™ front calls
This section describes front calls specific to the Android platform.

This table shows the functions implemented by the Android front-end in the "`android`" module.

Table 434: Android module front-end functions

Function Name	Description
`ui.Interface.frontCall("android", "askForPermission", [permission], [result])`	Ask the user to enable a dangerous feature on the Android device.
`ui.Interface.frontCall("android", "showAbout", [],[])`	Shows the GMA about box displaying version information.
`ui.Interface.frontCall("android", "showSettings", [], [])`	Shows the GMA settings box controlling debug options.
`ui.Interface.frontCall("android", "startActivity", [action, data, category, type, component, extras], [])`	Starts an external Android application (activity), and returns to the GMA application immediately.
`ui.Interface.frontCall("android", "startActivityForResult", [action, data, category, type, component, extras], [outdata, outextras])`	Starts an external application (Android activity) and waits until the activity is closed.

askForPermission (Android™)
Ask the user to enable a dangerous feature on the Android device.

Syntax

```
ui.Interface.frontCall("android","askForPermission",
  [permission], [result])
```

1. *permission* - Identifies the Android permission to enable.
2. *result* - Holds the execution status of the front call:
   - "`ok`" : the user accepted the permission.
   - "`rejected`" : the user refused the permission.

Usage

The "`askForPermission`" front call opens a message box to let the end user confirm the access to a "dangerous" Android permission, in order to enable a risky feature of the mobile device for the current app.

> **Important:** Starting with Android 6, permissions to access dangerous mobile functions are no longer asked during app installation: The app must ask the user for dangerous permissions when needed, by using the `askForPermission` front call.

The *permissions* parameter defines the Android permission to be asked. It must be a string representing one of the Android permission contants, as defined in Android's Manifest permissions, prefixed by the `"android.permission."` string. For example, the `"android.permission.WRITE_EXTERNAL_STORAGE"` string can be used to identify the permission to access the SDCARD storage unit.

> **Important:** Specific Android permissions required by the app still need to be specified when building the app. However, it is not needed to specifiy Android permissions required for built-in front calls: For example, if the app code makes a `choosePhoto` front call, the GMA will implicitely ask the user and set the Android permission to access the photo gallery. For more details, see Android permissions on page 2577.

The front call will raise a runtime exception if the permission identifier is not valid.

Example

The following code example asks the user to access the SDCARD, and handles the user choice:

```
DEFINE result STRING
CALL ui.Interface.frontCall(
        "android", "askForPermission",
        ["android.permission.WRITE_EXTERNAL_STORAGE"],
        [result] )
CASE result
   WHEN "ok"
      CALL os.Path.mkDir("/sdcard/myfiles")
   WHEN "rejected"
      ERROR "SDCARD access was denied by user"
END CASE
```

showAbout (Android™)
Shows the GMA about box displaying version information.

Syntax

```
ui.Interface.frontCall("android", "showAbout",
  [],[])
```

Usage

This front call simply shows a typical about box, indicating GMA version information.

> **Important:** This front call is only available for an application running on an Android device.

No input parameters are required, and no parameters are returned.

showSettings (Android™)
Shows the GMA settings box controlling debug options.

Syntax

```
ui.Interface.frontCall("android", "showSettings", [], [])
```

Usage

This front call opens the settings box to enable or disable GMA programming options.

> **Important:** This front call is only available for an application running on an Android device.

No input parameters are required, and no parameters are returned.

The following features can be controlled with the GMA settings box:

- HTTP debug server on port 6480 (to inspect the AUI tree and show app logs)
- GUI display (FGLSERVER on page 185) and remote debug with `fgldb` on port 6400
- Android logcat recording
- Managing allowed certificates (SSH connections)
- Cookies cleanup (for SSO authentication tokens)

startActivity (Android™)
Starts an external Android application (activity), and returns to the GMA application immediately.

Syntax

```
ui.Interface.frontCall("android","startActivity",
  [action, data, category, type, component, extras],
  [])
```

1. *action* - Identifies the activity to be started on the Android device.
2. *data* - (optional) The data to operate on in the activity (URL, etc).
3. *category* - (optional) A comma separated list of categories.
4. *type* - (optional) Specifies the type of the data passed to the activity.
5. *component* - (optional) Specifies a component class to use for the intent.
6. *extras* - (optional) This is a JSON string containing parameters to pass to the activity.

Usage

The "`startActivity`" front call starts an external application (Android activity), and returns to the GMA application immediately after invoking the activity.

> **Important:** This front call is only available for an application running on an Android device.

This front call is similar to the `RUN WITHOUT WAITING` statement: It allows the user to switch between the GMA and the launched application.

The parameters passed to this front call are used to build an Android "intent" object to start an "activity". For more details about Android intent object, refer to the Android "Intent" defintion.

The *action* parameter defines the Android activity to perform, such as `"android.intent.action.MAIN"`, `"android.intent.action.VIEW"`, and so on.

The *data* (optional) parameter contains the data to operate on. This is the main parameter to transmit data to the activity. It can for example be an URL.

The *category* (optional) parameter contains a comma separated list of categories, where a category gives additional information about the action to execute. For example, `"android.intent.category.LAUNCHER"` means it should appear in the Launcher as a top-level application. See the Android documentation for details about possible categories for a given activity.

The *type* (optional) parameter defines the type (in fact, a MIME type) of the activity data. Normally the type is inferred from the data itself. By setting this attribute, you disable that evaluation and force an explicit type.

The *component* (optional) parameter defines the name of a component class to use for the intent. Normally this is determined by looking at the other information in the intent. The component name typically specified as "*apk-package-name/java-class-name*" or "*java-class-name*" (the APK package name is optional). If the APK package is not specified, GMA considers that the Java class is included in the current APK.

The *extras* (optional) parameter specifies a JSON string containing parameters to pass to the activity. This can be used to provide extended information to the component. For example, with an action sending an e-mail message, the extra data can include data to supply a subject, body, for the e-mail.

Example

The following code example starts the VIEW Android activity to show an image. The Genero program flow will continue after this call, but the started activity will be shown. Note that such action is rather performed with a `launchurl` front call.

```
CALL ui.Interface.frontCall(
        "android", "startActivity",
        [ "android.intent.action.VIEW",
          "file:///storage/path_to_image_file",
          NULL, "image/*" ],
        [ ] )
```

startActivityForResult (Android™)

Starts an external application (Android activity) and waits until the activity is closed.

Syntax

```
ui.Interface.frontCall("android", "startActivityForResult",
  [action, data, category, type, component, extras],
  [outdata, outextras])
```

1. *action* - Identifies the activity to be started on the Android device.
2. *data* - (optional) The data to operate on in the activity (URL, etc).
3. *category* - (optional) A comma separated list of categories.
4. *type* - (optional) Specifies the type of the data passed to the activity.
5. *component* - (optional) Specifies a component class to use for the intent.
6. *extras* - (optional) This is a JSON string containing parameters to pass to the activity.

Return values include:

1. *outdata* - holds the flat value returned by the invoked activity.
2. *outextras* - holds the JSON data of structured value returned by the invoked activity.

The return values depend entirely on the invoked activity.

Usage

The "`startActivityForResult`" front call starts an external application (Android activity), then waits for the user to exit the external application prior to returning the the GMA application.

> **Important:** This front call is only available for an application running on an Android device.

This front call is similar to the `RUN` statement: The user cannot return to the GMA application while the activity is executing.

The parameters passed to this front call are used to build an Android "intent" object to start an "activity". For more details about Android intent object, refer to the Android "Intent" defintion.

The *action* parameter defines the Android activity to perform, such as `"android.intent.action.MAIN"`, `"android.intent.action.VIEW"`, and so on.

The *data* (optional) parameter contains the data to operate on. This is the main parameter to transmit data to the activity. It can for example be an URL.

The *category* (optional) parameter contains a comma separated list of categories, where a category gives additional information about the action to execute. For example, `"android.intent.category.LAUNCHER"` means it should appear in the Launcher as a top-level application. See the Android documentation for details about possible categories for a given activity.

The *type* (optional) parameter defines the type (in fact, a MIME type) of the activity data. Normally the type is inferred from the data itself. By setting this attribute, you disable that evaluation and force an explicit type.

The *component* (optional) parameter defines the name of a component class to use for the intent. Normally this is determined by looking at the other information in the intent. The component name typically specified as "*apk-package-name/java-class-name*" or "*java-class-name*" (the APK package name is optional). If the APK package is not specified, GMA considers that the Java class is included in the current APK.

The *extras* (optional) parameter specifies a JSON string containing parameters to pass to the activity. This can be used to provide extended information to the component. For example, with an action sending an e-mail message, the extra data can include data to supply a subject, body, for the e-mail.

The *outdata* returning argument will contain the flag value returned from the activity, typically when the data is simple and not structured.

The *outextras* returning argument can hold JSON data of any structured value returned by the invoked activity, or `NULL` in case of error (for example, when the application corresponding to the activity is not installed)

Example

This example invokes the barcode scanner application, and returns the scanned barcode.

```
IMPORT util
...
DEFINE data, extras STRING,
       json_object util.JSONObject,
       scanned_value STRING
...
CALL ui.Interface.frontCall(
        "android", "startActivityForResult",
        [ "com.google.zxing.client.android.SCAN",
          NULL, "android.intent.category.DEFAULT" ],
        [ data, extras ])
IF extras IS NULL THEN
  -- If the application isn't installed invoke
  -- the Play Store to give the user a chance to install it
  CALL ui.Interface.frontCall("standard", "launchurl",
          ["market://details?
id=com.google.zxing.client.android"], [])
ELSE
  LET json_object = util.JSONObject.parse(extras)
  -- Fetch the scanned value
  LET scanned_value = json_object.get("SCAN_RESULT")
END IF
```

Genero Mobile iOS front calls

This section describes front calls specific to the iOS platform.

This table shows the functions implemented by the iOS front-end in the "`ios`" module.

Table 435: iOS module front-end functions

Function Name	Description
`ui.Interface.frontCall("ios", "getBadgeNumber", [],[value])`	Returns the current badge number associated to the app.
`ui.Interface.frontCall("ios", "newContact", [defaults],[vcard])`	Lets the user input contact information to create a new entry in the contact database of the mobile device.
`ui.Interface.frontCall("ios", "setBadgeNumber", [value], [])`	Sets the current badge number associated to the app.

getBadgeNumber (iOS)

Returns the current badge number associated to the app.

Syntax

```
ui.Interface.frontCall("ios", "getBadgeNumber",
  [],[value])
```

- *value* - Holds the current badge number.

Usage

The iOS "`getBadgeNumber`" front call returns the current badge number associated to the app.

> **Important:** This front call is only available for an application running on an iOS device.

The badge number appears on the app icon and is typically used for Push notifications on page 2599.

> **Important:** In order to query or set the badge number, the app program must have executed a `registerForRemoteNotifications` front call before (in the current or prior execution instance). This registration is required in order to set the appropriate app permissions to access badge number data.

Example

```
DEFINE value INTEGER
CALL ui.interface.frontcall("ios","getBadgeNumber",[],[value])
```

newContact (iOS)

Lets the user input contact information to create a new entry in the contact database of the mobile device.

Syntax

```
ui.Interface.frontCall("ios", "newContact",
  [defaults],[vcard])
```

- *defaults* - A vCard string with default values for the new contact input.

- *vcard* - Holds the vCard string of the new created contact.

Usage

The iOS "`newContact`" front call opens the contact input form on the mobile device, with default values passed in the vCard structure of the first parameter, lets the user enter contact information.

> **Important:** This front call is only available for an application running on an iOS device.

If the contact creation is validated, the front call returns the completed vCard string. If the contact import is canceled, the front-end returns `NULL`.

Example

```
DEFINE defaults, vcard STRING
LET defaults="BEGIN:VCARD\n"
      ||"VERSION:3.0\n"
      ||"N:Willi;;;;\n"
      ||"TEL;type=HOME;type=VOICE;type=pref:03812225610\n"
      ||"END:VCARD\n"
CALL ui.interface.frontcall("ios","newContact",[defaults],
[vcard])
```

setBadgeNumber (iOS)

Sets the current badge number associated to the app.

Syntax

```
ui.Interface.frontCall("ios", "setBadgeNumber",
  [value], [])
```

- *value* - Holds the badge number to be set.

Usage

The iOS "`setBadgeNumber`" front call sets the badge number associated to the app.

> **Important:** This front call is only available for an application running on an iOS device.

The badge number appears on the app icon and is typically used for Push notifications on page 2599.

> **Important:** In order to query or set the badge number, the app program must have executed a `registerForRemoteNotifications` front call before (in the current or prior execution instance). This registration is required in order to set the appropriate app permissions to access badge number data.

Example

```
DEFINE value INTEGER
LET value = 2
CALL ui.interface.frontcall("ios","setBadgeNumber",[value],[])
```

Extension packages

Several utility classes and functions are provided in additional packages to be included with the `IMPORT` instruction.

- The util package on page 1947
- The os package on page 1990
- The com package on page 2009
- The xml package on page 2103
- The security package on page 2278

The util package

These topics cover the classes for the `util` package.

The util.Date class

The `util.Date` class provides `DATE` data-type related utility methods.

This class is provided in the `util` C-Extension library; To use the `util.Date` class, you must import the `util` package in your program:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

util.Date methods

Methods for the `util.Date` class.

Table 436: Class methods

Name	Description
`util.Date.isLeapYear( year )` `  RETURNING res BOOLEAN`	Checks is the year passed as parameter is a leap year.

`util.Date.parse(`
`  src STRING,`
`  fmt STRING`
`  )`
`  RETURNING res DATE`
	Converts a string to a `DATE` value according to a format specification.

util.Date.parse

Converts a string to a `DATE` value according to a format specification.

Syntax

```
util.Date.parse(
  src STRING,
  fmt STRING
 )
  RETURNING res DATE
```

1. *src* is the source string to be parsed.
2. *fmt* is the format specification (see Formatting DATE values on page 220).

Usage

The `util.Date.parse()` method parses a string according to a format specification, to produce a `DATE` value.

The format specification must be a combination of `dd`, `mm`, `yyyy` place holders as with the `USING` operator.

The method returns `NULL`, if the source string cannot be converted to a `DATE` value according to the format specification.

For more details about the supported formats, see Formatting DATE values on page 220.

Example

```
IMPORT util
MAIN
    DISPLAY util.Date.parse( "2014-03-15", "yyyy-mm-dd" )
END MAIN
```

util.Date.isLeapYear
Checks is the year passed as parameter is a leap year.

Syntax

```
util.Date.isLeapYear( year )
  RETURNING res BOOLEAN
```

1. *year* is an `INTEGER` representing a year.
2. *res* is `TRUE` if year is a leap year, otherwise res is `FALSE`.

Usage

The `util.Date.isLeapYear()` method returns `TRUE` if the year passed in parameter is a leap year.

Example

```
IMPORT util
MAIN
    DISPLAY util.Date.isLeapYear( 2003 )
    DISPLAY util.Date.isLeapYear( 2004 )
END MAIN
```

The util.Datetime class
The `util.Datetime` class provides `DATETIME` data-type related utility methods.

This class is provided in the `util` C-Extension library; To use the `util.Datetime` class, you must import the `util` package in your program:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

util.Datetime methods

Methods for the `util.Datetime` class.

Table 437: Class methods

Name	Description

`util.Datetime.format(`
`  value DATETIME q1 TO q2,`
`  fmt STRING`
`)`
`  RETURNING res STRING`
	Formats a datetime value according to format specification.

`util.Datetime.fromSecondsSinceEpoch(`
`  seconds FLOAT`
`)`
`  RETURNING local DATETIME q1 TO q2`
	Converts a number of seconds since Epoch to a datetime.

`util.Datetime.getCurrentAsUTC( )` `  RETURNING utc DATETIME YEAR TO FRACTION(5)`	Returns the current date/time in UTC.

`util.Datetime.parse(`
`  src STRING,`
`  fmt STRING`
`)`
`  RETURNING res DATETIME q1 TO q2`
	Converts a string to a `DATETIME` value according to a format specification.

`util.Datetime.toLocalTime(`
`  utc DATETIME q1 TO q2`
`)`
`  RETURNING local DATETIME q1 TO q2`
	Converts a UTC datetime to the local time.

`util.Datetime.toSecondsSinceEpoch(`
`  local DATETIME q1 TO q2`
`)`
`  RETURNING seconds FLOAT`
	Converts a datetime to a number of seconds since Epoch.

`util.Datetime.toUTC(`
`  local DATETIME q1 TO q2`
`)`
`  RETURNING utc DATETIME q1 TO q2`
	Converts a datetime value to the UTC datetime.

util.Datetime.format

Formats a datetime value according to format specification.

Syntax

```
util.Datetime.format(
   value DATETIME q1 TO q2,
   fmt STRING
)
```

```
  RETURNING res STRING
```

1. *value* is the datetime value to be formatted.
2. *fmt* is the format string, as described in Formatting DATETIME values on page 221.

Usage

The `util.Datetime.format()` method formats a `DATETIME` value according to the format specification.

The format string must be a combination of place holders such as `%Y`, `%m`, `%d`, as described in Formatting DATETIME values on page 221.

If the source value is `NULL` the result will be `NULL`.

Example

```
IMPORT util
MAIN
    DISPLAY util.Datetime.format( CURRENT, "%Y-%m-%d %H:%M" )
END MAIN
```

util.Datetime.fromSecondsSinceEpoch
Converts a number of seconds since Epoch to a datetime.

Syntax

```
util.Datetime.fromSecondsSinceEpoch(
  seconds FLOAT
 )
  RETURNING local DATETIME q1 TO q2
```

1. *seconds* is the number of seconds since Epoch. This can be a whole integer or a decimal, if the target datetime
2. *local* is the local datetime value.

Usage

The `util.Datetime.fromSecondsSinceEpoch()` method converts the number of seconds since the Unix Epoch (1970-01-01 00:00:00 GMT) passed as parameter, to a `DATETIME` value, in the local time.

Important: If the number of seconds passed as parameter is a floating point number including a fraction of seconds, the result will be a `DATETIME YEAR TO FRACTION(N)`, otherwise it is `DATETIME YEAR TO SECOND`.

Example

```
IMPORT util
MAIN
    DEFINE dt DATETIME YEAR TO SECOND
    LET dt = util.Datetime.fromSecondsSinceEpoch( 9876234 )
    DISPLAY dt
END MAIN
```

util.Datetime.getCurrentAsUTC
Returns the current date/time in UTC.

Syntax

```
util.Datetime.getCurrentAsUTC( )
  RETURNING utc DATETIME YEAR TO FRACTION(5)
```

1. *utc* is the datetime value in UTC, with the precision `DATETIME YEAR TO FRACTION(5)`.

Usage

The `util.Datetime.getCurrentAsUTC()` method returns the current system date/time in UTC (Universal Time).

This method is provided to solve the daylight saving time transition issue of the `util.Datetime.toUTC()` method.

> **Note:** The precision of the value returned by this method is a `DATETIME YEAR TO FRACTION(5)`. Note that this precision is different from the default `CURRENT` precision when no qualifiers are specified.

Example

```
IMPORT util
MAIN
    DEFINE utc DATETIME YEAR TO FRACTION(5)
    LET utc = util.Datetime.getCurrentAsUTC( )
    DISPLAY "Current UTC: ", utc
END MAIN
```

util.Datetime.parse
Converts a string to a `DATETIME` value according to a format specification.

Syntax

```
util.Datetime.parse(
  src STRING,
  fmt STRING
)
  RETURNING res DATETIME q1 TO q2
```

1. *src* is the source string to be parsed.
2. *fmt* is the format specification (see Formatting DATETIME values on page 221).

Usage

The `util.Datetime.parse()` method parses a string according to a format specification, to produce a `DATETIME` value.

The format specification must be a combination of place holders such as `%Y`, `%m`, `%d`, etc.

The precision of the resulting `DATETIME` value depends on the format specification. For example, when using `"%y-%m-%d %H:%M"`, the resulting value will be a `DATETIME YEAR TO MINUTE`.

The method returns `NULL`, if the source string cannot be converted to a `DATETIME` value according to the format specification.

For more details about the supported formats, see Formatting DATETIME values on page 221.

Example

```
IMPORT util
MAIN
    DEFINE dt DATETIME YEAR TO MINUTE
    LET dt = util.Datetime.parse( "2014-12-24 23:45", "%Y-%m-%d
 %H:%M" )
    DISPLAY dt
END MAIN
```

util.Datetime.toLocalTime
Converts a UTC datetime to the local time.

Syntax

```
util.Datetime.toLocalTime(
  utc DATETIME q1 TO q2
 )
  RETURNING local DATETIME q1 TO q2
```

1. *utc* is the datetime value in UTC.
2. *local* is the local tinezone datetime value.

Usage

The `util.Datetime.toLocalTime()` method converts a `DATETIME` value from "Coordinated Universal Time" (UTC), also known as "Greenwich Mean Time" (GMT), to the local timezone datetime.

Example

```
IMPORT util
MAIN
    DEFINE loc DATETIME YEAR TO SECOND
    LET loc = util.Datetime.toLocalTime( DATETIME(2015-08-22
 15:34:56) YEAR TO SECOND )
    DISPLAY "LOC: ", loc
END MAIN
```

util.Datetime.toSecondsSinceEpoch
Converts a datetime to a number of seconds since Epoch.

Syntax

```
util.Datetime.toSecondsSinceEpoch(
  local DATETIME q1 TO q2
 )
  RETURNING seconds FLOAT
```

1. *local* is the local datetime value.
2. *seconds* is the number of seconds since Epoch. Note that this is a `FLOAT` value as the source can be a `DATETIME YEAR TO FRACTION(N)`.

Usage

The `util.Datetime.toSecondsSinceEpoch()` method converts the `DATETIME` value passed as parameter to a number of seconds since the Unix Epoch (1970-01-01 00:00:00 GMT)

Important: The result is a whole number when the source is a `DATETIME YEAR TO SECOND`, but will be a floating point number when the source is a `DATETIME YEAR TO FRACTION(N)`, to include the fractional part.

Example

```
IMPORT util
MAIN
    DEFINE sec INTEGER, loc DATETIME YEAR TO SECOND
    LET loc = CURRENT YEAR TO SECOND
    LET sec = util.Datetime.toSecondsSinceEpoch( loc )
    DISPLAY sec
END MAIN
```

util.Datetime.toUTC
Converts a datetime value to the UTC datetime.

Syntax

```
util.Datetime.toUTC(
   local DATETIME q1 TO q2
)
  RETURNING utc DATETIME q1 TO q2
```

1. *local* is the local timezone datetime value.
2. *utc* is the datetime value in UTC.

Usage

The `util.Datetime.toUTC()` method converts the local timezone `DATETIME` value passed as parameter to the "Coordinated Universal Time" (UTC), also known as "Greenwich Mean Time" (GMT).

The `toUTC()` method on local timezone information settings.

Fall/Autumn daylight saving time transition period

Important: The `toUTC()` function cannot determine if the local datetime value represents a time before or after the daylight saving time change, when the value is in the hour of the daylight saving time transition period in the fall (this is for example, the hour 02:00 PM to 03:00 PM on the last Sunday of October in Europe and first Sunday of November in the USA). Depending on the operating system, the `toUTC()` method can interpret the local time as a Summer time or as a Winter time. In order to get the current system time in UTC, use the `util.Datetime.getCurrentAsUTC()` method.

The `DATETIME` value passed as parameter to the `toUTC()` method is the datetime in the local timezone. However, this value does not contain the GMT offset indicator or daylight saving time information.

When passing local datetime values in the hour of the daylight saving time transition period in the fall (when clocks roll back one hour), the `toUTC()` function cannot determine if the local datetime value represents a point in time before or after the daylight saving time transition occured. Depending on the operating system, the `toUTC()` method can interpret the local time as a Summer time or as a Winter time. As a result, the conversion to the UTC time can be mis-interpreted.

For example, in Europe, the fall daylight saving time changes on the 25 of October, at 3:00 PM. The ambigous period is between 2:00 PM and 3:00 PM (local time). If you pass for example, the datetime value 2015-10-25 02:34:11 to the `toUTC()` method, there is no way for the method to know if this local time is the time before (CEST / UTC+2h) or after (CET / UTC+1h) the daylight saving time change.

This behavior can be illustrated with the following code example:

```
IMPORT util

MAIN
    DISPLAY "Original UTC         Local time (Paris)   toUTC(local-time)
 ( toUTC() - Orig UCT )"
    CALL test( "2015-10-24 23:59:59" )
    CALL test( "2015-10-25 00:59:59" )
    CALL test( "2015-10-25 01:59:59" )
    CALL test( "2015-10-25 02:59:59" )
END MAIN

FUNCTION test(utc)
    DEFINE utc, loc, utc2 DATETIME YEAR TO SECOND
    LET loc = util.Datetime.toLocalTime(utc)
    LET utc2 = util.Datetime.toUTC(loc)
    DISPLAY SFMT("%1  %2  %3  %4", utc,loc,utc2,utc2-utc)
END FUNCTION
```

The above code will produce the following output on Linux, with with `TZ='Europe/Paris'`:

```
Original UTC         Local time (Paris)   toUTC(local-time)     ( toUTC() -
 Orig UCT )
2015-10-24 23:59:59  2015-10-25 01:59:59  2015-10-24 23:59:59          0
 00:00:00
2015-10-25 00:59:59  2015-10-25 02:59:59  2015-10-25 00:59:59          0
 00:00:00
2015-10-25 01:59:59  2015-10-25 02:59:59  2015-10-25 00:59:59         -0
 01:00:00
2015-10-25 02:59:59  2015-10-25 03:59:59  2015-10-25 02:59:59          0
 00:00:00
```

As you can see, the local time 2015-10-25 02:59:59 is always converted to UTC 2015-10-25 00:59:59.

Example

```
IMPORT util
MAIN
    DEFINE utc DATETIME YEAR TO SECOND
    LET utc = util.Datetime.toUTC( DATETIME(2015-08-22 15:34:56)
 YEAR TO SECOND )
    DISPLAY "UTC: ", utc
END MAIN
```

The util.Interval class

The `util.Interval` class provides `INTERVAL` data-type related utility methods.

This class is provided in the `util` C-Extension library; To use the `util.Interval` class, you must import the `util` package in your program:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

util.Interval methods

Methods for the `util.Interval` class.

Table 438: Class methods

Name	Description

`util.Interval.format(`
`  value INTERVAL q1 TO q2,`
`  fmt STRING`
`)`
`  RETURNING res STRING`
	Formats an interval value according to format specification.

`util.Interval.parse(`
`  src STRING,`
`  fmt STRING`
`)`
`  RETURNING res DATETIME q1 TO q2`
	Converts a string to a `DATETIME` value according to a format specification.

util.Interval.format

Formats an interval value according to format specification.

Syntax

```
util.Interval.format(
   value INTERVAL q1 TO q2,
   fmt STRING
)
  RETURNING res STRING
```

1. *value* is the interval value to be formatted.
2. *fmt* is the format string, as described in Formatting INTERVAL values on page 223.

Usage

The `util.Interval.format()` method formats an `INTERVAL` value according to the format specification.

The format string must be a combination of place holders such as `%Y`, `%m`, `%d`, as described in Formatting INTERVAL values on page 223.

If the source value is `NULL` the result will be `NULL`.

Example

```
IMPORT util
MAIN
    DEFINE iv INTERVAL DAY(6) TO MINUTE
    LET iv = "-157 11:23"
    DISPLAY util.Interval.format(iv, "%d %H:%M")
END MAIN
```

util.Interval.parse
Converts a string to a DATETIME value according to a format specification.

Syntax

```
util.Interval.parse(
  src STRING,
  fmt STRING
)
  RETURNING res DATETIME q1 TO q2
```

1. *src* is the source string to be parsed.
2. *fmt* is the format specification (see Formatting INTERVAL values on page 223).

Usage

The `util.Interval.parse()` method parses a string according to a format specification, to produce an `INTERVAL` value.

The format specification must be a combination of place holders such as `%Y`, `%m`, `%d`, etc.

The precision of the resulting `INTERVAL` value depends on the format specification. For example, when using `"%Y-%m"`, the resulting value will be an `INTERVAL YEAR TO MONTH`.

The method returns `NULL`, if the source string cannot be converted to an `INTERVAL` value according to the format specification.

For more details about the supported formats, see Formatting INTERVAL values on page 223.

Example

```
IMPORT util
MAIN
    DEFINE iv INTERVAL DAY(6) TO FRACTION(5)
    LET iv = util.Interval.parse( "-37467 + 23:45:34.12345", "%d
 + %H:%M:%S%F5" )
END MAIN
```

The util.Strings class

The `util.Strings` class provides `STRING` data-type related utility methods.

This class is provided in the `util` C-Extension library; To use the `util.Strings` class, you must import the `util` package in your program:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

util.Strings methods

Methods for the `util.Strings` class.

Table 439: Class methods

Name	Description
`util.Strings.base64Decode( source STRING, filename STRING`	Decodes a Base64 encoded string and writes the bytes to a file.

Name	Description
)	

`util.Strings.base64Encode(`
`  filename STRING`
`  )`
`  RETURNING result STRING`
	Converts the content of a file to a Base64 encoded string.

`util.Strings.base64DecodeToString(`
`  source STRING`
`  )`
`  RETURNING result STRING`
	Decodes a base64 encoded string and returns the corresponding string.

`util.Strings.base64EncodeFromString(`
`  source STRING`
`  )`
`  RETURNING result STRING`
	Converts the string passed as parameter to a Base64 encoded string.

`util.Strings.urlDecode(`
`  source STRING`
`  )`
`  RETURNING result STRING`
	Converts the URL-encoded string to a string in the current application locale.

`util.Strings.urlEncode(`
`  source STRING`
`  )`
`  RETURNING result STRING`
	Converts a string from the current codeset to a URL-encoded string.

util.Strings.base64Decode
Decodes a Base64 encoded string and writes the bytes to a file.

Syntax

```
util.Strings.base64Decode(
  source STRING,
  filename STRING
 )
```

1. *source* is the Base64 encoded string.
2. *filename* is the name of the file to write to.

Usage

The `util.Strings.base64Decode()` method converts the Base64 encoded string passed as first parameter, and writes the bytes to file specified as second parameter.

Example

```
IMPORT util
MAIN
    DEFINE base64 STRING
    LET base64 = util.Strings.base64Encode( "picture1.png" )
    DISPLAY base64
```

```
    CALL util.Strings.base64Decode( base64, "picture2.png" )
END MAIN
```

util.Strings.base64Encode
Converts the content of a file to a Base64 encoded string.

Syntax

```
util.Strings.base64Encode(
  filename STRING
 )
  RETURNING result STRING
```

1. *filename* is the name of the file to read from.
2. *result* is the resulting Base64 encoded string.

Usage

The `util.Strings.base64Encode()` method reads the content of the file passed as parameter, and converts the bytes to a Base64 encoded string.

Example

```
IMPORT util
MAIN
    DISPLAY util.Strings.base64Encode( "picture.png" )
END MAIN
```

util.Strings.base64DecodeToString
Decodes a base64 encoded string and returns the corresponding string.

Syntax

```
util.Strings.base64DecodeToString(
  source STRING
 )
  RETURNING result STRING
```

1. *source* is the Base64 encoded string.
2. *result* is the decoded string.

Usage

The `util.Strings.base64DecodeToString()` method converts the Base64 encoded string passed as parameter to an array of bytes, then it converts the byte array to a string representation in the current locale, and returns that string.

If the Base64 source string contains a sequence of bytes that does not represent a valid character in the current application locale, the function returns `NULL`.

Note: In contrast to util.Strings.urlDecode on page 1959, the original string is not converted from UTF-8 to the application character encoding: The Base64 source string must represent valid characters in the current application locale.

Example

```
IMPORT util
MAIN
```

```
    DEFINE base64 STRING
    LET base64 = util.Strings.base64EncodeFromString( "Forêt" )
    DISPLAY base64
    DISPLAY util.Strings.base64DecodeToString( base64 )
END MAIN
```

util.Strings.base64EncodeFromString
Converts the string passed as parameter to a Base64 encoded string.

Syntax

```
util.Strings.base64EncodeFromString(
  source STRING
 )
  RETURNING result STRING
```

1. *source* is the source string to convert in Base64.
2. *result* is the resulting Base64 encoded string.

Usage

The `util.Strings.base64EncodeFromString()` method first converts the string passed as parameter to an array of bytes, then it converts the array of bytes the a Base64 representation, and returns the resulting Base64 encoded string.

> **Note:** In contrast to util.Strings.urlEncode on page 1960, the original string is not converted from the application locale to UTF-8, before performing the encoding to Base64: The resulting Base64 encoded string will contain byte sequences representing characters in the current application locale.

Example

```
IMPORT util
MAIN
    DISPLAY util.Strings.base64EncodeFromString( "Forêt" )
END MAIN
```

util.Strings.urlDecode
Converts the URL-encoded string to a string in the current application locale.

Syntax

```
util.Strings.urlDecode(
  source STRING
 )
  RETURNING result STRING
```

1. *source* is the URL-encoded source string (UTF-8 bytes).
2. *result* is the resulting Base64 encoded string.

Usage

The `util.Strings.urlDecode()` method converts the URL-encoded string passed as parameter to a character string.

The source string must contain ASCII characters and/or `%xx` hexadecimal representation of UTF-8 encoding bytes.

The decoder is error tolerant:

- Alphabetical characters of a %xx element can be uppercase or lowercase (%b2 = %B2).
- If the source string contains a set of %xx elements that represent a UTF-8 encoded character which is not existing in the current application locale, it will be converted to a ? question mark.
- If the percent character is not followed by two hexadecimal digits, then a "%" is copied to the result string and the decoder continues at the next character.

Example

```
IMPORT util
MAIN
    DISPLAY util.Strings.urlDecode("abc%C3%84%E2%82%AC")
END MAIN

Output:
abcÄ€
```

util.Strings.urlEncode
Converts a string from the current codeset to a URL-encoded string.

Syntax

```
util.Strings.urlEncode(
  source STRING
 )
  RETURNING result STRING
```

1. *source* is the source string to url-encode.
2. *result* is the resulting url encoded string.

Usage

The `util.Strings.urlEncode()` method converts the character string passed as parameter to a URL-encoded string.

All characters not matching `[-_.~a-zA-Z0-9]` are "percent encoded": Percent-encoding involves converting those characters to UTF-8 and representing its corresponding byte values by a percent sign ("%") and a pair of hexadecimal digits.

Example

```
IMPORT util
MAIN
    DISPLAY util.Strings.urlEncode("abcÄ€")
END MAIN

Output:
abc%C3%84%E2%82%AC
```

The util.Math class

The `util.Math` class provides basic mathematical functions based on floating point numbers (`FLOAT`).

This class does not have to be instantiated; it provides class methods for the current program.

This class is provided in the `util` C-Extension library; To use the `util.Math` class, you must import the `util` package in your program:

```
IMPORT util
```

util.Math methods

Methods for the `util.Math` class.

Table 440: Class methods

Name	Description
`util.Math.acos( val FLOAT ) RETURNING result FLOAT`	Computes the arc cosine of the passed value, measured in radians.
`util.Math.asin( val FLOAT ) RETURNING result FLOAT`	Computes the arc sine of the passed value, measured in radians.
`util.Math.atan( val FLOAT ) RETURNING result FLOAT`	Computes the arc tangent of the passed value, measured in radians.
`util.Math.cos( val FLOAT ) RETURNING result FLOAT`	Computes the cosine of the passed value, measured in radians.
`util.Math.exp( val FLOAT ) RETURNING result FLOAT`	Computes the base-e exponential of the value passed as parameter.
`util.Math.log( val FLOAT ) RETURNING result FLOAT`	Computes the natural logarithm of the passed value.
`util.Math.pi() RETURNING result FLOAT`	Returns the FLOAT value of PI.
`util.Math.pow( x FLOAT, y FLOAT ) RETURNING result FLOAT`	Computes the value of *x* raised to the power *y*.
`util.Math.rand( max INTEGER ) RETURNING result INTEGER`	Returns a positive pseudo-random number.
`util.Math.sin( val FLOAT ) RETURNING result FLOAT`	Computes the sine of the passed value, measured in radians.
`util.Math.sqrt( val FLOAT )`	Returns the square root of the argument provided.

Name	Description
`RETURNING result FLOAT`	
`util.Math.srand()`	Initializes the pseudo-random numbers generator.
`util.Math.tan( val FLOAT ) RETURNING result FLOAT`	Computes the tangent of the passed value, measured in radians.
`util.Math.toDegrees( val FLOAT ) RETURNING result FLOAT`	Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
`util.Math.toRadians( val FLOAT ) RETURNING result FLOAT`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

util.Math.acos
Computes the arc cosine of the passed value, measured in radians.

Syntax

```
util.Math.acos(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided is invalid.

util.Math.asin
Computes the arc sine of the passed value, measured in radians.

Syntax

```
util.Math.asin(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided is invalid.

util.Math.atan
Computes the arc tangent of the passed value, measured in radians.

Syntax

```
util.Math.atan(
```

```
    val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided is invalid.

util.Math.cos
Computes the cosine of the passed value, measured in radians.

Syntax

```
util.Math.cos(
    val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage:

Returns `NULL` if the argument provided is invalid.

util.Math.exp
Computes the base-e exponential of the value passed as parameter.

Syntax

```
util.Math.exp(
    val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided on error.

util.Math.pi
Returns the FLOAT value of PI.

Syntax

```
util.Math.pi()
  RETURNING result FLOAT
```

util.Math.pow
Computes the value of *x* raised to the power *y*.

Syntax

```
util.Math.pow(
    x FLOAT,
    y FLOAT )
  RETURNING result FLOAT
```

1. *x* is the value to be raised.
2. *y* is the power operand.

Usage

The function returns `NULL` if one of the argument provided is invalid.

If *x* is negative, the caller should ensure that *y* is an integer value.

util.Math.rand
Returns a positive pseudo-random number.

Syntax

```
util.Math.rand(
   max INTEGER )
  RETURNING result INTEGER
```

1. *max* is the maximum random number that can be generated.

Usage

The `rand()` function returns a pseudo-random integer number between zero and *max*.

> **Important:**
>
> The `srand()` function initializes the pseudo-random numbers generator. It must be called before subsequent calls to the `rand()` function. If you do not call the `srand()` function, the `rand()` function will generate the same sequence of numbers for every program execution. The numbers generated by `rand()` can vary according to the operating system.

The maximum random number returned by the `rand()` function is 2,147,483,646.

The `rand()` function returns zero if the argument is lower or equal to 0.

Example

```
IMPORT util
MAIN
    DEFINE i SMALLINT
    DISPLAY "Before srand() call:"
    FOR i=1 TO 3
        DISPLAY util.Math.rand(100)
    END FOR
    CALL util.Math.srand()
    DISPLAY "After srand() call:"
    FOR i=1 TO 3
        DISPLAY util.Math.rand(100)
    END FOR
END MAIN
```

(run this example several times)

util.Math.sin
Computes the sine of the passed value, measured in radians.

Syntax

```
util.Math.sin(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided is invalid.

util.Math.sqrt
Returns the square root of the argument provided.

Syntax

```
util.Math.sqrt(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

The function returns `NULL` if the argument provided is invalid.

util.Math.srand
Initializes the pseudo-random numbers generator.

Syntax

```
util.Math.srand()
```

Usage

The `srand()` function initializes the pseudo-random numbers generator. It must be called before subsequent calls to the `rand()` function. If you do not call the `srand()` function, the `rand()` function will generate the same sequence of numbers for every program execution. The numbers generated by `rand()` can vary according to the operating system.

util.Math.tan
Computes the tangent of the passed value, measured in radians.

Syntax

```
util.Math.tan(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided is invalid.

util.Math.log
Computes the natural logarithm of the passed value.

Syntax

```
util.Math.log(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value.

Usage

Returns `NULL` if the argument provided is invalid.

util.Math.toDegrees
Converts an angle measured in radians to an approximately equivalent angle measured in degrees.

Syntax

```
util.Math.toDegrees(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value to be converted to degrees.

util.Math.toRadians
Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Syntax

```
util.Math.toRadians(
   val FLOAT )
  RETURNING result FLOAT
```

1. *val* is a floating point value to be converted to radians.

The util.JSON class
The `util.JSON` class provides a basic interface to convert program variable values to/from JSON data.

The `util.JSON` class is provided in the `util` C-Extension library; To use the `util.JSON` class, you must import the `util` package in your program:

```
IMPORT util
```

This class does not have to be instantiated; it provides class methods for the current program.

The purpose of the `util.JSON` class is to convert a JSON string from/to a BDL variable, to interface with other software based on the JSON format.

The BDL variable can be a simple variable (defined with a primitive type), a structured variable (`RECORD`), or dynamic array.

It is not possible to modify JSON elements with this class. In order to manipulate JSON objects, use the `util.JSONObject` and `util.JSONArray` classes.

util.JSON methods
Methods for the `util.JSON` class.

Table 441: Class methods

Name	Description
`util.JSON.format( source STRING ) RETURNING result STRING`	Formats JSON string with indentation.
`util.JSON.parse( source STRING,`	Parses a JSON string and fills program variables with the values.

Name	Description
destination { RECORD | DYNAMIC ARRAY })	
util.JSON.proposeType(*source* STRING) RETURNING *result* STRING	Describes the record structure that can hold a given JSON data string.
util.JSON.stringify(*source* { RECORD | DYNAMIC ARRAY }) RETURNING *result* STRING	Transforms a record variable to a flat JSON formatted string.

util.JSON.format
Formats JSON string with indentation.

Syntax

```
util.JSON.format(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string value that contains JSON formatted data.
2. *result* is a string that is well formatted and indented.

Usage

The `util.JSON.format()` class method takes a JSON formatted string as parameter and indents the JSON string.

The main purpose of this method is to beautify a JSON data string that is on a single line, by adding line breaks and indentation.

util.JSON.parse
Parses a JSON string and fills program variables with the values.

Syntax

```
util.JSON.parse(
   source STRING,
   destination { RECORD | DYNAMIC ARRAY } )
```

1. *source* is a string value that contains JSON formatted data.
2. *destination* is the variable to be initialized with values of the JSON string.

> **Important:** The *dest* record is passed by reference to the method.

Usage

The `util.JSON.parse()` class method scans the JSON source string passed as parameter and fills the destination variable members by name.

The destination variable should have the same structure as the JSON source data, it can be a `RECORD` or a `DYNAMIC ARRAY`.

See JSON to Genero BDL conversion rules on page 1987 for details on how the destination variable is populated when the structures are not identical.

Example

```
IMPORT util
MAIN
    DEFINE cust_rec RECORD
               cust_num INTEGER,
               cust_name VARCHAR(30),
               order_ids DYNAMIC ARRAY OF INTEGER
           END RECORD
    DEFINE js STRING
    LET js='{ "cust_num":2735, "cust_name":"McCarlson",
              "order_ids":[234,3456,24656,34561] }'
    CALL util.JSON.parse( js, cust_rec )
    DISPLAY cust_rec.cust_name
    DISPLAY cust_rec.order_ids[4]
END MAIN
```

util.JSON.proposeType
Describes the record structure that can hold a given JSON data string.

Syntax

```
util.JSON.proposeType(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string value that contains JSON formatted data.
2. *result* is a string that represents the definition of a `RECORD`.

Usage

The `util.JSON.proposeType()` class method takes a JSON formatted string as parameter and generates the `RECORD` definition that corresponds to the source JSON string.

This method is useful to define a record variable that must hold the given JSON string.

Example

```
IMPORT util
MAIN
    DEFINE js STRING
    LET js='{ "cust_num":2735, "cust_name":"McCarlson",
              "orderids":[234,3456,24656,34561] }'
    DISPLAY util.JSON.proposeType( js )
END MAIN
```

Displays:

```
 RECORD
    cust_num FLOAT,
    cust_name STRING,
    orderids DYNAMIC ARRAY OF FLOAT
END RECORD
```

util.JSON.stringify
Transforms a record variable to a flat JSON formatted string.

Syntax

```
util.JSON.stringify(
   source { RECORD | DYNAMIC ARRAY } )
  RETURNING result STRING
```

1. *source* is the program variable to be converted to a JSON string.
2. *result* is a JSON formatted string created from the source record.

Usage

The `util.JSON.stringify()` class method takes a `RECORD` or `DYNAMIC ARRAY` variable as parameter, and generates the corresponding data string in JSON format, as defined in the [RFC4627] specification.

For more details about FGL to JSON conversion, see Genero BDL to JSON conversion rules on page 1988.

The method raises error -8110 if the JSON string cannot be generated.

Example

```
IMPORT util
MAIN
    DEFINE cust_rec RECORD
                cust_num INTEGER,
                cust_name VARCHAR(30),
                order_ids DYNAMIC ARRAY OF INTEGER
            END RECORD
    DEFINE js STRING
    LET cust_rec.cust_num = 345
    LET cust_rec.cust_name = "McMaclum"
    LET cust_rec.order_ids[1] = 4732
    LET cust_rec.order_ids[2] = 9834
    LET cust_rec.order_ids[3] = 2194
    LET js = util.JSON.stringify( cust_rec )
    DISPLAY util.JSON.format( js )
END MAIN
```

Displays following output:

```
{
    "cust_num": 345,
    "cust_name": "McMaclum",
    "order_ids": [4732,9834,2194
    ]
}
```

Examples

Example 1: Reading a JSON file

This program reads JSON data from `customers.json`, parses the line to fill the program variables, coverts the program variable back to JSON and writes a formatted JSON string to the standard output.

We assume that the source file contains the list of customers in a single line that can be read with `base.Channel.readLine()`:

```
IMPORT util
MAIN
  DEFINE custlist DYNAMIC ARRAY OF RECORD
            num INTEGER,
            name VARCHAR(40)
          END RECORD
  DEFINE ch base.Channel
  LET ch = base.Channel.create()
  CALL ch.openFile("customers.json","r")
  CALL util.JSON.parse( ch.readLine(), custlist )
  DISPLAY custlist.getLength()
  DISPLAY util.JSON.format( util.JSON.stringify(custlist) )
  CALL ch.close()
END MAIN

-- customers.json file:
[ {"num":823, "name":"Mark Renbing" }, {"num":234, "name":"Clark
 Gambler" } ]
```

Note that the JSON file does not contain the name of the dynamic array (custlist), but starts directly with the JSON array in `[ ]` square braces.

The util.JSONObject class

The `util.JSONObject` class provides methods to handle an structured data object following the JSON string syntax.

The `util.JSONObject` class is provided in the `util` C-Extension library; To use the `util.JSONObject` class, you must import the `util` package in your program:

```
IMPORT util
```

A `JSONObject` is an unordered collection of name/value pairs. The format of a JSON object string is a coma-separated "name":value pairs, wrapped in curly braces. The value can be simple numeric or string value, but it can also be an array of values enclosed in square braces, or a sub-element enclosed in curly braces:

```
{
  "cust_num":2735,
  "cust_name":"McCarlson",
  "order_ids":[234,3456,24656,34561],
  "address": {
          "street":"34, Sunset Bld",
          "city":"Los Angeles",
          "state":"CA"
        }
}
```

A `JSONObject` object must be created before usage with one of the class methods like `util.JSONObject.create()`.

The `JSONObject` class provides methods for accessing, adding/replacing or deleting the values by name with the `get()`, `put()` and `remove()` methods.

The `get()` method can return a simple value, a `util.JSONObject` or a `util.JSONArray` object reference.

The `put()` method can take a simple value, a `RECORD`, or an `ARRAY` as parameter.

If the structure of the JSON object is not known at compile time, you can introspect the elements of the object with the `getLength()`, `getType()` and `name()` methods.

util.JSONObject methods
Methods for the `util.JSONObject` class.

Table 442: Class methods

Name	Description
`util.JSONObject.create()` `  RETURNING object util.JSONObject`	Creates a new JSON object.

`util.JSONObject.fromFGL(`
`  source RECORD )`
`  RETURNING object util.JSONObject`
	Creates a new JSON object from a `RECORD`.

`util.JSONObject.parse(`
`  source STRING )`
`  RETURNING result util.JSONObject`
	Parses a JSON string and creates a JSON object from it.

Table 443: Object methods

Name	Description

`util.JSONObject.get(`
`  name STRING )`
`  RETURNING result result-type`
	Returns the value corresponding to the specified entry name.

`util.JSONObject.getLength()` `  RETURNING len INTEGER`	Returns the number of name-value pairs in the JSON object.

`util.JSONObject.getType(`
`  name STRING )`
`  RETURNING type STRING`
	Returns the type of a JSON object element.

`util.JSONObject.has(`
`  name STRING )`
`  RETURNING result BOOLEAN`
	Checks if the JSON object contains a specific entry name.

`util.JSONObject.name(`
`  index INTEGER )`
`  RETURNING result STRING`
	Returns the name of a JSON object entry by position.

`util.JSONObject.put(`
`  name STRING,`
`  value value-type )`
	Sets a name-value pair in the JSON object.

`util.JSONObject.remove(`	Removes the specified element in the JSON object.

Name	Description
name STRING)	
util.JSONObject.toFGL(*dest* RECORD)	Fills a record variable with the entries contained in the JSON object.
util.JSONObject.toString() RETURNING *result* STRING	Builds a JSON string from the values contained in the JSON object.

util.JSONObject.create
Creates a new JSON object.

Syntax

```
util.JSONObject.create()
  RETURNING object util.JSONObject
```

Usage

The `util.JSONObject.create()` method create a new JSON object.

The new created object must be assigned to a program variable defined with the `util.JSONObject` type.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    LET obj = util.JSONObject.create()
    ...
END MAIN
```

util.JSONObject.fromFGL
Creates a new JSON object from a `RECORD`.

Syntax

```
util.JSONObject.fromFGL(
   source RECORD )
  RETURNING object util.JSONObject
```

1. *source* is the `RECORD` variable used to create the JSON object.

Usage

The `util.JSONObject.fromFGL()` method creates a new JSON object from the `RECORD` variable passed as parameter.

The new created object must be assigned to a program variable defined with the `util.JSONObject` type.

The members of the `RECORD` are converted to name/value pairs in the JSON object.

For more details about FGL to JSON conversion, see Genero BDL to JSON conversion rules on page 1988.

Example

```
IMPORT util
MAIN
    DEFINE cust_rec RECORD
                cust_num INTEGER,
                cust_name VARCHAR(30),
                order_ids DYNAMIC ARRAY OF INTEGER
            END RECORD
    DEFINE obj util.JSONObject
    LET cust_rec.cust_num = 345
    LET cust_rec.cust_name = "McMaclum"
    LET cust_rec.order_ids[1] = 4732
    LET cust_rec.order_ids[2] = 9834
    LET cust_rec.order_ids[3] = 2194
    LET obj = util.JSONObject.fromFGL(cust_rec)
    DISPLAY obj.toString()
END MAIN
```

util.JSONObject.parse
Parses a JSON string and creates a JSON object from it.

Syntax

```
util.JSONObject.parse(
   source STRING )
  RETURNING result util.JSONObject
```

1. *source* is a string value that contains JSON formatted data.

Usage

The `util.JSONObject.parse()` method scans the JSON source string passed as parameter and creates a JSON object from it.

The new created object must be assigned to a program variable defined with the `util.JSONObject` type.

The source string must follow the JSON format specification. It can contain multi-level structured data, but it must start with a curly brace.

The method raises error -8109 if the JSON source string is not properly formatted.

Example

```
IMPORT util
MAIN
    DEFINE js STRING
    DEFINE obj util.JSONObject
    LET js='{ "cust_num":2735, "cust_name":"McCarlson",
              "orderids":[234,3456,24656,34561] }'
    LET obj = util.JSONObject.parse( js )
    DISPLAY obj.get("cust_name")
END MAIN
```

util.JSONObject.get
Returns the value corresponding to the specified entry name.

Syntax

```
util.JSONObject.get(
   name STRING )
  RETURNING result result-type
```

1. *name* is the string identifying the JSON object property.
2. *result-type* can be a simple type, a `util.JSONObject` or a `util.JSONArray` object reference.

Usage

The `get()` method returns the value or JSON object corresponding to the element name passed as parameter.

If the element identified by the name is a simple value, the method returns a string. If the element is structured, the method returns a `util.JSONObject` instance and the returned object must be assigned to a program variable defined with the `util.JSONObject` type. If the element is a list of values, the method a `util.JSONArray` instance and the returned object must be assigned to a program variable defined with the `util.JSONArray` type.

A name/value pair can be set with the `put()` method.

Example

```
IMPORT util
MAIN
    DEFINE obj, sub util.JSONObject
    DEFINE jarr util.JSONArray
    DEFINE rec RECORD
                id INTEGER,
                name STRING
            END RECORD
    DEFINE arr DYNAMIC ARRAY OF INTEGER
    DEFINE x INT
    LET obj = util.JSONObject.create()
    -- Simple value
    CALL obj.put("simple", 234)
    LET x = obj.get("simple")
    -- Sub-element
    LET rec.id = 234
    LET rec.name = "Barton"
    CALL obj.put("record", rec)
    LET sub = obj.get("record")
    -- Array
    LET arr[1] = 234
    LET arr[2] = 2837
    CALL obj.put("array", arr)
    LET jarr = obj.get("array")
END MAIN
```

util.JSONObject.getLength
Returns the number of name-value pairs in the JSON object.

Syntax

```
util.JSONObject.getLength()
```

```
   RETURNING len INTEGER
```

Usage

The `getLength()` method returns the number of name-value pairs in the JSON object.

This method can be used in conjunction with the `name()` and `getType()` methods to read the entries of a JSON object.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    DEFINE i INTEGER
    LET obj = util.JSONObject.parse('{"id":123,"name":"Scott"}')
    FOR i=1 TO obj.getLength()
        DISPLAY i, ": ", obj.name(i), "=", obj.get(obj.name(i))
    END FOR
END MAIN
```

util.JSONObject.getType
Returns the type of a JSON object element.

Syntax

```
util.JSONObject.getType(
   name STRING )
  RETURNING type STRING
```

1. *name* is the name of the element.

Usage

The `getType()` method returns the JSON data type name corresponding to the entry identified by the name passed as parameter.

This method can be used in conjunction with the `name()` and `getLength()` methods to read the entries of a JSON object.

Possible values returned by this method are:

- `NUMBER`: A numeric value.
- `STRING`: A string value delimited by double quotes.
- `BOOLEAN`: A boolean value (true/false)
- `NULL`: A un-existing element.
- `OBJECT`: A structured object.
- `ARRAY`: An ordered list of elements.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    LET obj = util.JSONObject.create()
    CALL obj.put("id", 8723)
    DISPLAY obj.getType("id") -- NUMBER
    CALL obj.put("name", "Brando")
    DISPLAY obj.getType("name") -- STRING
```

```
    DISPLAY obj.getType("undef") -- NULL
END MAIN
```

util.JSONObject.has
Checks if the JSON object contains a specific entry name.

Syntax

```
util.JSONObject.has(
   name STRING )
  RETURNING result BOOLEAN
```

1. *name* is a string identifying a JSON object property.

Usage

The `has()` method determines if the JSON object holds a property identified by the name passed as parameter.

The method returns `TRUE` if the name/value pair exists in the JSON object.

A name/value pair can be set with the `put()` method.

util.JSONObject.name
Returns the name of a JSON object entry by position.

Syntax

```
util.JSONObject.name(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the index of the name-value pair in the JSON object.

Usage

The `name()` method returns the entry name in the JSON object at the given position.

The index corresponding to the first name-value pair is 1.

If no entry exists at the given index, the method returns `NULL`.

This method can be used in conjunction with the `getLength()` and `getType()` methods to read the entries of a JSON object.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    DEFINE i INTEGER
    LET obj = util.JSONObject.parse('{"id":123,"name":"Scott"}')
    FOR i=1 TO obj.getLength()
        DISPLAY i, ": ", obj.name(i)
    END FOR
END MAIN
```

util.JSONObject.put
Sets a name-value pair in the JSON object.

Syntax

```
util.JSONObject.put(
   name STRING,
   value value-type )
```

1. *name* is a string defining the entry name.
2. *value* is the value to be associated to the name.
3. *value-type* can be a simple string or numeric type, a `RECORD` or an `DYNAMIC ARRAY`.

Usage

The `put()` method adds a name-value pair to the JSON object.

The first parameter is the name of the element. The second parameter can be a simple string or numeric value, or a complex variable defined as `RECORD` or `DYNAMIC ARRAY`.

If the element exists, the existing value is replaced.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    DEFINE rec RECORD
                id INTEGER,
                name STRING
            END RECORD
    DEFINE arr DYNAMIC ARRAY OF INTEGER
    LET obj = util.JSONObject.create()
    CALL obj.put("simple", 234)
    LET rec.id = 234
    LET rec.name = "Barton"
    CALL obj.put("record", rec)
    LET arr[1] = 234
    LET arr[2] = 2837
    CALL obj.put("array", arr)
    DISPLAY obj.toString()
END MAIN
```

util.JSONObject.remove
Removes the specified element in the JSON object.

Syntax

```
util.JSONObject.remove(
   name STRING )
```

1. *name* is the string identifying the JSON object property.

Usage

The `remove()` method deletes a name-value pair identified by the name passed as parameter.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    LET obj = util.JSONObject.create()
    CALL obj.put("address", "5 Brando Street")
    CALL obj.remove("address")
    DISPLAY obj.get("address")
END MAIN
```

util.JSONObject.toFGL
Fills a record variable with the entries contained in the JSON object.

Syntax

```
util.JSONObject.toFGL(
  dest RECORD )
```

1. *dest* is the variable to be set with values of the JSON string.

Important: The *dest* is a `RECORD` variable is passed by reference to the method.

Usage

The `toFGL()` method fills the `RECORD` variable passed as parameter with the corresponding values defined in the JSON object.

The destination record must have the same structure as the JSON source data. For more details see JSON to Genero BDL conversion rules on page 1987.

Example

```
IMPORT util
MAIN
    DEFINE cust_rec RECORD
                cust_num INTEGER,
                cust_name VARCHAR(30),
                order_ids DYNAMIC ARRAY OF INTEGER
            END RECORD
    DEFINE js STRING
    DEFINE obj util.JSONObject
    LET js='{ "cust_num":2735, "cust_name":"McCarlson",
              "order_ids":[234,3456,24656,34561] }'
    LET obj = util.JSONObject.parse( js )
    CALL obj.toFGL( cust_rec )
    DISPLAY cust_rec.cust_name
    DISPLAY cust_rec.order_ids[4]
END MAIN
```

util.JSONObject.toString
Builds a JSON string from the values contained in the JSON object.

Syntax

```
util.JSONObject.toString()
  RETURNING result STRING
```

Usage

The `toString()` method produces a JSON formatted string from the name-value pairs contained in the JSON object.

Example

```
IMPORT util
MAIN
    DEFINE obj util.JSONObject
    LET obj = util.JSONObject.create()
    CALL obj.put("num", "75263")
    CALL obj.put("name", "Ferguson")
    CALL obj.put("address", "12 Marylon Street")
    DISPLAY obj.toString()
END MAIN
```

The util.JSONArray class

The `util.JSONArray` class provides methods to handle an array of values, following the JSON string syntax.

The `util.JSONArray` class is provided in the `util` C-Extension library; To use the `util.JSONArray` class, you must import the `util` package in your program:

```
IMPORT util
```

A `JSONArray` is an sequence of unnamed values. The format of a JSON array string is a list of values wrapped in square braces with commas between the values:

```
[123,546,"abc","def","xyz"]
```

A `JSONArray` object must be created before usage with one of the class methods like `util.JSONArray.create()`.

The `JSONArray` class provides methods for accessing, adding/replacing or deleting the array values by index with the `get()`, `put()` and `remove()` methods.

If the structure of the JSON array is not known at compile time, you can introspect the elements of the array with the `getLength()` and `getType()` methods.

util.JSONArray methods

Methods for the `util.JSONArray` class.

Table 444: Class methods

Name	Description
`util.JSONArray.create()` `  RETURNING` *`array`* `util.JSONArray`	Creates a new JSON array object.

`util.JSONArray.fromFGL(`
`  `*`source`* `DYNAMIC ARRAY )`
`  RETURNING` *`array`* `util.JSONArray`
	Creates a new JSON array object from a `DYNAMIC ARRAY`.

`util.JSONArray.parse(` `  `*`source`* `STRING )`	Parses a JSON string and creates a JSON array object from it.

Name	Description
`RETURNING result util.JSONArray`	

Table 445: Object methods

Name	Description
`util.JSONArray.get( index INTEGER ) RETURNING result result-type`	Returns the value of a JSON array element.
`util.JSONArray.getLength() RETURNING length INTEGER`	Returns the number of elements in the JSON array object.
`util.JSONArray.getType( index INTEGER ) RETURNING type STRING`	Returns the type of a JSON array element.
`util.JSONArray.put( index INTEGER, value value-type )`	Sets an element by position in the JSON array object.
`util.JSONArray.remove( index INTEGER )`	Removes the specified entry in the JSON array object.
`util.JSONArray.toFGL( dest DYNAMIC ARRAY )`	Fills a dynamic array variable with the elements contained in the JSON array object.
`util.JSONArray.toString() RETURNING result STRING`	Builds a JSON string from the elements contained in the JSON array object.

util.JSONArray.create
Creates a new JSON array object.

Syntax

```
util.JSONArray.create()
  RETURNING array util.JSONArray
```

Usage

The `util.JSONArray.create()` method create a new JSON array object.

The new created object must be assigned to a program variable defined with the `util.JSONArray` type.

Example

```
IMPORT util
MAIN
```

```
    DEFINE arr util.JSONArray
    LET arr = util.JSONArray.create()
    ...
END MAIN
```

util.JSONArray.fromFGL
Creates a new JSON array object from a `DYNAMIC ARRAY`.

Syntax

```
util.JSONArray.fromFGL(
   source DYNAMIC ARRAY )
  RETURNING array util.JSONArray
```

1. *source* is the `DYNAMIC ARRAY` variable used to create the JSON array object.

Usage

The `util.JSONArray.fromFGL()` method creates a new JSON array from the `DYNAMIC ARRAY` variable passed as parameter.

The new created object must be assigned to a program variable defined with the `util.JSONArray` type.

The members of the `DYNAMIC ARRAY` are converted to a list of name/value pairs in the JSON array object.

The dynamic array can be structured with a `RECORD` definition: the elements of the array will be converted individually.

For more details about FGL to JSON conversion, see Genero BDL to JSON conversion rules on page 1988.

Example

```
IMPORT util
MAIN
    DEFINE da DYNAMIC ARRAY OF INTEGER
    DEFINE arr util.JSONArray
    LET da[1] = 123
    LET da[2] = 972
    LET arr = util.JSONArray.fromFGL(da)
    DISPLAY arr.toString()
END MAIN
```

util.JSONArray.parse
Parses a JSON string and creates a JSON array object from it.

Syntax

```
util.JSONArray.parse(
   source STRING )
  RETURNING result util.JSONArray
```

1. *source* is a string value that contains JSON formatted data as a list of elements delimited by square braces.

Usage

The `util.JSONArray.parse()` method scans the JSON source string passed as parameter and creates a new JSON array object from it.

The new created object must be assigned to a program variable defined with the `util.JSONArray` type.

The source string must follow the JSON format specification. Elements of the list can contain multi-level structured data, but the string must follow the JSON array string syntax '`[ element, ... ]`' with square braces.

The method raises error -8109 if the JSON source string is not properly formatted.

Example

```
IMPORT util
MAIN
    DEFINE da DYNAMIC ARRAY OF INTEGER
    DEFINE arr util.JSONArray
    LET arr = util.JSONArray.parse("[1,2,3,4,5]")
    DISPLAY arr.toString()
END MAIN
```

util.JSONArray.get
Returns the value of a JSON array element.

Syntax

```
util.JSONArray.get(
   index INTEGER )
  RETURNING result result-type
```

1. *index* is the index of the element in the JSON array object.
2. *result-type* can be a simple type, a `util.JSONObject` or a `util.JSONArray` object reference.

Usage

The `get()` method returns the value or JSON object corresponding to the element at the given position.

The index corresponding to the first element is 1.

If no element exists at the given index, the method returns `NULL`.

If the element identified by the name is a simple value, the method returns a string. If the element is structured, the method returns a `util.JSONObject` instance and the returned object must be assigned to a program variable defined with the `util.JSONObject` type. If the element is a list of values, the method a `util.JSONArray` instance and the returned object must be assigned to a program variable defined with the `util.JSONArray` type.

A name/value pair can be set with the `put()` method.

Example

```
IMPORT util
MAIN
    DEFINE arr util.JSONArray
    LET arr = util.JSONArray.parse('[123,"abc",null]')
    DISPLAY arr.get(2) -- abc
END MAIN
```

util.JSONArray.getLength
Returns the number of elements in the JSON array object.

Syntax

```
util.JSONArray.getLength()
  RETURNING length INTEGER
```

Usage

The `getLength()` method returns the number of elements in the JSON array object.

This method can be used in conjunction with the `get()` and `getType()` method to read elements of a JSON array object.

Example

```
IMPORT util
MAIN
    DEFINE arr util.JSONArray
    DEFINE i INTEGER
    LET arr = util.JSONArray.parse('[123,8723,9232]')
    FOR i=1 TO arr.getLength()
        DISPLAY i, ": ", arr.get(i)
    END FOR
END MAIN
```

util.JSONArray.getType
Returns the type of a JSON array element.

Syntax

```
util.JSONArray.getType(
   index INTEGER )
  RETURNING type STRING
```

1. *index* is the ordinal position of the element.

Usage

The `getType()` method returns the data type name corresponding to the JSON array element at the given position.

The index corresponding to the first element is 1.

This method can be used in conjunction with the `getLength()` method to read the entries of a JSON array object.

Possible values returned by this method are:

- `NUMBER`: A numeric value.
- `STRING`: A string value delimited by double quotes.
- `BOOLEAN`: A boolean value (true/false)
- `NULL`: A un-existing element.
- `OBJECT`: A structured object.
- `ARRAY`: An ordered list of elements.

Example

```
IMPORT util
MAIN
    DEFINE arr util.JSONArray
    LET arr = util.JSONArray.parse('[123,"abc",null]')
    DISPLAY arr.getType(1) -- NUMBER
    DISPLAY arr.getType(2) -- STRING
    DISPLAY arr.getType(3) -- NULL
END MAIN
```

util.JSONArray.put
Sets an element by position in the JSON array object.

Syntax

```
util.JSONArray.put(
   index INTEGER,
   value value-type )
```

1. *index* is the index of the element in the JSON array object.
2. *value* is the value to be associated to the index.
3. *value-type* can be a simple string or numeric type, a `RECORD` or an `DYNAMIC ARRAY`.

Usage

The `put()` method sets an element value by position in the JSON array object.

The first parameter is the index of the element. The second parameter can be a simple string or numeric value, or a complex variable defined as `RECORD` or `DYNAMIC ARRAY`.

The index corresponding to the first element is 1.

If the element exists, the existing value is replaced.

Example

```
IMPORT util
MAIN
    DEFINE ja util.JSONArray
    DEFINE rec RECORD
                id INTEGER,
                name STRING
            END RECORD
    DEFINE arr DYNAMIC ARRAY OF INTEGER
    LET ja = util.JSONArray.create()
    CALL ja.put(1, 234)
    LET rec.id = 234
    LET rec.name = "Barton"
    CALL ja.put(2, rec)
    LET arr[1] = 234
    LET arr[2] = 2837
    CALL ja.put(3, arr)
    DISPLAY ja.toString()
END MAIN
```

util.JSONArray.remove
Removes the specified entry in the JSON array object.

Syntax

```
util.JSONArray.remove(
   index INTEGER )
```

1. *index* is the index of the element in the JSON array object.

Usage

The `remove()` method deletes an element in the JSON array object at the given position.

The index corresponding to the first element is 1.

If no element exists at the specified index, the method returns silently.

Example

```
IMPORT util
MAIN
    DEFINE arr util.JSONArray
    LET arr = util.JSONArray.parse('["aa","bb","cc"]')
    CALL arr.remove(2)
    DISPLAY arr.get(2) -- cc
END MAIN
```

util.JSONArray.toFGL
Fills a dynamic array variable with the elements contained in the JSON array object.

Syntax

```
util.JSONArray.toFGL(
   dest DYNAMIC ARRAY )
```

1. *dest* is the array variable to be set with values of the JSON string.

Important: The *dest* is a dynamic array passed by reference to the method.

Usage

The `toFGL()` method fills the `DYNAMIC ARRAY` passed as parameter with the corresponding values defined in the JSON array object.

The destination array must have the same structure as the JSON source data. For more details see JSON to Genero BDL conversion rules on page 1987.

Example

```
IMPORT util
MAIN
    DEFINE ja util.JSONArray
    DEFINE arr DYNAMIC ARRAY OF STRING
    LET ja = util.JSONArray.parse('["aa","bb","cc"]')
    CALL ja.toFGL(arr)
    DISPLAY arr[2] -- bb
END MAIN
```

util.JSONArray.toString
Builds a JSON string from the elements contained in the JSON array object.

Syntax

```
util.JSONArray.toString()
  RETURNING result STRING
```

Usage

The `toString()` method produces a JSON formatted string from the elements contained in the JSON array object.

Example

```
IMPORT util
MAIN
    DEFINE ja util.JSONArray
    LET ja = util.JSONArray.create()
    CALL ja.put(1,"aa")
    CALL ja.put(2,"bb")
    CALL ja.put(3,"cc")
    DISPLAY ja.toString() -- ["aa","bb","cc"]
END MAIN
```

JSON classes

Gives a basic introduction to JSON.

What is JSON?

JSON (JavaScript™ Object Notation) is a well known lightweight data-interchange format for JavaScript™.

A JSON string (or object) is a comma-separated list of name/value pairs, with a `:` colon separating the key and the value. The list of name/value pairs is enclosed in `{}` curly braces. The names are delimited by double-quotes. The value can be a single numeric value, a double-quotes string, an array, or a sub-element. Arrays are defined by a comma-separated list of values enclosed in `[]` square brackets. Sub-elements are defined inside {} curly braces and defineds name/value pairs.

For example:

```
{
   "cust_num":865234,
   "cust_name":"McCarlson",
   "order_ids":[234,3456,24656,34561],
   "address": {
          "street":"34, Sunset Bld",
          "city":"Los Angeles",
          "state":"CA"
       }
}
```

For more details, see http://www.json.org.

JSON utility classes

The `util` library provides a set of JSON classes to convert JSON documents to/from `RECORD` and `ARRAY` variables, and to manipulate JSON objects, if you need to handle JSON objects that do not map to a `RECORD` or `ARRAY`:

- The util.JSON class on page 1966
- The util.JSONObject class on page 1970
- The util.JSONArray class on page 1979

JSON to Genero BDL conversion rules

The JSON utility classes implement methods that can fill a Genero BDL `RECORD` or `DYNAMIC ARRAY`. This topic describes how JSON data is converted to Genero BDL data.

The destination record or array must have the same structure as the JSON source data.

Each JSON element is assigned to a record member by name, not by position. Elements in the JSON string can be at a different ordinal position as the corresponding members in the destination record.

Element name matching is case-insensitive. For example, if the Genero BDL record member is defined as `CustNo`, and the JSON data string contains the `"custno":999` name/value pair, the value will be assigned. However, since Genero BDL record member names are used as-is to write JSON data, is it strongly recommended to define the Genero BDL records with the exact names used in JSON data string.

Elements in the JSON string that do not match an Genero BDL record member are ignored; no error is thrown if there is no corresponding Genero BDL member.

Genero BDL record members that have no matching JSON element are initialized to `NULL`.

> **Important:** JSON specifications allow you to define element names with characters that cannot be used in Genero BDL identifiers. For example, a JSON element name can be `"customer.name"` or `"customer:name"`, however it is not possible to define a program variable with these same names. To work around this issue, define the record elements with underscores in place of unsupported characters, and before assigning the JSON string to the Genero BDL record, replace all element names by the corresponding record member names (using `base.StringBuffer.replace` to do the replacements).

The JSON value must match the data format of the destination member. If the value does not correspond to the type (for example, if the JSON value is a character string while the target record member is defined with a numeric type), the target member will be set to `NULL`.

JSON arrays delimited by square brackets are used to fill a program array of the destination record. The destination array should be a dynamic array. If the array is defined as static, the additional elements of the source JSON array will be discarded, while missing elements will be initialized to `NULL`.

The JSON source string must follow the JSON format specification. It can contain multilevel structured data. If the source string is not well formatted, the runtime system will throw error -8109.

Table 446: JSON to Genero BDL type conversion rules

Target Genero BDL type	Description
`BOOLEAN`	The JSON value should be `null`, `true` or `false`. If the JSON value is a number or a string, the language conversion rules from number/string to `BOOLEAN` apply.
`TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `SMALLFLOAT`, `FLOAT`, `DECIMAL`, `MONEY`	A JSON number can be assigned to any language numeric type. The limits of the target type cause potential overflows errors. On error the target variable will be initialized to `NULL`, the parser continues without an error.
`DATE`	The JSON value must be a string formatted as `"YYYY-MM-DD"`.
`DATETIME`	If the value is a JSON string, it must be formatted as `"YYYY-MM-DD hh:mm:ss.fffff"`,

Target Genero BDL type	Description
	or represent as an ISO 8601 formatted date-time, in UTC (with Z indicator) or with a timezone offset (+/-hh[:mm]). For example: `"2013-02-21T15:18:44.456Z"`, `"2013-02-21T20:18:44.456+02:00"`. If the value is a JSON number, it is interpreted as UNIX™ time (seconds since the Epoch 00:00:00 UTC, January 1, 1970). Note that the `YYYY-MM-DD hh:mm:ss.ffff` format is used to represent the local time. When exchanging date-time values in communications across different time zones, consider to convert date-time values to Coordinated Universal Time (UTC), by using the `util.Datetime` methods.
`INTERVAL`	The JSON valid must be a string formatted as "`YYYY-MM`" or "`DD hh:mm:ss.fffff`", according to interval class of the target variable.
`BYTE`	The JSON string value must be encoded in Base64. The Base64 encoding is described in [RFC4648].
`TEXT`, `CHAR`, `VARCHAR`, `STRING`	If the value is a number, the result string uses the locale specific decimal point. Any character in the Basic Multilingual Plane (U+0000 through U+FFFF) may be escaped: `\u` followed by exactly 4 hexadecimal digits (`[0-9a-fA-F]`). The hexadecimal digits encode the code point. Escaping of characters outside the Basic Multilingual Plane my be escaped by there UTF-16 surrogate pairs. Example: This is the representation of the G clef character (`U+1D11E`) "`\uD834\uDD1E`".

Genero BDL to JSON conversion rules

The JSON utility classes implement methods that can convert an Genero BDL `RECORD` or `DYNAMIC ARRAY` to a JSON data string. This topic describes how Genero BDL data is converted to JSON data.

To name the JSON elements, the names of the record members are used as defined in the program source. Since JSON is case-sensitive, make sure the names of the Genero BDL record members match exactly the names expected in the resulting JSON data string: `CustNo` will be different from `custNo`.

> **Important:** JSON specifications allow you to define element names with characters that cannot be used in Genero BDL identifiers. For example, a JSON element name can be `"customer.name"` or `"customer:name"`, however it is not possible to define a program variable with these same names. To work around this issue, define the record elements with underscores in place of unsupported characters, and before assigning the JSON string to the Genero BDL record, replace all element names by the corresponding record member names (using `base.StringBuffer.replace` to do the replacements).

If a record member is `NULL`, then the resulting JSON string or object omits that member.

Program array members in the record are converted to JSON arrays delimited by square brackets (`[]`).

Table 447: Genero BDL to JSON type conversion rules

Target Genero BDL type	Description
`BOOLEAN`	Will be serialized with the JSON values `true` or `false`.
`TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT`, `SMALLFLOAT`, `FLOAT`, `DECIMAL`, `MONEY`	Any numeric type will be serialized to this form: an optional minus sign (`-`), a sequence of digits (`0-9`), containing a optional decimal separator (`.`), followed by an optional exponent. The exponent has the form (e) followed by an optional minus sign and an sequence of digits. The representation of numeric values does not depend from the current locale. The decimal separator is always a dot (`.`). `MONEY` values will be represented like `DECIMAL` values: the currency symbol will be omitted.
`DATE`	The date value will be formatted as `"YYYY-MM-DD"` (with double quotes)
`DATETIME`	The date-time value will be formatted as `"YYYY-MM-DD hh:mm:ss.fffff"` (with double quotes), according to the date-time type definition. For example, a `DATETIME HOUR TO MINUTE` will produce `"hh:mm"` formatted values. Note that the `YYYY-MM-DD hh:mm:ss.ffff` format is used to represent the local time. When exchanging date-time values in communications across different time zones, consider to convert date-time values to Coordinated Universal Time (UTC), by using the `util.Datetime` methods.
`INTERVAL`	The interval value will be formatted as `"YYYY-MM"` or `"DD hh:mm:ss.fffff"` (with double quotes), according to the interval type definition.
`BYTE`	Will be serialized to a Base64 encoded double quoted string. The Base64 encoding is described in [RFC4648].

`TEXT`, `CHAR`, `VARCHAR`, `STRING`	Character string data will be serialized as a double quoted string with backslash escaping.
	List of characters requiring escaping:
	`\\ backslash U+005C`
	`\" quotation mark U+0022`
	`\b backspace U+0008`
	`\f form feed U+000C`
	`\n line feed U+000A`
	`\r carriage return U+000D`
	`\t tab U+0009`

Other	Any other type will be serialized as a double quoted (") string.

The os package

These topics cover the classes for the `os` package.

The os.Path class

The `os.Path` class provides functions to manipulate files and directories on the machine where the program executes.

This class is provided in the `util` C-Extension library; To use the `os.Path` extension, you must import the `os` package in your program:

```
IMPORT os
```

In order to manipulate files, this API give you access to low-level system functions. Pay attention to operating system specific conventions like path separators. Some functions are OS specific, like `rwx()` which works only on UNIX™ systems.

os.Path methods

Table 448: Class methods

Name	Description
`os.Path.atime( fname STRING) RETURNING result STRING`	Returns the time of the last file access.
`os.Path.baseName( filename STRING) RETURNING result STRING`	Returns the last element of a path.
`os.Path.chDir( newdir STRING) RETURNING result INTEGER`	Changes the current working directory.
`os.Path.chOwn( fname STRING, uid INTEGER, gui INTEGER ) RETURNING result INTEGER`	Changes the UNIX™ owner and group of a file.
`os.Path.chRwx( fname STRING, mode INTEGER ) RETURNING result INTEGER`	Changes the UNIX™ permissions of a file.
`os.Path.chVolume( new STRING) RETURNING result INTEGER`	Changes the current working volume.
`os.Path.copy( source STRING, dest STRING )`	Creates a new file by copying an existing file.

Name	Description
`RETURNING result INTEGER`	
`os.Path.delete( dname STRING) RETURNING result INTEGER`	Deletes a file or a directory.
`os.Path.dirClose( dirhandle INTEGER)`	Closes the directory referenced by the directory opened by `os.Path.diropen()`.
`os.Path.dirFMask( mask INTEGER)`	Defines a filter mask for `os.Path.dirOpen()`.
`os.Path.dirName( filename STRING) RETURNING result STRING`	Returns all components of a path excluding the last one.
`os.Path.dirNext( dirhandle INTEGER) RETURNING direntry STRING`	Reads the next entry in the directory opened with `os.Path.dirOpen()`.
`os.Path.dirOpen( dname STRING) RETURNING dirhandle INTEGER`	Opens a directory and returns an integer handle to this directory.
`os.Path.dirSort( criteria STRING, order INTEGER )`	Defines the sort criteria and sort order for `os.Path.dirOpen()`.
`os.Path.executable( fname STRING) RETURNING result INTEGER`	Checks if a file is executable.
`os.Path.exists( fname STRING) RETURNING result INTEGER`	Checks if a file exists.
`os.Path.extension( fname STRING) RETURNING result STRING`	Returns the file extension.
`os.Path.fullPath( path STRING) RETURNING result STRING`	Returns the canonical equivalent of a path.
`os.Path.gid(`	Returns the UNIX™ group id of a file.

Name	Description
fname STRING) RETURNING *id* INTEGER	
os.Path.homeDir() RETURNING *homedir* STRING	Returns the path to the HOME directory of the current user.
os.Path.isDirectory(*fname* STRING) RETURNING *result* BOOLEAN	Checks if a file is a directory.
os.Path.isFile(*fname* STRING) RETURNING *result* BOOLEAN	Checks if a file is a regular file.
os.Path.isHidden(*fname* STRING) RETURNING *result* BOOLEAN	Checks if a file is hidden.
os.Path.isLink(*fname* STRING) RETURNING *result* BOOLEAN	Checks if a file is UNIX™ symbolic link.
os.Path.isRoot(*path* STRING) RETURNING *result* BOOLEAN	Checks if a file path is a root path.
os.Path.join(*begin* STRING, *end* STRING) RETURNING *result* STRING	Joins two path segments adding the platform-dependent separator.
os.Path.makeTempName() RETURNING *result* STRING	Generates a temporary file name.
os.Path.mkDir(*dname* STRING) RETURNING *result* INTEGER	Creates a new directory.
os.Path.mtime(*fname* STRING) RETURNING *result* STRING	Returns the time of the last file modification.
os.Path.pathSeparator()	Returns the character used in environment variables to separate path elements.

Name	Description
`RETURNING result STRING`	
`os.Path.pathType( path STRING) RETURNING result STRING`	Checks if a path is a relative path or an absolute path.
`os.Path.pwd() RETURNING result STRING`	Returns the current working directory.
`os.Path.readable( fname STRING) RETURNING result INTEGER`	Checks if a file is readable.
`os.Path.rename( oldname STRING, newname STRING ) RETURNING result INTEGER`	Renames a file or a directory.
`os.Path.rootDir() RETURNING rootdir STRING`	Returns the root directory of the current working path.
`os.Path.rootName( filename STRING) RETURNING result STRING`	Returns the file path without the file extension of the last element of the file path.
`os.Path.rwx( fname STRING) RETURNING mode INTEGER`	Returns the UNIX™ file permissions of a file.
`os.Path.separator() RETURNING result STRING`	Returns the character used to separate path segments.
`os.Path.size( fname STRING) RETURNING result INTEGER`	Returns the size of a file.
`os.Path.type( fname STRING) RETURNING result STRING`	Returns the file type as a string.
`os.Path.uid( fname STRING) RETURNING id INTEGER`	Returns the UNIX™ user id of a file.
`os.Path.volumes()`	Returns the available volumes.

Name	Description
`RETURNING volumes STRING`	
`os.Path.writable( fname STRING) RETURNING result INTEGER`	Checks if a file is writable.

os.Path.atime
Returns the time of the last file access.

Syntax

```
os.Path.atime(
   fname STRING)
  RETURNING result STRING
```

1. *fname* is the name of the file.

Usage

The function returns a string containing the last access time for the specified file, in the standard format 'YYYY-MM-DD HH:MM:SS'.

If the function fails, it returns NULL.

os.Path.baseName
Returns the last element of a path.

Syntax

```
os.Path.baseName(
   filename STRING)
  RETURNING result STRING
```

1. *filename* is the name of the file.

Usage

This method extracts the last component of a path provided as argument.

For example, if you pass "/root/dir1/file.ext" as the parameter, it will return "file.ext".

See Example 1: Extracting the parts of a file name on page 2007 for more examples.

os.Path.copy
Creates a new file by copying an existing file.

Syntax

```
os.Path.copy(
   source STRING,
   dest STRING )
  RETURNING result INTEGER
```

1. *source* is the name of the file to copy.
2. *dest* is the destination name of the copied file.

Usage

The function returns TRUE if the file has been successfully copied, FALSE otherwise.

os.Path.chDir
Changes the current working directory.

Syntax

```
os.Path.chDir(
   newdir STRING)
  RETURNING result INTEGER
```

1. *newdir* is the directory to select.

Usage

Use this function to change the current working directory.

The function returns TRUE if the current directory could be successfully selected, FALSE otherwise.

os.Path.chRwx
Changes the UNIX™ permissions of a file.

Syntax

```
os.Path.chRwx(
   fname STRING,
   mode INTEGER )
  RETURNING result INTEGER
```

1. *fname* is the name of the file.
2. *mode* is the UNIX™ permission combination in decimal (not octal!).

Usage

This method can only be used on UNIX™!

Function returns TRUE on success, FALSE otherwise.

The *mode* must be a decimal value which is the combination of read, write and execution bits for the user, group and other part of the UNIX™ file permission. Make sure to pass the *mode* as the decimal version of permissions, not as octal (the chrwx UNIX™ command takes an octal value as parameter). For example, to set `-rw-r--r--` permissions, you must pass (((4+2) *64) + (4 * 8) + 4) = 420 to this method.

os.Path.chOwn
Changes the UNIX™ owner and group of a file.

Syntax

```
os.Path.chOwn(
   fname STRING,
   uid INTEGER,
   gui INTEGER )
  RETURNING result INTEGER
```

1. *fname* is the name of the file.
2. *uid* is the user id.
3. *gui* is the group id.

Usage

This method can only be used on UNIX™!

Function returns TRUE on success, FALSE otherwise.

os.Path.chVolume
Changes the current working volume.

Syntax

```
os.Path.chVolume(
   new STRING)
  RETURNING result INTEGER
```

1. *new* is the volume to select as the new current working volume.

Usage

To change the current volume to C:

```
LET result = os.Path.chVolume("C:\\")
```

The function returns TRUE if the current working volume could be successfully changed, FALSE otherwise.

os.Path.delete
Deletes a file or a directory.

Syntax

```
os.Path.delete(
   dname STRING)
  RETURNING result INTEGER
```

1. *dname* is the name of the file or directory to delete.

Usage

A directory can only be deleted if it is empty.

The function TRUE if the file has been successfully deleted, FALSE otherwise.

os.Path.dirClose
Closes the directory referenced by the directory opened by `os.Path.diropen()`.

Syntax

```
os.Path.dirClose(
   dirhandle INTEGER)
```

1. *dirhandle* is the directory handle of the directory to close.

Usage

This function closes the directory search handle opened with `os.Path.dirOpen()`.

os.Path.dirFMask
Defines a filter mask for `os.Path.dirOpen()`.

Syntax

```
os.Path.dirFMask(
   mask INTEGER)
```

1. *mask* defines the filter mask.

Usage

When you call this function, you define the filter mask for any subsequent `os.Path.dirOpen()` call.

By default, all kinds of directory entries are selected by the `dirOpen()` function. You can restrict the number of entries by using a filter mask.

The parameter of the `os.Path.dirFMask()` function must be a combination of the following bits:

- `0x01` = Exclude hidden files (.*)
- `0x02` = Exclude directories
- `0x04` = Exclude symbolic links
- `0x08` = Exclude regular files

For example, to retrieve only regular files, you must call:

```
CALL os.Path.dirFMask( 1 + 2 + 4 )
```

os.Path.dirName
Returns all components of a path excluding the last one.

Syntax

```
os.Path.dirName(
   filename STRING)
  RETURNING result STRING
```

1. *filename* is the name of the file.
2. *result* contains all the elements of the path excluding the last one.

Usage

This method removes the last component of a path provided as argument.

For example, if you pass `"/root/dir1/file.ext"` as the parameter, it will return `"/root/dir1"`.

See Example 1: Extracting the parts of a file name on page 2007 for more examples.

os.Path.dirNext
Reads the next entry in the directory opened with `os.Path.dirOpen()`.

Syntax

```
os.Path.dirNext(
   dirhandle INTEGER)
  RETURNING direntry STRING
```

1. *dirhandle* is the directory handle of the directory to read.
2. *direntry* is the name of the entry read or `NULL` if all entries have been read.

Usage

This function returns the next entry of the directory opened with `os.Path.dirOpen()`.

os.Path.dirOpen
Opens a directory and returns an integer handle to this directory.

Syntax

```
os.Path.dirOpen(
   dname STRING)
  RETURNING dirhandle INTEGER
```

1. *dname* is the name of the directory.
2. *dirhandle* is the directory handle.

Usage

This function creates a handle to scan the elements of a directory.

The function returns a value of 0 if it fails to open the directory.

Before calling the `dirOpen()` method, you can define a filter with `os.Path.dirFMask()`, and a sort order with `os.Path.dirSort()`.

os.Path.dirSort
Defines the sort criteria and sort order for `os.Path.dirOpen()`.

Syntax

```
os.Path.dirSort(
   criteria STRING,
   order INTEGER )
```

1. *criteria* is the sort criteria.
2. *order* defines ascending (1) or descending (-1) order.

Usage

When you call this function, you define the sort criteria and sort order for any subsequent `os.Path.dirOpen()` call.

The *criteria* parameter must be one of the following strings:

- `"undefined"` = No sort. This is the default. Entries are read as returned by the OS functions.
- `"name"` = Sort by file name.
- `"size"` = Sort by file size.
- `"type"` = Sort by file type (directory, link, regular file).
- `"atime"` = Sort by access time.
- `"mtime"` = Sort by modification time.
- `"extension"` = Sort by file extension.

When sorting by name, directory entries will be ordered according to the current locale.

When sorting by any criteria other than the file name, entries having the same value for the given criteria are ordered by name following the value of the *order* parameter.

os.Path.executable
Checks if a file is executable.

Syntax

```
os.Path.executable(
   fname STRING)
  RETURNING result INTEGER
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file is executable, FALSE otherwise.

os.Path.exists
Checks if a file exists.

Syntax

```
os.Path.exists(
   fname STRING)
  RETURNING result INTEGER
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file exists, FALSE otherwise.

os.Path.extension
Returns the file extension.

Syntax

```
os.Path.extension(
   fname STRING)
  RETURNING result  STRING
```

1. *fname* is the file name.

Usage

The function returns the string following the last dot found in *fname*.

If *fname* does not have an extension, the function returns NULL.

os.Path.fullPath
Returns the canonical equivalent of a path.

Syntax

```
os.Path.fullPath(
   path STRING)
  RETURNING result STRING
```

1. *path* is the path to complete.

Usage

The `os.path.fullPath()` class method takes a path as parameter and resolves extra path separator characters (`/` on UNIX™, `\` on Windows™), as well as references to current (`.`) and parent directory (`..`). The result is called a canonical path.

On UNIX™, symbolic links are not followed. Use the `os.Path.isLink()` method to identify symbolic links.

Example

```
DISPLAY os.Path.fullPath("/home/usr//scott/tmp/../images")
```

Resolves to:

```
/home/usr/scott/images
```

os.Path.gid
Returns the UNIX™ group id of a file.

Syntax

```
os.Path.gid(
   fname STRING)
  RETURNING id INTEGER
```

1. *fname* is the name of the file.
2. *id* is the group id.

Usage

This method can only be used on UNIX™!

Function returns -1 if it fails to get the user id.

os.Path.homeDir
Returns the path to the HOME directory of the current user.

Syntax

```
os.Path.homeDir()
  RETURNING homedir STRING
```

1. *homedir* Path to the HOME directory of the user.

os.Path.isDirectory
Checks if a file is a directory.

Syntax

```
os.Path.isDirectory(
   fname STRING)
  RETURNING result BOOLEAN
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file is a directory, FALSE otherwise.

os.Path.isFile
Checks if a file is a regular file.

Syntax

```
os.Path.isFile(
   fname STRING)
  RETURNING result BOOLEAN
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file is a regular file, FALSE otherwise.

os.Path.isHidden
Checks if a file is hidden.

Syntax

```
os.Path.isHidden(
   fname STRING)
  RETURNING result BOOLEAN
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file is hidden, FALSE otherwise.

For example, on UNIX™, files starting with a dot in the file name are considered as hidden when using the ls command.

os.Path.isLink
Checks if a file is UNIX™ symbolic link.

Syntax

```
os.Path.isLink(
   fname STRING)
  RETURNING result BOOLEAN
```

1. *fname* is the file name.

Usage

The function returns TRUE if the files is a symbolic link, FALSE otherwise.

This method can only be used on UNIX™!

os.Path.isRoot
Checks if a file path is a root path.

Syntax

```
os.Path.isRoot(
   path STRING)
  RETURNING result BOOLEAN
```

x

1. *path* is the path to check.

Usage

The function returns TRUE if the path is a root path, FALSE otherwise.

On UNIX™ the root path is '/'.

On Windows™ the root path matches "[a-zA-Z]:\".

os.Path.join
Joins two path segments adding the platform-dependent separator.

Syntax

```
os.Path.join(
   begin STRING,
   end STRING )
  RETURNING result STRING
```

1. *begin* is the beginning path segment.
2. *end* is the ending path segment.

Usage

Use this method to construct a path with no system-specific code to use the correct path separator:

```
LET path = os.Path.join(os.Path.homedir(), name)
```

This method returns the ending path segment if it is an absolute path.

If one of the arguments is NULL, the function returns NULL.

os.Path.makeTempName
Generates a temporary file name.

Syntax

```
os.Path.makeTempName()
  RETURNING result STRING
```

Usage

This method creates a new temporary file path, with the unique file name, in the temporary directory of the process.

The temporary directory is found according to the type of platform, see DBTEMP environment variable for more details.

> **Note:** If a file is created with the given path, it must be deleted explicitely.

os.Path.mtime
Returns the time of the last file modification.

Syntax

```
os.Path.mtime(
   fname STRING)
  RETURNING result STRING
```

1. *fname* is the name of the file.
2. *result* is the last modification time.

Usage

The function returns a string containing the last modification time for the specified file, in the standard format 'YYYY-MM-DD HH:MM:SS'.

If the function fails, it returns NULL.

os.Path.mkDir
Creates a new directory.

Syntax

```
os.Path.mkDir(
   dname STRING)
  RETURNING result INTEGER
```

1. *dname* is the name of the directory to create.

Usage

The function returns TRUE if the directory has been successfully created, FALSE otherwise.

os.Path.pathSeparator
Returns the character used in environment variables to separate path elements.

Syntax

```
os.Path.pathSeparator()
  RETURNING result STRING
```

Usage

You typically use this method to build a path from two components.

On UNIX™, the path separator is ':'.

On Windows™, the path separator is ';'.

os.Path.pathType
Checks if a path is a relative path or an absolute path.

Syntax

```
os.Path.pathType(
   path STRING)
  RETURNING result STRING
```

1. *path* is the path to check.

Usage

The function returns "absolute" if the path is an absolute path, or "relative" if the path is a relative path.

If the path is NULL, the function returns NULL.

os.Path.pwd
Returns the current working directory.

Syntax

```
os.Path.pwd()
  RETURNING result STRING
```

Usage

This function returns the path of the current working directory.

On a mobile device, this front call returns the current application working directory:

- On Android™, it returns the directory where the program executes.
- On iOS, it returns the "Documents" directory under the application directory.

os.Path.readable
Checks if a file is readable.

Syntax

```
os.Path.readable(
   fname STRING)
  RETURNING result INTEGER
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file is readable, FALSE otherwise.

os.Path.rename
Renames a file or a directory.

Syntax

```
os.Path.rename(
   oldname STRING,
   newname STRING )
  RETURNING result INTEGER
```

1. *oldname* is the current name of the file or directory to be renamed.
2. *newname* is the new name to assign to the file or directory.

Usage

The function returns TRUE if the file or directory has been successfully renamed, FALSE otherwise.

On UNIX™ platforms, you can rename/move files and directories.

On Microsoft™ Windows™ platforms only files can be renamed/moved. However, on Windows™ you can move files across disks and directories.

os.Path.separator
Returns the character used to separate path segments.

Syntax

```
os.Path.separator()
```

```
   RETURNING result STRING
```

Usage

Use this method to build a path from two components.

On UNIX™, the directory separator is '/'.

On Windows™, the directory separator is '\'.

os.Path.size
Returns the size of a file.

Syntax

```
os.Path.size(
   fname STRING)
  RETURNING result INTEGER
```

1. *fname* is the file name.

Usage

The function returns the size in bytes for the specified file.

os.Path.rootDir
Returns the root directory of the current working path.

Syntax

```
os.Path.rootDir()
  RETURNING rootdir STRING
```

1. *rootdir* is the root directory of the current working path.

Usage

On UNIX™, it always returns "/".

On Windows™ it returns the current working drive as "[a-zA-Z]:\"

os.Path.rootName
Returns the file path without the file extension of the last element of the file path.

Syntax

```
os.Path.rootName(
   filename STRING)
  RETURNING result STRING
```

1. *filename* is the file path.

Usage

This method removes the file extension from the path provided as parameter.

For example, if you pass "/root/dir1/file.ext" as the parameter it will return "/root/dir1/file".

See Example 1: Extracting the parts of a file name on page 2007 for more examples.

os.Path.rwx
Returns the UNIX™ file permissions of a file.

Syntax

```
os.Path.rwx(
   fname STRING)
  RETURNING mode INTEGER
```

1. *fname* is the name of the file.
2. *mode* is the combination of permissions for user, group and other.

Usage

This method can only be used on UNIX™!

Function returns -1 if it fails to get the permissions.

The *mode* is returned as a decimal value which is the combination of read, write and execution bits for the user, group and other part of the UNIX™ file permission. For example, if a file has the `-rwxr-xr-x` permissions, the method returns ((4+2+1) * 64 + (4+1) * 8) + (4+1)) = 493.

os.Path.type
Returns the file type as a string.

Syntax

```
os.Path.type(
   fname STRING)
  RETURNING result STRING
```

1. *fname* is the file name.

Usage

On UNIX™, this method follows symbolic links. Use the `os.Path.islink()` method to identify symbolic links.

The possible values returned by this method are:

1. `file`: the file is a regular file
2. `directory`: the file is a directory
3. `socket`: the file is a socket
4. `fifo`: the file is a fifo
5. `block`: the file is a block device
6. `char`: the file is a character device

os.Path.uid
Returns the UNIX™ user id of a file.

Syntax

```
os.Path.uid(
   fname STRING)
  RETURNING id INTEGER
```

1. *fname* is the name of the file.
2. *id* is the user id.

Usage

This method can only be used on UNIX™!

Function returns -1 if it fails to get the user id.

os.Path.volumes
Returns the available volumes.

Syntax

```
os.Path.volumes()
  RETURNING volumes STRING
```

1. *volumes* contains the list of all available volumes separated by "|".

Usage

To display the list of available volumes, a volume is identified by its letter, followed by a colon and a backslash (:\).

```
DISPLAY os.Path.volumes()
```

Output example:

```
C:\|E:\|F:\
```

os.Path.writable
Checks if a file is writable.

Syntax

```
os.Path.writable(
   fname STRING)
  RETURNING result INTEGER
```

1. *fname* is the file name.

Usage

The function returns TRUE if the file is writable, FALSE otherwise.

Examples
Example 1: Extracting the parts of a file name

This program uses the file functions to extract the directory name, the base name, the root name, and the file extension:

```
IMPORT os
MAIN
  DISPLAY "Dir name   = ", os.Path.dirName(arg_val(1))
  DISPLAY "Base name  = ", os.Path.baseName(arg_val(1))
  DISPLAY "Root name  = ", os.Path.rootName(arg_val(1))
  DISPLAY "Extension  = ", os.Path.extension(arg_val(1))
END MAIN
```

Example results:

Table 449: Example results

Path	os.Path.dirname	os.Path.basename	os.Path.rootname	os.Path.extension
.	.	.		NULL
..	.	..	.	NULL
/	/	/	/	NULL
/usr/lib	/usr	lib	/usr/lib	NULL
/usr/	/	usr	/usr/	NULL
usr	.	usr	usr	NULL
file.xx	.	file.xx	file	xx
/tmp.yy/ file.xx	/tmp.yy	file.xx	/tmp.yy/file	xx
/tmp.yy/ file.xx.yy	/tmp.yy	file.xx.yy	/tmp.yy/ file.xx	yy
/tmp.yy/	/	tmp.yy	/tmp.yy/	NULL
/tmp.yy/.	/tmp.yy	.	/tmp.yy/	NULL

These examples use UNIX™ file names. On Windows™ the result would be different, as the file name separator is a backslash (\).

Example 2: Browsing directories

This program takes a directory path as an argument and scans the content recursively:

```
IMPORT os

MAIN
  CALL showDir(arg_val(1))
END MAIN

FUNCTION showDir(path)
  DEFINE path STRING
  DEFINE child STRING
  DEFINE h INTEGER

  IF NOT os.Path.exists(path) THEN
     RETURN
```

```
  END IF

  IF NOT os.Path.isDirectory(path) THEN
     DISPLAY " ", os.Path.baseName(path)
     RETURN
  END IF

  DISPLAY "[", path, "]"
  CALL os.Path.dirSort("name", 1)
  LET h = os.Path.dirOpen(path)
  WHILE h > 0
     LET child = os.Path.dirNext(h)
     IF child IS NULL THEN EXIT WHILE END IF
     IF child == "." OR child == ".." THEN CONTINUE WHILE END IF
     CALL showDir( os.Path.join( path, child ) )
  END WHILE

  CALL os.Path.dirClose(h)

END FUNCTION
```

The com package

The Genero Web Services `com` package provides classes and methods that allow you to perform tasks associated with creating Services and Clients, and managing the services.

Use the `IMPORT` statement at the top of the module using this library:

```
IMPORT com
```

Web services classes

The Web services classes manage Web Services servers.

- The WebService class on page 2009
- The WebOperation class on page 2018
- The WebServiceEngine class on page 2025
- The HTTPServiceRequest class on page 2036

The WebService class

The `com.WebService` class provides an interface to create and manage Genero Web Services.

The `com.WebServices` class is used to implement a Web Service on the server.

> **Important:** This Web Services class is not supported on GMI mobile devices.

com.WebServices methods

Methods for the `com.WebService` class.

Table 450: Class methods

Name	Description

`com.WebService.CreateStatefulWebService(`
`  name STRING,`
`  namespace STRING,`
`  state state-type )`
`  RETURNING result com.WebService`
	Creates a new object to implement a stateful Web Service.

`com.WebService.CreateWebService(` `  name STRING,`	Creates a new object to implement a Web Service.

Name	Description
namespace STRING) RETURNING *result* com.WebService	

Table 451: Object methods

Name	Description
createFault(*fault fault-type*, *encoded* BOOLEAN)	Creates a new object to implement a Web Service.
createHeader(*header header-type*, *encoded* BOOLEAN)	Defines the header for the Web Service object.
generateWSDL(*location* STRING) RETURNING *result* xml.DomDocument	Creates a `xml.DomDocument` object with the WSDL corresponding to the Web Service object.
publishOperation(*operation* com.WebOperation, *role* STRING)	Publishes a Web Operation.
registerInputHTTPVariable(*http-in http-in-type*)	Registers the record variable for HTTP input.
registerInputRequestHandler(*funcname* STRING)	Registers the function to be executed on incoming SOAP requests.
registerOutputHTTPVariable(*http-out http-out-type*)	Registers the record variable for HTTP output.
registerOutputRequestHandler(*funcname* STRING)	Registers the function to be executed just before the SOAP response is forwarded to the client.
registerWSDLHandler(*funcname* STRING)	Registers the function to be executed when a WSDL is generated.
saveWSDL(*location* STRING) RETURNING *result* INTEGER	Writes to a file the WSDL corresponding to the Web Service object.
setComment(	Defines the comment for the Web Service object.

Name	Description
comment STRING)	
setFeature(*name* STRING, *value* STRING)	Defines a feature for the current Web Service object.

com.WebService.createFault

Creates a new object to implement a Web Service.

Syntax

```
createFault(
   fault fault-type,
   encoded BOOLEAN )
```

1. *fault* defines the header for the Web Service object.
2. *fault-type* is a simple data type, a RECORD or an ARRAY.
3. *encoded* specifies the encoding mechanism.

Usage

The `createFault()` method creates a global fault for this Web Service object.

The *fault* parameter can be of any type and defines the SOAP fault in a SOAP response. In case of SOAP fault, the client for this Web Service will receive a variable with the same structure.

When *encoded* is TRUE, the SOAP Section 5 encoding mechanism is used, when FALSE, the XML Schema mechanism is used.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.createHeader

Defines the header for the Web Service object.

Syntax

```
createHeader(
   header header-type,
   encoded BOOLEAN )
```

1. *header* defines the header for the Web Service object.
2. *header-type* is a simple data type or a RECORD structure, or an ARRAY.
3. *encoded* specifies the encoding mechanism.

Usage

The `createHeader()` method creates a global header for the current Web Service object.

The Web Service header is defined by the first parameter. This will define SOAP headers exchanged by the client and server.

When *encoded* is TRUE, the SOAP Section 5 encoding mechanism will be used. If FALSE, the XML Schema mechanism will be used.

Important: Since Web Services headers are generally in Document Style, we recommend to set the *encoded* parameter to FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.CreateWebService

Creates a new object to implement a Web Service.

Syntax

```
com.WebService.CreateWebService(
   name STRING,
   namespace STRING )
  RETURNING result com.WebService
```

1. *name* is the Web Service identifier.
2. *namespace* is the name space for the Web Service name.

Usage

The com.WebService.CreateWebService() class method creates a new com.WebService object implementing a Web Service.

The *name* and *namespace* must uniquely identify the Web Service across the entire application, when multiple Web Service programs run on the same server. In theory, *namespace+name* must be unique on the internet.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.CreateStatefulWebService

Creates a new object to implement a stateful Web Service.

Syntax

```
com.WebService.CreateStatefulWebService(
   name STRING,
   namespace STRING,
   state state-type )
  RETURNING result com.WebService
```

1. *name* is the Web Service identifier.
2. *namespace* is the name space for the Web Service name.
3. *state* is used to identify the state between the client and server.
4. *state-type* is a regular data type or RECORD structure.

Usage

The com.WebService.CreateStatefulWebService() class method creates a new com.WebService object implementing a Web Service that is stateful.

The *name* and *namespace* must uniquely identify the Web Service across the entire application, when multiple Web Service programs run on the same server. In theory, *namespace+name* must be unique on the internet.

The *state* variable used to identify the state between the client and the server:

- For a WS-Addressing stateful service, the *state* variable must be a `RECORD` with the following structure, with the `W3CEndpointReference` variable attribute:

```
RECORD ATTRIBUTES(W3CEndpointReference)
  address STRING, -- The location of the Web Service (for ex: URL)
  ref RECORD
    ... (other members defining the state)
  END RECORD
END RECORD
```

- For a stateful service based on HTTP cookies, the *state* variable must be a simple variable defined with a basic data type.

It is up to the programmer to manage the *state* variable and to restore the service state from a database.

When creating a stateful Web Service, all published Web Operations require a session in the client request excepted those defined as '`initiateSession`'.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.generateWSDL

Creates a `xml.DomDocument` object with the WSDL corresponding to the Web Service object.

Syntax

```
generateWSDL(
   location STRING )
  RETURNING result xml.DomDocument
```

1. *location* is the URL where the Web Service will be deployed.

Usage

The `generateWSDL()` method creates a new `xml.DomDocument` object containing the WSDL data of the Web Service object.

The URL where the Web Service will be deployed must be specified.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.publishOperation

Publishes a Web Operation.

Syntax

```
publishOperation(
   operation com.WebOperation,
   role STRING )
```

1. *operation* is the Web Operation object.
2. *role* identifies uniquely the Web Operation.

Usage

The `publishOperation()` method publishes the Web Operation specified by the `com.WebOperation` object passed as parameter.

The *role* identifies the operation, if several operations have the same name, by setting the SOAPAction HTTP header. Usually this parameter is set to NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.registerInputHTTPVariable

Registers the record variable for HTTP input.

Syntax

```
registerInputHTTPVariable(
   http-in http-in-type )
```

1. *http-in* is the HTTP input record variable.
2. *http-in-type* must be a RECORD with following structure:

```
RECORD
  verb  STRING,
  url   STRING,
  headers DYNAMIC ARRAY OF RECORD
    name  STRING,
    value  STRING
  END RECORD
END RECORD
```

Usage

The registerInputHTTPVariable() method registers a program variable with a specific structure, that will be filled with the HTTP request headers when a Web Operation arrives.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.registerInputRequestHandler

Registers the function to be executed on incoming SOAP requests.

Syntax

```
registerInputRequestHandler(
   funcname STRING )
```

1. *funcname* is the name of a program function.

Usage

The registerInputRequestHandler() method registers a function to be called when an incoming SOAP request is received and before the SOAP engine has processed it.

The callback function must be defined with a unique parameter of type xml.DomDocument, and must return the reference to this object, or NULL:

```
FUNCTION myRequestInputHandler( in )
    DEFINE in xml.DomDocument
    ...
    RETURN in
END FUNCTION
```

The input callback function typically modifies the content of the SOAP input request DOM document object passed as parameter.

When returning `NULL` from the input callback function, the output callback function will be called with the default SOAP fault node, which can then be modified.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.registerOutputHTTPVariable

Registers the record variable for HTTP output.

Syntax

```
registerOutputHTTPVariable(
   http-out http-out-type )
```

1. *http-out* is the HTTP output record variable.
2. *http-out-type* is a `RECORD` with the following structure:

```
RECORD
  code INTEGER,
  desc STRING,
  headers DYNAMIC ARRAY OF RECORD
    name  STRING,
    value  STRING
  END RECORD
END RECORD
```

Usage

The `registerOutputHTTPVariable()` method registers a program variable with a specific structure, that will be used to fill the HTTP response headers when a Web Operation is completed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.registerOutputRequestHandler

Registers the function to be executed just before the SOAP response is forwarded to the client.

Syntax

```
registerOutputRequestHandler(
   funcname STRING )
```

1. *funcname* is the name of a program function.

Usage

The `registerOutputRequestHandler()` method registers a function to be called just after the SOAP engine has processed the request and before the SOAP response is forwarded to the client.

The output callback function must be defined with a unique parameter of type `xml.DomDocument`, and must return the reference to this object:

```
FUNCTION myRequestOutputHandler( out )
    DEFINE out xml.DomDocument
    ...
```

```
    RETURN out
END FUNCTION
```

The output callback function typically modifies the content of the SOAP output request DOM document object passed as parameter.

If `NULL` was returned from the input callback function, the output callback function will be called with the default SOAP fault node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.registerWSDLHandler

Registers the function to be executed when a WSDL is generated.

Syntax

```
registerWSDLHandler(
   funcname STRING )
```

1. *funcname* is the name of a program function.

Usage

The `registerWSDLHandler()` method registers a function to be called when the WSDL of the current Web Service object is generated.

The callback function must be defined with a unique parameter of type `xml.DomDocument`, and must return the reference to this object:

```
FUNCTION myWSDLHandler( wsdl )
    DEFINE wsdl xml.DomDocument
    ...
    RETURN wsdl
END FUNCTION
```

The callback function typically modifies the content of the WSDL DOM document object passed as parameter.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.saveWSDL

Writes to a file the WSDL corresponding to the Web Service object.

Syntax

```
saveWSDL(
   location STRING )
  RETURNING result INTEGER
```

1. *location* is the URL where the Web Service will be deployed.

Usage

The `saveWSDL()` method writes the WSDL data corresponding to the Web Service object.

The URL where the Web Service will be deployed must be specified.

The name of the file will be the name of the Web Service defined by the *name* parameter passed to the `createWebService()` or `createStatefulWebService()` methods.

The method returns 0 if the file was saved, -1 in case of error.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.setComment

Defines the comment for the Web Service object.

Syntax

```
setComment(
   comment STRING )
```

1. *comment* is the description of the Web Service.

Usage

The `setComment()` method defines the comment associated to a `com.WebService` object.

The comment will be used when generating the WSDL file, as defined by the standard.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebService.setFeature

Defines a feature for the current Web Service object.

Syntax

```
setFeature(
   name STRING,
   value STRING )
```

1. *name* is the name of the Web Service feature.
2. *value* is the value of the feature.

Usage

The `setFeature()` method defines a feature for the current Web Service object by specifying a feature name and a value.

The features names are predefined. The second parameter must a a valid value for the specified feature.

Table 452: Support Web Service features for the setFeature() method

Name	Description
`Soap1.1`	Defines whether the Web Service supports the SOAP 1.1 protocol. Default value is FALSE.
`Soap1.2`	Defines whether the Web Service supports the SOAP 1.2 protocol. Default value is FALSE.
`WS-Addressing1.0`	Defines whether the Web Service supports WS-Addressing 1.0. Valid values include:

Name	Description
	• TRUE - The service supports WS-Addressing 1.0 and accepts requests without WS-Addressing. • REQUIRED - The service supports WS-Addressing 1.0 and accepts only requests with WS-Addressing. • FALSE - WS-Addressing 1.0 is disabled (Default).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The WebOperation class
The com.WebOperation class provides an interface to create and manage the operations of a Genero Web Service.

> **Important:** This Web Services class is not supported on GMI mobile devices.

The Web Operation can be created as RPC Style or Document Style. Both RPC/Literal and Doc/Literal Styles are WS-I compliant (standards set by the Web Services Interoperability organization).

RPC Style Service (RPC/Literal) is generally used to execute a function, such as a service that returns a stock option. Document Style Service (Doc/Literal) is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document; this is the MS. Net default. The input or output RECORD cannot have XMLNamespace attributes set on their members.

Calling the appropriate function to create the desired style is the only difference in your Genero code that creates the service. The remainder of the code that describes the service is the same, regardless of whether you want to create an RPC or Document style of service.

Do not use the setInputEncoded() and setOutputEncoded() methods, as they will specify the RPC/Encoded Style, which is not recommended (see Choosing a Web Service Style).

Since release 2.0 GWS allows you to create RPC Style and Document Style operations in the same Web Service. However, we do not recommend this, as it is not WS-I compliant.

com.WebOperation methods
Methods for the com.WebOperation class.

Table 453: Class methods

Name	Description
com.WebOperation.CreateDOCStyle(*function* STRING, *operation* STRING, *input* RECORD, *output* RECORD) RETURNING *result* com.WebOperation	Creates a new Web Operation object with Document style.
com.WebOperation.CreateOneWayDOCStyle(*function* STRING, *operation* STRING, *input* RECORD)	Creates a new Web Operation object with One-Way Document style.

Name	Description
RETURNING *result* com.WebOperation	
com.WebOperation.CreateOneWayRPCStyle(*function* STRING, *operation* STRING, *input* RECORD) RETURNING *result* com.WebOperation	Creates a new Web Operation object with One-Way RPC style.
com.WebOperation.CreateRPCStyle(*function* STRING, *operation* STRING, *input* RECORD, *output* RECORD) RETURNING *result* com.WebOperation	Creates a new Web Operation object with RPC style.

Table 454: Object methods

Name	Description
addFault(*fault fault-type*, *wsaaction* STRING)	Adds a fault to the current Web Operation definition.
addInputHeader(*header header-type*)	Adds an input header for the current Web Operation definition.
addOutputHeader(*header header-type*)	Adds an output header for the current Web Operation definition.
initiateSession(*initiator* BOOLEAN)	Defines the Web Operation as session initiator.
setComment(*comment* STRING)	Sets the comment for the Web Operation object.
setInputAction(*ident* STRING)	Sets the WS-Addressing action identifier of the input operation.
setInputEncoded(*encoded* BOOLEAN)	Defines the encoding mechanism for Web Operation input parameters.
setOutputAction(*ident* STRING)	Sets the WS-Addressing action identifier of the output operation.
setOutputEncoded(	Defines the encoding mechanism for Web Operation output parameters.

Name	Description
encoded BOOLEAN)	

com.WebOperation.addFault

Adds a fault to the current Web Operation definition.

Syntax

```
addFault(
   fault fault-type,
   wsaction STRING )
```

1. *fault* is a program variable defining the fault.
2. *fault-type* is a simple data type, a RECORD or an ARRAY.
3. *wsaction* defines the type of action.

Usage

Adds a fault the Web Operation can throw during operation processing, where *fault* is any variable previously created as fault of the `com.WebService` object, and *wsaction* the WS-Addressing action identifier if WS-Addressing is supported. If WS-Addressing is not supported, pass NULL as second parameter.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.addInputHeader

Adds an input header for the current Web Operation definition.

Syntax

```
addInputHeader(
   header header-type )
```

1. *header* is a program variable defining the header.
2. *header-type* is a simple data type, a RECORD or an ARRAY.

Usage

This method adds a header to the Web Operation object for input parameters.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.addOutputHeader

Adds an output header for the current Web Operation definition.

Syntax

```
addOutputHeader(
   header header-type )
```

1. *header* is a program variable defining the header.
2. *header-type* is a simple data type, a RECORD or an ARRAY.

Usage

This method adds a header to the Web Operation object for input parameters.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.CreateDOCStyle

Creates a new Web Operation object with Document style.

Syntax

```
com.WebOperation.CreateDOCStyle(
   function STRING,
   operation STRING,
   input RECORD,
   output RECORD)
  RETURNING result com.WebOperation
```

1. *function* is the name of the program function to be called to process the XML operation.
2. *operation* is the name of the XML operation.
3. *input* is the variable defining the input parameters of the operation (or `NULL` if there is none).
4. *output* is the variable defining the output parameters of the operation (or `NULL` if there is none).

Usage

This method creates a Request-Response Document style `com.WebOperation` object, where *function* is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.CreateRPCStyle

Creates a new Web Operation object with RPC style.

Syntax

```
com.WebOperation.CreateRPCStyle(
   function STRING,
   operation STRING,
   input RECORD,
   output RECORD)
  RETURNING result com.WebOperation
```

1. *function* is the name of the program function to be called to process the XML operation.
2. *operation* is the name of the XML operation.
3. *input* is the input record defining the input parameters of the operation (or `NULL` if there is none).
4. *output* is the output record defining the output parameters of the operation (or `NULL` if there is none).

Usage

This method creates a Request-Response RPC style `com.WebOperation` object, where *function* is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.CreateOneWayDOCStyle

Creates a new Web Operation object with One-Way Document style.

Syntax

```
com.WebOperation.CreateOneWayDOCStyle(
   function STRING,
   operation STRING,
   input RECORD)
  RETURNING result com.WebOperation
```

1. *function* is the name of the program function to be called to process the XML operation.
2. *operation* is the name of the XML operation.
3. *input* is the variable defining the input parameters of the operation (or NULL if there is none).

Usage

This method creates a One-Way DOC style com.WebOperation object, where *function* is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

There is no output parameter to be returned to the client.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.CreateOneWayRPCStyle

Creates a new Web Operation object with One-Way RPC style.

Syntax

```
com.WebOperation.CreateOneWayRPCStyle(
   function STRING,
   operation STRING,
   input RECORD)
  RETURNING result com.WebOperation
```

1. *function* is the name of the program function to be called to process the XML operation.
2. *operation* is the name of the XML operation.
3. *input* is the input record defining the input parameters of the operation (or NULL if there is none).

Usage

This method creates a One-Way RPC Style com.WebOperation object, where *function* is the name of the program function that is executed to process the XML operation.

The function name must be a string literal, not a string variable, due to operation publication restrictions.

There is no output parameter to be returned to the client.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.initiateSession

Defines the Web Operation as session initiator.

Syntax

```
initiateSession(
   initiator BOOLEAN )
```

1. *initiator* must be TRUE to define a session initiator.

Usage

Pass TRUE as parameter to `initiateSession()` in order to define the current Web Operation as a session initiator.

A new session must be instantiated in this operation, and must be returned to the client via the *state* variable defined at service creation.

This method works only for stateful web services.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.setComment

Sets the comment for the Web Operation object.

Syntax

```
setComment(
   comment STRING )
```

1. *comment* is the comment to be set.

Usage

The `setComment()` method defines a comment to the current Web Operation object.

The comment will appear in the WSDL of the service.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.setInputAction

Sets the WS-Addressing action identifier of the input operation.

Syntax

```
setInputAction(
   ident STRING )
```

1. *ident* is the WSA action identifier.

Usage

When WS-Addressing is enabled, this method defines the WS-Addressing action identifier of the input operation.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.setInputEncoded

Defines the encoding mechanism for Web Operation input parameters.

Syntax

```
setInputEncoded(
   encoded BOOLEAN )
```

1. *encoded* is a boolean defining the encoding mechanism to be used.

Usage

When the parameter is TRUE, the SOAP Section 5 encoding mechanism is used, FALSE indicates the XML Schema mechanism.

The XML Schema mechanism (FALSE) is not recommended.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.setOutputAction

Sets the WS-Addressing action identifier of the output operation.

Syntax

```
setOutputAction(
   ident STRING )
```

1. *ident* is the WSA action identifier.

Usage

When WS-Addressing is enabled, this method defines the WS-Addressing action identifier of the ouput operation.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebOperation.setOutputEncoded

Defines the encoding mechanism for Web Operation output parameters.

Syntax

```
setOutputEncoded(
   encoded BOOLEAN )
```

1. *encoded* is a boolean defining the encoding mechanism to be used.

Usage

When the parameter is TRUE, the SOAP Section 5 encoding mechanism is used, FALSE indicates the XML Schema mechanism.

The XML Schema mechanism (FALSE) is not recommended.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The WebServiceEngine class

The `com.WebServiceEngine` class provides an interface to manage the Web Services engine.

> **Important:** This Web Services class is not supported on GMI mobile devices.

com.WebServiceEngine methods

Methods for the `com.WebServiceEngine` class.

Table 455: Class methods

Name	Description
`com.WebServiceEngine.Flush()` `  RETURNING status INTEGER`	Forces the Web Service engine to immediately flush the response of the web service operation.

`com.WebServiceEngine.GetHTTPServiceRequest(`
`  timeout INTEGER)`
`  RETURNING result com.HTTPServiceRequest`
	Get a handle for an incoming HTTP service request.

`com.WebServiceEngine.GetOption(`
`  option STRING )`
`  RETURNING result STRING`
	Returns the value of a Web Service engine option.

`com.WebServiceEngine.HandleRequest(`
`  timeout INTEGER,`
`  status INTEGER )`
`  RETURNING result com.HTTPServiceRequest`
	Get a handle for an incoming HTTP service request.

`com.WebServiceEngine.ProcessServices(`
`  timeout INTEGER )`
`  RETURNING status INTEGER`
	Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

`com.WebServiceEngine.RegisterService(` `  service com.WebService )`	Registers a service in the engine.

`com.WebServiceEngine.SetFaultCode(`
`  code STRING,`
`  code_ns STRING )`
	Get a handle for an incoming HTTP service request.

`com.WebServiceEngine.SetFaultDetail(` `  fault STRING )`	Defines the published SOAP Fault.
`com.WebServiceEngine.SetFaultString(` `  desc STRING )`	Defines the description of a SOAP Fault.
`com.WebServiceEngine.SetOption(` `  option STRING,`	Sets an option for the Web Service engine.

Name	Description
value STRING)	
com.WebServiceEngine.Start()	Starts the Web Service engine.

com.WebServiceEngine.Flush

Forces the Web Service engine to immediately flush the response of the web service operation.

Syntax

```
com.WebServiceEngine.Flush()
  RETURNING status INTEGER
```

Usage

The `com.WebServiceEngine.flush()` class method allows to return the response inside a high-level web service operation, before the end of the web service function.

When this method is used, any other web operation output parameter changes are ignored.

The *status* returned by the method provides information about the execution of the last web operation. A return status of zero means OK. For a complete list of error codes, see Error codes of com.WebServicesEngine on page 2034

Note: The return status of the `com.WebServiceEngine.flush()` method has the same meaning as a status returned by `com.WebServiceEngine.ProcessServices()`, with the additional status code -32, meaning that the flush method has been called outside a web operation execution context.

Note: `com.WebServiceEngine.ProcessServices()` and `com.WebServiceEngine.HandleRequest()` can return the status code of -31, meaning that the flush function has been called in the last executed web operation.

Example:

In this code example, the `flush()` method is used to force the response of the web service operation.

```
DEFINE echoBoolean_in, echoBoolean_out RECORD
    a_boolean  BOOLEAN ATTRIBUTES(XMLName="Boolean")
  END RECORD

MAIN
  DEFINE ret INTEGER
  ...
  WHILE true
    LET ret = com.WebServiceEngine.ProcessServices(-1)
    CASE ret
      WHEN 0
        DISPLAY "Request automatically processed."
      WHEN -31
        DISPLAY "Operation has been flushed."
      ...
  END WHILE
  ...
END MAIN

FUNCTION echoBoolean()
```

```
  DEFINE ret INTEGER
  -- Assign output parameter with input parameter
  LET echoBoolean_out.a_boolean = echoBoolean_in.a_boolean
  -- Immediate flush of web operation
  LET ret = com.WebServiceEngine.flush()
  IF ret != 0 THEN
     DISPLAY "ERROR Code : ",ret
     EXIT PROGRAM (1)
  END IF
  -- Changing the output parameters after flush() would have no
 effect.
END FUNCTION
```

com.WebServiceEngine.GetHTTPServiceRequest

Get a handle for an incoming HTTP service request.

Syntax

```
com.WebServiceEngine.GetHTTPServiceRequest(
   timeout INTEGER)
  RETURNING result com.HTTPServiceRequest
```

1. *timeout* is the timeout in seconds.

Usage

The `com.WebServiceEngine.GetHTTPServiceRequest()` class method returns a `com.HTTPServiceRequest` object to handle an incoming HTTP request, or `NULL` if there was no request during the given period of time.

The *timeout* parameter defines the time in seconds to wait for an incoming request. A value of -1 means infinite wait. When the timeout occurs, the method returns `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

Any new call to this function will raise an error until the previous HTTP request was handled by sending a response back to the client, or destroyed.

The error -15575 can be thrown if the GAS disconnects the Web Services program.

URLs are sent in UTF-8 on the network, if the web services server is not able to convert UTF-8 URLs back to fglrun locale charset, error -15552 will be thrown. As a general advice, run you WS server program in UTF-8.

Example

```
TRY
  WHILE true
    LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)
    IF req IS NULL THEN
      DISPLAY "HTTP request timeout...: ", CURRENT YEAR TO
 FRACTION
    ELSE
      CALL req.sendTextResponse(200,NULL,"It works")
    END IF
  END WHILE
CATCH
  IF status == -15575 THEN
    DISPLAY "Disconnected : ",SQLCA.SQLERRM
```

```
  ELSE
    DISPLAY "ERROR : ",status,SQLCA.SQLERRM
  END IF
END TRY
```

com.WebServiceEngine.GetOption

Returns the value of a Web Service engine option.

Syntax

```
com.WebServiceEngine.GetOption(
   option STRING )
  RETURNING result STRING
```

1. *option* is the name of the option to queried.

Usage

The `com.WebServiceEngine.GetOption()` class method returns the current value of the given the Web Services engine option.

See WebServiceEngine options on page 2032 for the supported options.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebServiceEngine.HandleRequest

Get a handle for an incoming HTTP service request.

Syntax

```
com.WebServiceEngine.HandleRequest(
   timeout INTEGER,
   status INTEGER )
  RETURNING result com.HTTPServiceRequest
```

1. *timeout* is the timeout in seconds.
2. *status* is an `INTEGER` variable receiving the method execution status.

Usage

The `com.WebServiceEngine.HandleRequest()` class method returns a `com.HTTPServiceRequest` object to handle an incoming HTTP request, or `NULL` if there was no request during the given period of time.

The *timeout* parameter defines the time in seconds to wait for an incoming request. A value of -1 means infinite wait.

- If there is no request in the given period of time, or if there is an error, the status code is updated by reference, and a `NULL` object is returned.
- If the request is intended to a registered web service, it is processed automatically. The status code is updated by reference and a `NULL` object is returned.
- If the request isn't dedicated to a registered web service, a status code of value 1 is returned by reference, and a valid instance of an `com.HTTPServiceRequest` object, immediately usable to handle the incoming request, is returned.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Any new call to this function will raise an error until the previous HTTP request was handled by sending a response back to the client, or destroyed.

The *status* returned by the method provides information about the execution of the last web operation. A return status of zero means OK. For a complete list of error codes, see Error codes of com.WebServicesEngine on page 2034

> **Note:** If the com.WebServiceEngine.Flush() method is used, the return status handling must be done in the web operation function, while com.WebServiceEngine.HandleRequest() will return the code -31, to indicated that a flush was done.

The error -15575 can be thrown if the GAS disconnects the Web Services program.

URLs are sent in UTF-8 on the network, if the web services server is not able to convert UTF-8 URLs back to fglrun locale charset, error -15552 will be thrown. As a general advice, run you WS server program in UTF-8.

com.WebServiceEngine.ProcessServices

Specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

Syntax

```
com.WebServiceEngine.ProcessServices(
   timeout INTEGER )
  RETURNING status INTEGER
```

1. *timeout* is the timeout in seconds.

Usage

The com.WebServiceEngine.ProcessServices() class method specifies the wait period for an HTTP input request, to process an operation of one of the registered Web Services.

The *timeout* parameter defines the wait period for an HTTP input request, to process an operation of one of the registered Web Services. The value -1 specifies an infinite waiting time.

The *status* returned by the method provides information about the execution of the last web operation. A return status of zero means OK. For a complete list of error codes, see Error codes of com.WebServicesEngine on page 2034

The execution status is typically handled in a CASE / END CASE block, to treat all possible execution cases. For a complete example of execution status handling, see Process the requests on page 2477.

> **Note:** If the com.WebServiceEngine.Flush() method is used, the return status handling must be done in the web operation function, while com.WebServiceEngine.ProcessServices() will return the code -31, to indicated that a flush was done.

com.WebServiceEngine.SetFaultCode

Get a handle for an incoming HTTP service request.

Syntax

```
com.WebServiceEngine.SetFaultCode(
   code STRING,
   code_ns STRING )
```

1. *code* is the timeout in seconds.

2. *code_ns* is the timeout in seconds.

Usage

The `com.WebServiceEngine.SetFaultCode()` class method defines a user SOAP Fault code to be returned to the client, where *code* is the mandatory SOAP Fault code and *code_ns* is the mandatory code namespace.

This method must be called inside a Web Service operation.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebServiceEngine.SetFaultDetail

Defines the published SOAP Fault.

Syntax

```
com.WebServiceEngine.SetFaultDetail(
   fault STRING )
```

1. *fault* is the published fault.

Usage

The `com.WebServiceEngine.SetFaultDetail()` class method defines the published SOAP Fault to be returned to the client when operation has finished, where *fault* is one of the published variables defined as Fault for that operation.

This method must be called inside a Web Service operation.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebServiceEngine.SetFaultString

Defines the description of a SOAP Fault.

Syntax

```
com.WebServiceEngine.SetFaultString(
   desc STRING )
```

1. *desc* is the description of the fault.

Usage

The `com.WebServiceEngine.SetFaultString()` class method defines a user SOAP Fault description to be returned to the client, where *desc* contains the description of the fault.

This method must be called inside a Web Service operation.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebServiceEngine.SetOption

Sets an option for the Web Service engine.

Syntax

```
com.WebServiceEngine.SetOption(
   option STRING,
   value STRING )
```

1. *option* is the name of the option to set.
2. *value* is the value of the option to set.

Usage

The `com.WebServiceEngine.SetOption()` class method configures the Web Services engine with options.

See WebServiceEngine options on page 2032 for the supported options.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebServiceEngine.RegisterService

Registers a service in the engine.

Syntax

```
com.WebServiceEngine.RegisterService(
   service com.WebService )
```

1. *service* is the service object to register.

Usage

The `com.WebServiceEngine.RegisterService()` class method registers the `com.WebService` object passed as parameter.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.WebServiceEngine.Start

Starts the Web Service engine.

Syntax

```
com.WebServiceEngine.Start()
```

Usage

The `com.WebServiceEngine.Start()` class method starts the engine for all registered Web Services.

If you run the Web Services server program in standalone mode, you must set FGLAPPSERVER. If you run the Web Services server program through the Genero Application Server, the FGLAPPSERVER variable is automatically set by the Genero Application Server. Do NOT manually set FGLAPPSERVER in this case.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

WSDL generation options notes
These notes should be reviewed prior to WSDL generation.

1. For the `DECIMAL(5,2)` data type, when `wsdl_decimalsize` is `TRUE`, the generated WSDL file contains the total size and the size of the fractional part of the decimal:

```
<types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.mycompany.com/types/">
      <simpleType name="echoDecimal5_2_a_dec5_2_out_FGLDecimal">
         <restriction base="decimal">
            <totalDigits value="5" />
             <fractionDigits value="2" />
          </restriction>
      </simpleType>
   </schema>
 </types>
<message name="echoDecimal5_2">
  <part name="dec5_2" type="f:echoDecimal5_2_a_dec5_2_in_FGLDecimal" />
 </message>
```

   When `wsdl_decimalsize` is `FALSE`, the total size and the size of the fractional part are not mentioned:

```
<message name="echoDecimal5_2">
  <part name="dec5_2" type="xsd:decimal" />
 </message>
```

2. If the WSDL file does not contain the size, the client application has no way of knowing the size. In this case, a default value for the size is generated. For example, the exported server type `DECIMAL(5,2)` becomes a `DECIMAL(32)` on the client side.
3. It is better to keep the options `wsdl_arraysize`, `wsdl_stringsize` and `wsdl_decimalsize` set to `TRUE` so that the client program can do exact type mapping. The default for all three options is `TRUE`.
4. When setting a facet constraint attribute on a simple data type, the generation of the WSDL will take this attribute into account even if an option has been set to perform the opposite.
5. When setting one facet constraint attribute, all of the default constraint attributes won't be generated anymore unless you specify them as facet constraint attributes.

WebServiceEngine options

Table 456: Options for the com.WebServiceEngine

Flag	Client or Server	Description
readwritetimeout	Client	Defines the default maximum time in seconds a client, a HTTP request/response and a TCP request/response have to wait before raising an error that the server doesn't return or accept data. A value of -1 means infinite wait. The default is -1.
connectiontimeout	Client	Defines the default maximum time in seconds a client, a HTTPRequest and a TCPRequest have to wait for the establishment of a connection with a server. A value of -1 means infinite wait.

Flag	Client or Server	Description
		The default is 30 seconds for non-Windows, 5 seconds for Windows™.
maximumresponselength	Both	Defines the maximum authorized size in **KBytes** for a client, server, HTTP or TCP response, before a break (when it stops and returns from the function because the amount of data surpassed the maximumresponselength.) A value of -1 means no limit. The default is -1.
wsdl_decimalsize	Server	Defines whether the precision and scale of a DECIMAL variable will be taken into account during the WSDL generation. See WSDL generation options notes on page 2032. A value of zero means FALSE. The default is TRUE.
wsdl_arraysize	Server	Defines whether the size of a BDL array will be taken into account during the WSDL generation. See WSDL generation options notes on page 2032. A value of zero means FALSE. The default is TRUE.
wsdl_stringsize	Server	Defines whether the size of a CHAR or VARCHAR variable will be taken into account during the WSDL generation. See WSDL generation options notes on page 2032. A value of zero means FALSE. The default is TRUE.
http_invoketimeout (deprecated)	Client	Defines the default maximum time in seconds a client has to wait before the client connection raises an error because the server is not responding. A value of -1 means that it has to wait until the server responds. The default is -1. **Important:** Deprecated - use readwritetimeout
server_readwritetimeout	Server	Defines how long a socket read or write operation can wait before before raising an error. The default value is 5 seconds. **Note:** Before this option was introduced, the default valus was -1 (infinite) and was configurable with the accept timeout parameter via `ProcessServices()` method.

Flag	Client or Server	Description
SoapModuleURI	Both	Defines the SOAP role of a Genero application with an URI to identify it along a SOAP message path. The default value is NULL.
tcp_connectiontimeout (deprecated)	Client	Defines the default maximum time in seconds a client has to wait for the establishment of a TCP connection with a server. A value of -1 means infinite wait. The default is 30 seconds for non-Windows, 5 seconds for Windows™. **Important:** Deprecated - use connectiontimeout

Error codes of com.WebServicesEngine
Error codes returned by `com.WebServiceEngine` methods.

Table 457: com.WebServiceEngine error codes

Number	Description
-1	`Timeout` com.WebServiceEngine.ProcessServices(x) timeout is reached. No requests to process during x seconds.
-2	`AsCloseCommand` GAS tells the DVM to shutdown. You must exit your application.
-3	`ConnectionBroken` Client has closed the connection in standalone GWS (without GAS).
-4	`ConnectionInterrupted` Ctrl-C received. Interruption received by DVM. You must exit your application.
-5	`BadHTTPHeader` Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-6	`MalformedSOAPEnvelope` Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-7	`MalformedXMLDocument` Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-8	`InternalHTTPError` Communication issue with application server or client.
-9	`Unsupported operation` The URL of the operation requested is unknown. Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.DONE

Number	Description
-10	`UnknownError` This is an internal error, contact the support team. You must exit your application.
-11	`WSDL generation failed` You need to debug your application.
-12	`WSDL Service not found` Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-13	`Reserved` No need to exit the application. A new request might not have the issue.
-14	`Incoming request overflow` You exceed the data maximum length allowed by com.WebServiceEngine.SetOption(maximumresponselength).
-15	`Server was not started` Call to com.WebServiceEngine.Start() failed. You must exit your application.
-16	`Request still in progress` With RESTful service, you are currently processing a request and has not yet send the response and try to process another request. You need to debug your application. It depends, you might not need to stop your application.
-17	`Stax response error` You need to debug your application. Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-18	`Input request handler error` You need to debug your application. Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-19	`Output request handler error` You need to debug your application. Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-20	`WSDL handler error` You need to debug your application. Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-21	`SOAP Version mismatch` Your client SOAP version does not match your server SOAP version, amend either your client or your server code.
-22	`SOAP header not understood` Modify your server code to handled the mustUnderstand attribute. Use the incoming request handler.
-23	`Deserialization error`

Number	Description
	Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.
-24	`Reserved error code -24` This error code is reserved for future use.
-25	`Web Services Addressing action is mandatory` Check that the WSA action is specified in the SOAP message.
-26	`Web Services Addressing message header is invalid` Check that the WSA header is correct in the SOAP message.
-27	`Web Services Addressing message header is mandatory` Check that the WSA header is specified in the SOAP message.
-28	`Web Services Addressing message protocol does not match` Check that the WSA message uses the protocol version of the client matches the version expected by the server.
-29	`Cookie error` Check that the HTTP request contains a valid cookie.
-30	`No active web operation` The method was called outside the context of a web operation processing.
-31	`Web Operation was flushed` This code is returned by the `ProcessServices()` or the `HandlerRequest()` method, to indicated that the `Flush()` method was called during the last web operation execution.
-32	`Serialization error` Check the message with FGLWSDEBUG or display SQLCA.SQLERRM.

The HTTPServiceRequest class

The `com.HTTPServiceRequest` class provides an interface to process incoming XML and TEXT requests over HTTP on the server side, with an access to the HTTP layer and additional XML streaming possibilities.

Important: This Web Services class is not supported on GMI mobile devices.

com.HTTPServiceRequest methods

Methods of the `com.HTTPServiceRequest` class.

Table 458: Object methods: Reading client requests

Name	Description
`beginXmlRequest()` `  RETURNING` *reader* `xml.StaxReader`	Starts an HTTP streaming request.
`endXmlRequest(`	Terminates an HTTP streaming request.

Name	Description
reader xml.StaxReader)	
getMethod() RETURNING *result* STRING	Returns the HTTP method of the service request.
getRequestHeader(name STRING) RETURNING *result* STRING	Returns the value of an HTTP header.
getRequestVersion() RETURNING *result* STRING	Returns the HTTP version of the service request.
getURL() RETURNING *result* STRING	Returns the URL of the HTTP service request.
getRequestHeaderCount() RETURNING *result* INTEGER	Returns number of request headers.
getRequestHeaderName(*index* INTEGER) RETURNING *result* STRING	Returns a request header name by position.
getRequestHeaderValue(*index* INTEGER) RETURNING *result* STRING	Returns a request header value by position.
hasRequestKeepConnection() RETURNING *result* BOOLEAN	Returns TRUE if the connection remains after sending a response.
readDataRequest(*body* BYTE)	Returns the body of a request into a BYTE.
readFileRequest( ) RETURNING *filename* STRING	Returns the body of a request into a file.
readFormEncodedRequest(*utf8* BOOLEAN) RETURNING *result* STRING	Returns the string of a GET request with UTF-8 conversion option.
readTextRequest() RETURNING *result* STRING	Returns the request body as a plain string.
readXmlRequest() RETURNING *result* xml.DomDocument	Returns the request body as an XML document.

Table 459: Object methods: Responding to the client

Name	Description
`beginXmlResponse( code INTEGER, desc STRING ) RETURNING writer xml.StaxWriter`	Starts an HTTP streaming response.
`endXmlResponse( writer xml.StaxWriter )`	Terminates an HTTP streaming response.
`sendDataResponse( code INTEGER, desc STRING, data BYTE )`	Sends and HTTP response with data of a BYTE variable.
`sendFileResponse( code INTEGER, desc STRING, filepath STRING )`	Sends and HTTP response with the data contained in a file.
`sendResponse( code INTEGER, desc STRING )`	Sends and HTTP response without body.
`sendTextResponse( code INTEGER, desc STRING, data STRING )`	Sends and HTTP response with data from a plain string.
`sendXmlResponse( code INTEGER, desc STRING, data xml.DomDocument )`	Sends and HTTP response with data from a XML document object.
`setResponseCharset( charset STRING )`	Defines the HTTP response character set.
`setResponseHeader( name STRING, value STRING )`	Defines a header for the HTTP response.
`setResponseVersion( version STRING )`	Defines the HTTP response version.

Table 460: Object methods: Incoming multipart request

Name	Description
`getRequestMultipartType()` `  RETURNING type STRING`	Returns the multipart type of an incoming request.

`getRequestPart(`
`  idx INTEGER)`
`  RETURNING part-object com.HTTPPart`
	Returns the HTTPPart object at the specified index position.

`getRequestPartCount()` `  RETURNING num INTEGER`	Returns the number of additional multipart elements.

`getRequestPartFromContentID(`
`  id STRING)`
`  RETURNING part-object com.HTTPPart`
	Returns the HTTPPart object of the given Content-ID value.

Table 461: Object methods: Outgoing multipart request

Name	Description

`setResponseMultipartType(`
`  type STRING,`
`  start STRING,`
`  boundary STRING )`
	Sets HTTP response in multipart mode of given type.

`addResponsePart(` `  part-object com.HTTPPart)`	Adds a new part to the HTTP root part response.

com.HTTPServiceRequest.addResponsePart

Adds a new part to the HTTP root part response.

Syntax

```
addResponsePart(
   part-object com.HTTPPart)
```

Usage

Adds a new part to the HTTP root part response. It will be sent after root part has be processed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.beginXmlRequest

Starts an HTTP streaming request.

Syntax

```
beginXmlRequest()
```

```
   RETURNING reader xml.StaxReader
```

1. *reader* is a new `xml.StaxReader` object that will be used for streaming.

Usage

The `beginXmlRequest()` method starts the streaming HTTP request and returns a `xml.StaxReader` object ready to read the XML from the client.

Supported methods are PUT and POST.

The request Content-Type header must be of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.beginXmlResponse

Starts an HTTP streaming response.

Syntax

```
beginXmlResponse(
   code INTEGER,
   desc STRING )
  RETURNING writer xml.StaxWriter
```

1. *code* is the status code of the response.
2. *desc* is the description of the response.
3. *writer* is a new `xml.StaxWriter` the will be used to write the HTTP body.

Usage

The `beginXmlResponse()` method starts a HTTP streaming response by sending the a status (*code*) and description (*desc*), followed by the headers previously set, and returns a `xml.StaxWriter` object ready to send XML as the HTTP body.

If the request failed to be read, its content will be discarded; for example, when a request is not well formatted.

The default Content-Type header is `text/xml`, but it can be changed if of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description according to the status code is sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.endXmlRequest

Terminates an HTTP streaming request.

Syntax

```
endXmlRequest(
   reader xml.StaxReader )
```

1. *reader* is the `xml.StaxReader` object used for streaming.

Usage

The `endXmlRequest()` method ends the streaming HTTP request by closing the `xml.StaxReader` object passed as parameter.

The *reader* object must be created with the `beginXmlRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.endXmlResponse

Terminates an HTTP streaming response.

Syntax

```
endXmlResponse(
   writer xml.StaxWriter )
```

1. *writer* is the `xml.StaxWriter` used to write the HTTP body.

Usage

The `endXmlResponse()` method terminates the HTTP streaming response by closing the `xml.StaxWriter` object created by `beginXmlResponse`.

The body of the request is discarded.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.getURL

Returns the URL of the HTTP service request.

Syntax

```
getURL()
  RETURNING result STRING
```

Usage

The `getURL()` method returns the entire URL request containing the host, port, document and query string.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

URLs are sent in UTF-8 on the network. If the query part of the URL cannot be converted from UTF-8 to the fglrun locale charset, STATUS will be set to -15552. Is this case, the document part of the URL is available, but the query string must be retrieved through HTTPServiceRequest.readFormEncodedRequest(). As a general advice, run you WS server program in UTF-8.

com.HTTPServiceRequest.getMethod

Returns the HTTP method of the service request.

Syntax

```
getMethod()
  RETURNING result STRING
```

Usage

The `getMethod()` method returns the HTTP method of the request (GET, POST, PUT, HEAD, DELETE).

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestVersion

Returns the HTTP version of the service request.

Syntax

```
getRequestVersion()
  RETURNING result STRING
```

Usage

The `getRequestVersion()` method returns the HTTP version of the request (1.0 or 1.1).

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestHeader

Returns the value of an HTTP header.

Syntax

```
getRequestHeader( name STRING )
  RETURNING result STRING
```

1. *name* is the name of an HTTP header.

Usage

The `getRequestHeader()` method returns the value of the HTTP header specified by the *name* parameter, or `NULL` if there is not found.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestHeaderCount

Returns number of request headers.

Syntax

```
getRequestHeaderCount()
  RETURNING result INTEGER
```

Usage

The `getRequestHeaderCount()` method returns the entire URL request containing the host, port, document and query string.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestHeaderName

Returns a request header name by position.

Syntax

```
getRequestHeaderName(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the ordinal position of the header.

Usage

The `getRequestHeaderName()` method returns the name of the header at the given position.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestHeaderValue

Returns a request header value by position.

Syntax

```
getRequestHeaderValue(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the ordinal position of the header.

Usage

The `getRequestHeaderValue()` method returns the value of the header at the given position.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestMultipartType
Returns the multipart type of an incoming request.

Syntax

```
getRequestMultipartType()
  RETURNING type STRING
```

Usage

Returns the multipart type of an incoming request, returns NULL if not a multipart request.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestPart
Returns the HTTPPart object at the specified index position.

Syntax

```
getRequestPart(
   idx INTEGER)
  RETURNING part-object com.HTTPPart
```

1. *idx* is the index position.

Usage

Returns the HTTPPart object at the specified index position.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Can raise error -15554 (Index is out of bounds).

com.HTTPServiceRequest.getRequestPartCount
Returns the number of additional multipart elements.

Syntax

```
getRequestPartCount()
  RETURNING num INTEGER
```

Usage

The root multipart is handled via standard `readTextRequest()`, `readXmlRequest()`, `readDataRequest()` and `beginXmlRequest()`.

The number of parts is only available when the entire request has been read.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.getRequestPartFromContentID

Returns the HTTPPart object of the given Content-ID value.

Syntax

```
getRequestPartFromContentID(
   id STRING)
  RETURNING part-object com.HTTPPart
```

1. *id* is a Content-ID value.

Usage

Returns the HTTPPart object of the given Content-ID value, returns NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.hasRequestKeepConnection

Returns TRUE if the connection remains after sending a response.

Syntax

```
hasRequestKeepConnection()
  RETURNING result BOOLEAN
```

Usage

The hasRequestKeepConnection() method returns whether the request expect the connection to stay open after the sending of the response.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.readDataRequest

Returns the body of a request into a BYTE.

Syntax

```
readDataRequest(
  body BYTE)
```

1. *body* is the BYTE variable that will be filled with the request body.

Usage

The readDataRequest() method returns the body of the request in a BYTE.

Supported methods are PUT and POST.

The BYTE variable must be located in memory, and will be filled with the request body. The existing content of the BYTE will be discarded.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.readFileRequest

Returns the body of a request into a file.

Syntax

```
readFileRequest( )
    RETURNING filename STRING
```

1. *filename* the absolute path to the file containing the HTTP response.

Usage

The `readFileRequest()` method returns the body of the request into a file on the disk.

The file is created in the temporary directory used by the runtime system (DBTEMP). The name of the file will be the basename found in the HTTP Content-Disposition Header, if this basename is not specified, the filename will be created with a UUID. If a file with the same name already exists in the temporary directory, the API prefixes the new file with a number. It is then of the form : `/tmp/ABC/filename_`*`index`*`.ext`, where *index* represents the number of files with the same name on disk.

Supported methods are PUT and POST.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.readFormEncodedRequest

Returns the string of a GET request with UTF-8 conversion option.

Syntax

```
readFormEncodedRequest(
   utf8 BOOLEAN )
  RETURNING result STRING
```

1. *utf8* defines if the string must be decoded to the current charset.

Usage

The `readFormEncodedRequest()` method returns the query of a POST "application/x-www-form-urlencoded" request or the query string of a GET request, decoded according to HTML4 or XFORM if *utf8* is `TRUE`.

> **Note:** If the result string contains & or = XForms special characters, these are doubled as follows: `na&&me=va==lue`.

If the *utf8* parameter is `TRUE`, the decoded query string is translated from UTF-8 to the current character set. This may lead to a conversion error.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.readTextRequest

Returns the request body as a plain string.

Syntax

```
readTextRequest()
  RETURNING result STRING
```

Usage

The `readTextRequest()` method returns the body of the request as a string.

Supported methods are PUT and POST.

The request Content-Type header can be of any form `*/*`. For example: `application/json`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.readXmlRequest

Returns the request body as an XML document.

Syntax

```
readXmlRequest()
  RETURNING result xml.DomDocument
```

Usage

The `readXmlRequest()` method returns the request as an entire XML document, contained in a `xml.DomDocument` object.

Supported methods are PUT and POST.

The request Content-Type header must be of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.sendDataResponse

Sends and HTTP response with data of a `BYTE` variable.

Syntax

```
sendDataResponse(
   code INTEGER,
   desc STRING,
   data BYTE )
```

1. *code* is the status code of the response.
2. *desc* is the description of the response.

3. *data* is the BYTE variable containing the data to be sent.

Usage

The `sendDataResponse()` method performs the HTTP response by sending the status (*code*) and description (*desc*), followed by the headers previously set and binary data contained in the BYTE program variable as body.

It is important for the server to return a correct status *code*, following the HTTP standards, otherwise the client may fail to interprete the response. For instance, if the request is malformed, the server should send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

The *data* parameter is defined as a BYTE and must be located in memory and not NULL, otherwise the operation fails.

The default Content-Type header is `application/octet-stream`, but it can be changed to any other mime type. For example: `image/jpeg`.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is NULL, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.sendFileResponse

Sends and HTTP response with the data contained in a file.

Syntax

```
sendFileResponse(
   code INTEGER,
   desc STRING,
   filepath STRING )
```

1. *code* is the status code of the response.
2. *desc* is the description of the response.
3. *filepath* is the path the file containing the data to be send.

Usage

The `sendFileResponse()` method performs the HTTP response by sending the status (*code*) and description (*desc*), followed by the headers previously set and the data contained in the specified file as body.

It is important for the server to return a correct status *code*, following the HTTP standards, otherwise the client may fail to interprete the response. For instance, if the request is malformed, the server should send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

If not defined by programmer, the HTTP headers are automatically set as follows:

- `Content-Type` is defined according to the file name extension. If the file extension is not recognized, `Content-Type` defaults to `application/octet-stream`.

Note: File extensions to `Content-Type` mapping can be customized in the file `FGLDIR/lib/wse/mime.cfg`.

- `Content-Disposition` is set with the base name of the given *filename* as follows: `attachment; filename="basename"`.

For example, when calling the method as follows:

```
CALL server.sendFileResponse( 200, NULL, "/opt/myapp/resources/logo.jpg" )
```

The resulting HTTP headers of the response will look like:

```
Content-Type: image/jpeg
Content-Disposition: attachment; filename="logo.jpg"
```

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.sendResponse

Sends and HTTP response without body.

Syntax

```
sendResponse(
   code INTEGER,
   desc STRING )
```

1. *code* is the status code of the response.
2. *desc* is the description of the response.

Usage

The `sendResponse()` method performs the HTTP response by sending the a status (*code*) and description (*desc*), followed by the headers previously set, without a body.

It is important for the server to return a correct status *code*, following the HTTP standards, otherwise the client may fail to interprete the response. For instance, if the request is malformed, the server should send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

If the description is `NULL`, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.sendTextResponse

Sends and HTTP response with data from a plain string.

Syntax

```
sendTextResponse(
   code INTEGER,
   desc STRING,
   data STRING )
```

1. *code* is the status code of the response.
2. *desc* is the description of the response.
3. *data* is the string containing the data to be sent.

Usage

The `sendTextResponse()` method performs the HTTP response by sending the a status (*code*) and description (*desc*), followed by the headers previously set, and text data contained in the string as body.

It is important for the server to return a correct status *code*, following the HTTP standards, otherwise the client may fail to interprete the response. For instance, if the request is malformed, the server should send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

The default Content-Type header is `text/plain`, but it can be changed if of the form `*/*`. For example: `application/json`.

Automatic conversion from locale to user-defined charset is performed when possible, otherwise throws an exception.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.sendXmlResponse

Sends and HTTP response with data from a XML document object.

Syntax

```
sendXmlResponse(
   code INTEGER,
   desc STRING,
   data xml.DomDocument )
```

1. *code* is the status code of the response.
2. *desc* is the description of the response.
3. *data* is the XML document containing the data to be sent.

Usage

The `sendXmlResponse()` method performs the HTTP response by sending the a status (*code*) and description (*desc*), followed by the headers previously set, and the XML data contained in the passed `xml.DomDocument` object as body.

It is important for the server to return a correct status *code*, following the HTTP standards, otherwise the client may fail to interprete the response. For instance, if the request is malformed, the server should send an HTTP response with the code of 400 (Bad Request). See HTTP status codes (wikipedia) for more details about common HTTP response codes.

The default Content-Type header is `text/xml`, but it can be changed if of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In HTTP 1.1, if the body size is greater than 32k, the response will be sent in several chunks of the same size.

If the description is `NULL`, a default description according to the status code is sent.

New incoming requests can be retrieved again with the `com.WebServiceEngine.GetHTTPServiceRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPServiceRequest.setResponseCharset

Defines the HTTP response character set.

Syntax

```
setResponseCharset(
   charset STRING )
```

1. *charset* is the HTTP response character set.

Usage

The `setResponseCharset()` method defines the character set to use when sending an HTTP response.

The server must send a response in a character set that the client understands.

If the response character set is not defined by `setResponseCharset()`, the same character set as the client request is used, or the implicit ISO-8859-1 charset is used if the character is not defined by the client request.

The method must be called before sending the response with one of `sendResponse`, `sendTextResponse`, `sendXmlResponse`, or `beginXmlResponse` and `endXmlResponse` methods.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.setResponseHeader

Defines a header for the HTTP response.

Syntax

```
setResponseHeader(
   name STRING,
```

```
   value STRING )
```

1. *name* is the name of a header to define.
2. *value* is the value of a header to define.

Usage

The `setResponseVersion()` method sets (or replaces) the name and value of a HTTP response header.

The Content-Length header cannot be set, because it is computed internally according to the body size.

The method must be called before sending the response with one of `sendResponse`, `sendTextResponse`, `sendXmlResponse`, or `beginXmlResponse` and `endXmlResponse` methods.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.setResponseMultipartType

Sets HTTP response in multipart mode of given type.

Syntax

```
setResponseMultipartType(
   type STRING,
   start STRING,
   boundary STRING )
```

1. *type* is one of the following:
   - form-data: Browser Xform with attachment
   - mixed: Parts are independent
   - related: Parts are dependent (Required for SOAP)
   - alternative: Parts are different type of a same document
   - *or any other type*
   - NULL: switch multipart mode off
2. *start* is the Content-ID value of root multipart document. Must be ASCII. (optional)
3. *boundary* is the string used as multipart boundary. Must be ASCII. (optional)

Usage

Sets HTTP response in multipart mode of given type. Calling one of the standard request method will send the HTTP response as given multipart type, even if no other part has been set.

The root HTTP part must be handled via the standard HTTPServiceRequest methods such as `sendTextRequest()`, `sendXmlRequest()`, `sendDataRequest()` and `BeginXmlResponse()`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPServiceRequest.setResponseVersion

Defines the HTTP response version.

Syntax

```
setResponseVersion(
   version STRING )
```

1. *version* is the HTTP response version.

Usage

The `setResponseVersion()` method defines the HTML response version (1.0 or 1.1).

If not set, the same version as the request is used.

The method must be called before sending the response with one of `sendResponse`, `sendTextResponse`, `sendXmlResponse`, or `beginXmlResponse` and `endXmlResponse` methods.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

HTTP classes

The HTTP classes manage HTTP client network operations on the client side.

- The HTTPRequest class on page 2053
- The HTTPResponse class on page 2070
- The HTTPPart class on page 2077

The HTTPRequest class

The `com.HTTPRequest` class provides an interface to perform asynchronous XML and TEXT requests over HTTP for a specified URL, with additional XML streaming possibilities, on the client side.

The `STATUS` variable is set to zero after a successful method call.

com.HTTPRequest methods
Methods for the `com.HTTPRequest` class.

Table 462: Class methods

Name	Description

`com.HTTPRequest.Create(`
`  url STRING )`
`  RETURNING result com.HTTPRequest`
	Creates an new HTTPRequest object from a URL.

Table 463: Object methods: Configuration methods

Name	Description
`clearAuthentication()`	Removes user-defined authentication.
`clearHeaders()`	Removes all user-defined HTTP request headers.
`removeHeader(` `  name STRING )`	Removes an HTTP header for the request according to a name.

`setAuthentication(`
`  login STRING,`
`  pass STRING,`
`  scheme STRING,`
	Defines the user login and password to authenticate to the server.

Name	Description
realm STRING)	
setAutoReply(*reply* BOOLEAN)	Defines the auto reply option for response methods.
setCharset(*charset* STRING)	Defines the charset used when sending text or XML.
setConnectionTimeOut(*timeout* INTEGER)	Defines the timeout for the establishment of the connection.
setHeader(*name* STRING, *value* STRING)	Sets an HTTP header for the request.
setMethod(*method* STRING)	Sets the HTTP method of the request.
setKeepConnection(*keep* BOOLEAN)	Defines if connection is kept open if a new request occurs.
setMaximumResponseLength(*length* INTEGER)	Defines the maximum size in Kbyte of a response.
setTimeOut(*timeout* INTEGER)	Defines the timeout for a reading or writing operation.
setVersion(*version* STRING)	Sets the HTTP version of the request.

Table 464: Object methods: Sending methods

Name	Description
beginXmlRequest() RETURNING *writer* xml.StaxWriter	Starts a streaming HTTP request.
endXmlRequest(*writer* xml.StaxWriter)	Terminates a streaming HTTP request.
doDataRequest(	Performs the request by sending binary data.

Name	Description
data BYTE)	
doFileRequest(*filepath* STRING)	Performs the request by sending data contained in a file.
doFormEncodedRequest(*query* STRING, *utf8* BOOLEAN)	Performs an "application/x-www-form-urlencoded forms" encoded query.
doRequest()	Performs the HTTP request.
doTextRequest(*data* STRING)	Performs the request by sending an entire string at once.
doXmlRequest(*data* xml.DomDocument)	Performs the request by sending an entire XML document at once.

Table 465: Object methods : Response methods

Name	Description
getAsyncResponse() RETURNING *result* com.HTTPResponse	When available, returns the response produced by one of request methods.
getResponse() RETURNING *result* com.HTTPResponse	Waits and returns the response produced by one of request methods.

Table 466: Object methods of com.HTTPRequest : Multipart methods

Name	Description
addPart(*p* com.HTTPPart)	Adds a new part to the HTTP root part request.
setMultipartType(*type* STRING, *start* STRING, *boundary* STRING)	Switch HTTPRequest in multipart mode of given type.

com.HTTPRequest.addPart

Adds a new part to the HTTP root part request.

Syntax

```
addPart( p com.HTTPPart )
```

1. *p* is the HTTPPart object.

Usage

Adds a new part to the HTTP root part request. This part is sent after root part has been processed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.beginXmlRequest

Starts a streaming HTTP request.

Syntax

```
beginXmlRequest()
  RETURNING writer xml.StaxWriter
```

1. *writer* is the xml.StaxWriter to be used to write the HTTP request.

Usage

The beginXmlRequest() starts a streaming HTTP request and returns an xml.StaxWriter object ready to send XML to the server.

Supported methods are PUT and POST.

The default Content-Type header is text/xml, but it can be changed if of the form */xml or */*+xml. For example: application/xhtml+xml.

In HTTP 1.1, if the body size is greater than 32 KB, the request will be sent in several chunks of the same size.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.clearAuthentication

Removes user-defined authentication.

Syntax

```
clearAuthentication()
```

Usage

Removes user-defined authentication.

If an authenticate entry exists in the FGLPROFILE file, it will be used for authentication, even if the user-defined authentication was removed.

> **Important:** The iOS HTTP stack doesn't provide a simple way to handle authentication. The GMI front-end uses the global iOS credential management system, that keeps credential value of previous request according to host and realm, until the keep-alive session is closed. Therefore, doing a clearAuthentication() on iOS devices is not working immediately.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.clearHeaders

Removes all user-defined HTTP request headers.

Syntax

```
clearHeaders()
```

Usage

Removes all user-defined HTTP request headers defined with the `setHeader()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.Create

Creates an new HTTPRequest object from a URL.

Syntax

```
com.HTTPRequest.Create(
   url STRING )
  RETURNING result com.HTTPRequest
```

1. *url* is the URL for the HTTP request.

Usage

Creates an `com.HTTPRequest` object by providing a mandatory URL with HTTP or HTTPS as protocol.

The *url* parameter can be an identifier of an URL mapping with an optional `alias://` prefix. See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.doDataRequest

Performs the request by sending binary data.

Syntax

```
doDataRequest(
   data BYTE )
```

1. *data* is the binary data.

Usage

Performs the request by sending binary data contained in the `BYTE` variable.

Supported methods are PUT and POST.

The `BYTE` must be located in memory and not `NULL` otherwise operation fails.

The default Content-Type header is `application/octet-stream`, but it can be changed to any other mime type. For example: `image/jpeg`.

In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.doFileRequest

Performs the request by sending data contained in a file.

Syntax

```
doFileRequest(
   filepath STRING )
```

1. *filepath* is the path to the file containing the data to be send.

Usage

Performs the request by sending data contained in the file passed as parameter. The data is sent as is without any further conversion.

Supported methods are PUT and POST.

If not defined by programmer, the HTTP headers are automatically set as follows:

- `Content-Type` is defined according to the file name extension. If the file extension is not recognized, `Content-Type` defaults to `application/octet-stream`.

  Note: File extensions to `Content-Type` mapping can be customized in the file `FGLDIR/lib/wse/mime.cfg`.
- `Content-Disposition` is set with the base name of the given *filename* as follows: `attachment; filename="`*`basename`*`"`.

For example, when calling the method as follows:

```
CALL request.doFileRequest( "/opt/myapp/resources/logo.jpg" )
```

The resulting HTTP headers of the POST or PUT will look like:

```
Content-Type: image/jpeg
Content-Disposition: attachment; filename="logo.jpg"
```

In HTTP 1.1, if the body size is greater than 32k, the request will be sent in several chunks of the same size.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.doFormEncodedRequest

Performs an "application/x-www-form-urlencoded forms" encoded query.

Syntax

```
doFormEncodedRequest(
   query STRING,
   utf8 BOOLEAN )
```

1. *query* is a list of name/value pairs separated by an &.
2. *utf8* defines if the query string is UTF-8 encoded.

Usage

The `doFormEncodedRequest()` metho performs request with an "application/x-www-form-urlencoded forms" encoded query.

Supported methods are GET and POST.

The *query* string is a list of name/value pairs separated by an ampersand (&). For example:

```
name1=value1&name2=value2&name3=value3
```

> **Note:** If you need to URL-encode the separator characters & and =, double them as following : `na&&me=va==lue`.

If the *utf8* parameter is TRUE, the query string is encoded in UTF-8 as specified in XForms1.0, otherwise in ASCII as specified in HTML4.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.doRequest

Performs the HTTP request.

Syntax

```
doRequest()
```

Usage

The `doRequest()` method performs the HTTP request.

Supported methods are GET, HEAD and DELETE.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going

on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.doTextRequest

Performs the request by sending an entire string at once.

Syntax

```
doTextRequest(
   data STRING )
```

1. *data* is a string containing the request.

Usage

Performs the request by sending an entire string at once.

Supported methods are PUT and POST.

The default Content-Type header is `text/plain`, but it can be changed if of the form `*/*`. For example: `application/json`.

Automatic character set conversion from the application locale to the user-defined charset is performed. In case of conversion error, the method throws an exception.

> **Note:** To avoid character conversion problems when sending text over HTTP, consider setting the same user-define character set as the program defined by the application locale (assuming that the server understands the client application character set).

In HTTP 1.1, if the body size is greater than 32 KB, the request will be sent in several chunks of the same size.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.doXmlRequest

Performs the request by sending an entire XML document at once.

Syntax

```
doXmlRequest(
   data xml.DomDocument )
```

1. *data* is the XML document containing the data to be sent..

Usage

The `doXmlRequest()` method performs the request by sending the entire passed `xml.DomDocument` at once.

Supported methods are PUT and POST.

The default Content-Type header is `text/xml`, but it can be changed if of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In HTTP 1.1, if the body size is greater than 32 KB, the request will be sent in several chunks of the same size.

The character set used to send the XML data is defined by the encoding attribute in the XML document prolog. It is recommended that you define the HTTP request character set to `NULL` with the `setCharSet()` method, or that you use the same character set that was set in the XML Document.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.endXmlRequest

Terminates a streaming HTTP request.

Syntax

```
endXmlRequest(
   writer xml.StaxWriter )
```

1. *writer* is the `xml.StaxWriter` used to write the HTTP request.

Usage

The `endXmlRequest()` method terminates a streaming HTTP request by closing the `xml.StaxWriter` object that was created with the `beginXmlRequest()` method.

This HTTP request method is non-blocking: It returns immediatly after the call. Use the com.HTTPRequest.getResponse on page 2062 method, to perform a synchroneous HTTP request, suspending the program flow until the response returns from the server. If the program must keep going on, use the com.HTTPRequest.getAsyncResponse on page 2061 method, to check if a response is available.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.getAsyncResponse

When available, returns the response produced by one of request methods.

Syntax

```
getAsyncResponse()
  RETURNING result com.HTTPResponse
```

Usage

If a response is available, the `getAsyncResponse()` method returns a `com.HTTPResponse` object corresponding to the response that was produced by a call to one of the request methods: `doRequest()`, doTextRequest(), `doXmlRequest()`, `doFormEncodedRequest()`, or `beginXmlRequest()` and `endXmlRequest()`.

Unlike `getResponse()`, the `getAsyncResponse()` method is non-blocking: it returns immediatly and does not stop the program flow when waiting for a response.

The method returns `NULL` if the HTTP response was not yet received from the server.

This method is typically called just after a `do*Request()` call, and if the returned value is `NULL`, it is called again after a short period of time, to check for a response. Within a dialog, use an `ON IDLE` block to issue a `getAsyncRequest()` every seconds for example.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.getResponse

Waits and returns the response produced by one of request methods.

Syntax

```
getResponse()
  RETURNING result com.HTTPResponse
```

Usage

The `getResponse()` method wait for a resonse from the server and returns a `com.HTTPResponse` object corresponding to the response that was produced by a call to one of the request methods: `doRequest()`, `doTextRequest()`, `doXmlRequest()`, `doFormEncodedRequest()`, or `beginXmlRequest()` and `endXmlRequest()`.

> **Note:** On iOS, a long running HTTP request will display a message box, to let the user cancel the request. If the user cancels the HTTP request, the error -15578 will be raised. This error can be trapped with `TRY/CATCH`.

Unlike `getAsyncResponse()`, the `getResponse()` method is blocking: it stops the program flow until the HTTP response is received from the server.

Define a response timeout with the com.HTTPRequest.setTimeOut on page 2067 method.

> **Note:** On iOS devices, when using this method, it is not possible to distinguish different timeouts for the connection and for read/write operation, defined respectively by the `setConnectionTimeOut()` and `setTimeOut()` methods. If both timeouts are defined, the shortest timeout will be used for the connection and read/write operations.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPRequest.removeHeader

Removes an HTTP header for the request according to a name.

Syntax

```
removeHeader(
   name STRING )
```

1. *name* is the HTTP header name to remove.

Usage

The `removeHeader()` method deletes an HTTP header identified by *name*.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setAuthentication

Defines the user login and password to authenticate to the server.

Syntax

```
setAuthentication(
   login STRING,
   pass STRING,
   scheme STRING,
   realm STRING )
```

1. *login* is the login name.
2. *pass* is the password.
3. *scheme* defines the method to be used during authentication.
4. *realm* defines the realm.

Usage

The `setAuthentication()` method defines the mandatory user login and password to authenticate to the server.

The *scheme* parameter defines the method to be used during authentication. The supported values for the *scheme* parameter are `Anonymous`, `Basic` and `Digest`. The default is `Anonymous`.

An optional *realm* can be specified.

With `Anonymous` or `Digest` authentication, you must re-send the request if you get a 401 or 407 HTTP return code (authorization required)

If a user-defined authentication is set and there is an authenticate entry for this URL in the FGLPROFILE file, the user-defined authentication has priority.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setAutoReply

Defines the auto reply option for response methods.

Syntax

```
setAutoReply(
   reply BOOLEAN )
```

1. *reply* defines auto-reply when `TRUE`.

Usage

The `setAutoReply()` method defines whether `getResponse()` or `getAsyncResponse()` will automatically perform another HTTP GET request if response contains HTTP Authentication, Proxy Authentication or HTTP redirect data.

Available for GET method and the HTTP HEAD method.

The default is `TRUE`.

> **Important:** On iOS devices, `setAutoReply()` is ignored for redirection in synchronous requests: The iOS HTTP stack does not allow to set an auto reply option when doing synchronous requests.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setCharset

Defines the charset used when sending text or XML.

Syntax

```
setCharset(
   charset STRING )
```

1. *charset* is the character set to use.

Usage

Defines the character set used when sending an HTTP request.

By default, no character set information will be transmitted in the HTTP header. This is also the case when specifying `NULL` as parameter for this method.

If no character set is specified in HTTP headers, ISO8859-1 will implicitely be used as defined by the HTTP standards.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setConnectionTimeOut

Defines the timeout for the establishment of the connection.

Syntax

```
setConnectionTimeOut(
   timeout INTEGER )
```

1. *timeout* is the number of seconds.

Usage

The `setConnectionTimeOut()` method sets the time value in seconds to wait for the establishment of the connection, before a break.

The value of -1 means infinite wait.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setHeader

Sets an HTTP header for the request.

Syntax

```
setHeader(
   name STRING,
   value STRING )
```

1. *name* is the HTTP header name.
2. *value* is the HTTP header value.

Usage

The `setHeader()` method defines an HTTP header with a *name* and *value* for the request.

If a header exists with the same name, it is replaced with the new value.

Setting a header after the body has been sent, or if a streaming operation has been started, will only be taken into account when a new request is reissued.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setKeepConnection

Defines if connection is kept open if a new request occurs.

Syntax

```
setKeepConnection(
   keep BOOLEAN )
```

1. *keep* defines if the connection is kept.

Usage

The `setKeepConnection()` method defines whether the connection should stay open if a new HTTP request occurs.

The default is `FALSE`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setMaximumResponseLength

Defines the maximum size in Kbyte of a response.

Syntax

```
setMaximumResponseLength(
   length INTEGER )
```

1. *length* is the maximum size in Kbytes.

Usage

The `setMaximumResponseLength()` method sets the maximum authorized size in Kbytes of the whole response (including headers, body and all control characters), before a break.

The value of -1 means no limit.

> **Note:** Setting the maximum response length is ignored for synchronous requests in a Genero Mobile for iOS (GMI) app: The iOS HTTP stack does not allow you to set a maximum response length when doing synchronous requests.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setMethod

Sets the HTTP method of the request.

Syntax

```
setMethod(
   method STRING )
```

1. *method* is the HTTP method of the request.

Usage

The `setMethod()` method defines the HTTP method of the request.

Supported methods are GET, PUT, POST, HEAD and DELETE.

The default is GET.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setMultipartType

Switch HTTPRequest in multipart mode of given type.

Syntax

```
setMultipartType(
   type STRING,
   start STRING,
   boundary STRING )
```

1. *type* is one of the following:

   - form-data: Browser Xform with attachment
   - mixed: Parts are independent
   - related: Parts are dependent (Required for SOAP)

- alternative: Parts are different type of a same document
- *or any other type*
- NULL: switch multipart mode off

2. *start* is the Content-ID value of root multipart document. (optional)

3. *boundary* is the string used as multipart boundary. (optional)

Usage

Switch HTTPRequest in multipart mode of given type. Calling one of the standard request method will send the HTTP request as given multipart type, even if no other part has been set.

> **Important:** Multipart HTTP requests is not supported on GMI mobile devices.

The root HTTP part is the part handled via the standard HTTPRequest methods such as `doTextRequest()`, `doXmlRequest()`, `doDataRequest()` and `beginXmlRequest()`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setTimeOut

Defines the timeout for a reading or writing operation.

Syntax

```
setTimeOut(
   timeout INTEGER )
```

1. *timeout* specifies the number of seconds.

Usage

The `setTimeOut()` method defines a delay in seconds, to wait for a HTTP request read or write operation. If the operation is not terminated after the timeout, it returns immediately with an error.

Use the value of -1 to define an infinite timeout.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPRequest.setVersion

Sets the HTTP version of the request.

Syntax

```
setVersion(
   version STRING )
```

1. *version* is the HTTP version of the request.

Usage

The `setVersion()` method defines the HTTP version of the request.

Accepted versions are 1.0 and 1.1 (only these two versions are supported).

The default is 1.1.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Examples

Examples using methods of the com.HTTPRequest class.

Example 1: HTTP GET request

```
IMPORT com

MAIN
  DEFINE req com.HTTPRequest
  DEFINE resp com.HTTPResponse
  TRY
    LET req = com.HTTPRequest.Create("http://localhost:8090/MyPage")
    # Set additional HTTP header with name 'MyHeader', and value 'High
 Priority'
    CALL req.setHeader("MyHeader","High Priority")
    CALL req.doRequest()
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
      DISPLAY "HTTP Error ("||resp.getStatusCode()||") ",
        resp.getStatusDescription()
    ELSE
      DISPLAY "HTTP Response is : ",resp.getTextResponse()
    END IF
  CATCH
    DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
  END TRY
END MAIN
```

Example 2 : XForms HTTP POST request

```
IMPORT com
IMPORT xml

MAIN
  DEFINE req com.HTTPRequest
  DEFINE resp com.HTTPResponse
  DEFINE doc xml.DomDocument
  TRY
    LET req = com.HTTPRequest.Create("http://localhost:8090/MyProcess")
    CALL req.setMethod("POST") # Perform an HTTP POST method
      # Param1 value is 'hello', Param2 value is 'how are you ?'
    CALL req.doFormEncodedRequest("Param1=hello&Param2=how are
 you ?",FALSE)
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
      DISPLAY  "HTTP Error ("||resp.getStatusCode()||") ",
        resp.getStatusDescription()
    ELSE
       # Expect a returned content type of the form */xml
      LET   doc = resp.getXmlResponse()
      DISPLAY   "HTTP XML Response is : ",doc.saveToString()
    END IF
  CATCH
    DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
  END TRY
END MAIN
```

Example 3 : Streaming HTTP PUT request

```
IMPORT com
IMPORT xml

MAIN
  DEFINE req com.HTTPRequest
  DEFINE resp com.HTTPResponse
  DEFINE writer xml.StaxWriter
  TRY
    LET req = com.HTTPRequest.Create("http://localhost:8090/MyXmlProcess")
    CALL req.setMethod("PUT") # Perform an HTTP PUT method
    CALL req.setHeader("MyHeader","Value of my header")
      # Retrieve an xml.StaxWriter to start xml streaming
    LET writer = req.beginXmlRequest()
    CALL writer.startDocument("utf-8","1.0",true)
    CALL writer.comment("My first XML document sent in streaming with
 genero")
    CALL writer.startElement("root")
    CALL writer.attribute("attr1","value1")
    CALL writer.endElement()
    CALL writer.endDocument()
    CALL req.endXmlRequest(writer) # End streaming request
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 201 OR resp.getStatusCode() != 204 THEN
      DISPLAY   "HTTP Error ("||resp.getStatusCode()||") ",
        resp.getStatusDescription()
    ELSE
      DISPLAY   "XML document was correctly put on the server"
    END IF
  CATCH
    DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
  END TRY
END MAIN
```

Example 4 : Asynchronous HTTP DELETE request

```
IMPORT com

MAIN
  DEFINE req com.HTTPRequest
  DEFINE resp com.HTTPResponse
  DEFINE url STRING
  DEFINE quit CHAR(1)
  DEFINE questionStr STRING
  DEFINE timeout INTEGER
  TRY
    WHILE TRUE
      PROMPT "Enter http url you want to delete ? "
         FOR url ATTRIBUTES (CANCEL=FALSE)
      LET req = com.HTTPRequest.Create(url)
      CALL req.setMethod("DELETE")
      CALL req.doRequest()
      # Retrieve asynchronous response for the first time
      LET resp = req.getAsyncResponse()
      CALL Update(resp) RETURNING questionStr,timeout
      WHILE quit IS NULL OR ( quit!="Y" AND quit!="N" )
        PROMPT questionStr FOR CHAR quit
          ATTRIBUTES (CANCEL=FALSE,ACCEPT=FALSE,SHIFT="up")
          ON IDLE timeout
          IF resp IS NULL THEN # If no response at first try,
                               # retrieve it again
            LET resp = req.getAsyncResponse() # as we now have time
```

```
            CALL Update(resp) RETURNING questionStr,timeout
          END IF
        END PROMPT
      END WHILE
      IF quit == "Y" THEN
        EXIT PROGRAM
      ELSE
        LET quit = NULL
      END IF
    END WHILE
  CATCH
    DISPLAY "ERROR :",STATUS,SQLCA.SQLERRM
  END TRY
END MAIN

FUNCTION Update(resp)
  DEFINE resp com.HTTPResponse
  DEFINE ret STRING
  IF resp IS NOT NULL THEN
    IF resp.getStatusCode() != 204 THEN
    LET   ret = "HTTP Error ("||resp.getStatusCode()||")
      :"||resp.getStatusDescription()||". Do you want to quit ? "
    ELSE
      LET   ret = "HTTP Page deleted. Do you want to quit ? "
    END IF
  RETURN ret, 0
  ELSE
    LET ret = "Do you want to quit ? "
    RETURN ret, 1
  END IF
END FUNCTION
```

The HTTPResponse class

The `com.HTTPResponse` class provides an interface to perform XML and TEXT responses over HTTP, with additional XML streaming possibilities, on the client side.

The `STATUS` variable is set to zero after a successful method call.

com.HTTPResponse methods

Methods for the `com.HTTPResponse` class.

Table 467: Object methods

Name	Description
`beginXmlResponse()` `  RETURNING` *`writer`* `xml.StaxWriter`	Starts a streaming HTTP response.
`endXmlResponse(` `  ` *`writer`* `xml.StaxWriter )`	Performs the HTTP request.
`getDataResponse(` `  ` *`data`* `BYTE )`	Returns the entire HTTP response in a BYTE.
`getFileResponse( )`	Returns the entire HTTP response in a file on the disk.

Name	Description
RETURNING *filename* STRING	
getHeader(*name* STRING) RETURNING *result* STRING	Returns the value of an HTTP header.
getHeaderCount() RETURNING *result* INTEGER	Returns the number of headers.
getHeaderName(*index* INTEGER) RETURNING *result* STRING	Returns the name of a header by position.
getHeaderValue(*index* INTEGER) RETURNING *result* STRING	Returns the value of a header by position.
getStatusCode() RETURNING *result* INTEGER	Returns the HTTP status code.
getStatusDescription() RETURNING *result* STRING	Returns the HTTP status description.
getTextResponse() RETURNING *data* STRING	Returns the entire HTTP response in a string.
getXmlResponse() RETURNING *data* xml.DomDocument	Returns the entire HTTP response in a DOM document.

Table 468: Object methods: Multipart methods

Name	Description
getMultipartType() RETURNING *result* STRING	Returns whether a response is multipart or not, and the kind of multipart if any.
getPart(*index* INTEGER) RETURNING *part-object* com.HTTPPart	Returns the HTTP part object at the specified index of the current HTTP response.
getPartCount() RETURNING *count* INTEGER	Returns the number of additional parts in the HTTP response.
getPartFromContentID(*id* STRING)	Returns the HTTP part object marked with the given Content-ID value as identifier, or NULL if none.

Name	Description
RETURNING *part-object* com.HTTPPart	

com.HTTPResponse.beginXmlResponse

Starts a streaming HTTP response.

Syntax

```
beginXmlResponse()
  RETURNING writer xml.StaxWriter
```

1. *writer* is the `xml.StaxWriter` to be used to write the HTTP request.

Usage

The `beginXmlResponse()` method the streaming HTTP response and returns a `xml.StaxReader` object ready to read XML from the server.

The Content-Type header must be of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPResponse.endXmlResponse

Performs the HTTP request.

Syntax

```
endXmlResponse(
   writer xml.StaxWriter )
```

1. *writer* is the `xml.StaxWriter` used to write the HTTP response.

Usage

The `endXmlResponse()` method ends the streaming HTTP response by closing the the `xml.StaxWriter` object that was created with the `beginXmlResponse()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPResponse.getDataResponse

Returns the entire HTTP response in a BYTE.

Syntax

```
getDataResponse(
   data BYTE )
```

1. *data* is a `BYTE` variable receiving the HTTP response data.

Usage

The `getDataResponse()` method returns the body of an HTTP response into a BYTE variable.

The BYTE variable must be located in memory, otherwise operation fails.

Returns binary data as response from a server into a BYTE.

Previous content is discarded.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPResponse.getFileResponse

Returns the entire HTTP response in a file on the disk.

Syntax

```
getFileResponse( )
   RETURNING filename STRING
```

1. *filename* is the absolute path to the file containing the HTTP response.

Usage

Reads an HTTP response and creates a file from it.

The file is created in the temporary directory used by the runtime system (DBTEMP). The name of the file will be the basename found in the HTTP Content-Disposition Header, if this basename is not specified, the filename will be created with a UUID. If a file with the same name already exists in the temporary directory, the API prefixes the new file with a number. It is then of the form : `/tmp/ABC/filename_index.ext`, where *index* represents the number of files with the same name on disk.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPResponse.getHeader

Returns the value of an HTTP header.

Syntax

```
getHeader( name STRING )
  RETURNING result STRING
```

1. *name* is the name of the HTTP header.

Usage

The `getHeader()` method returns the value of the HTTP header specified by the *name* parameter, or `NULL` if not found.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getHeaderCount

Returns the number of headers.

Syntax

```
getHeaderCount()
  RETURNING result INTEGER
```

Usage

The `getHeaderCount()` method returns the number of headers of the HTTP response.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getHeaderName

Returns the name of a header by position.

Syntax

```
getHeaderName(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the ordinal position of the header.

Usage

The `getHeaderName()` method returns the name of the HTTP response header according to the position passed as parameter.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getHeaderValue

Returns the value of a header by position.

Syntax

```
getHeaderValue(
   index INTEGER )
  RETURNING result STRING
```

1. *index* is the ordinal position of the header.

Usage

The `getHeaderValue()` method returns the value of the HTTP response header according to the position passed as parameter.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getMultipartType

Returns whether a response is multipart or not, and the kind of multipart if any.

Syntax

```
getMultipartType()
  RETURNING result STRING
```

Usage

Returns whether a response is multipart or not, and the kind of multipart if any.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getPartCount

Returns the number of additional parts in the HTTP response.

Syntax

```
getPartCount()
  RETURNING count INTEGER
```

Usage

Returns the number of additional parts in the HTTP response. The root part element must be handled via `getXmlResponse()`, `getTextResponse()`, `getDataResponse()` and `beginXmlResponse()`. In other words, there are `getPartCount()` +1 parts if `getMultipartType()` does not return NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getPart

Returns the HTTP part object at the specified index of the current HTTP response.

Syntax

```
getPart(
   index INTEGER )
  RETURNING part-object com.HTTPPart
```

1. *index* is the index number.

Usage

Returns the HTTP part object at the specified index of the current HTTP response.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Can raise error -15554 (Index is out of bounds).

com.HTTPResponse.getPartFromContentID

Returns the HTTP part object marked with the given Content-ID value as identifier, or NULL if none.

Syntax

```
getPartFromContentID(
   id STRING )
  RETURNING part-object com.HTTPPart
```

1. *name* is the name of the HTTP header.

Usage

Returns the HTTP part object marked with the given Content-ID value as identifier, or NULL if none.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getStatusCode

Returns the HTTP status code.

Syntax

```
getStatusCode()
  RETURNING result INTEGER
```

Usage

The `getStatusCode()` method returns the status code for the HTTP response.

When the returned HTTP status code is 401 or 407, authorization is required.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getStatusDescription

Returns the HTTP status description.

Syntax

```
getStatusDescription()
  RETURNING result STRING
```

Usage

The `getStatusDescription()` method returns a description of the HTTP reponse status.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPResponse.getTextResponse

Returns the entire HTTP response in a string.

Syntax

```
getTextResponse()
  RETURNING data STRING
```

Usage

The `getTextResponse()` method returns a HTTP reponse as a entire string.

- The Content-Type header can be of the form ***/***. For example: application/json.
- Automatic conversion to the locale charset is performed when possible, otherwise throws an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.HTTPResponse.getXmlResponse

Returns the entire HTTP response in a DOM document.

Syntax

```
getXmlResponse()
  RETURNING data xml.DomDocument
```

Usage

The `getXmlResponse()` method returns an HTTP response in a `xml.DomDocument` object.

The Content-Type header must be of the form `*/xml` or `*/*+xml`. For example: `application/xhtml+xml`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

Examples

Examples using methods of the `com.HTTPResponse` class.

For examples, see Examples on page 2068.

The HTTPPart class

The `com.HTTPPart` class provides an interface to manage the HTTP attachment sent or received in HTTP.

The `STATUS` variable is set to zero after a successful method call.

com.HTTPPart methods

Methods for the `com.HTTPPart` class.

Table 469: Class methods of com.HTTPPart

Name	Description

`CreateAttachment(`
`  filename STRING)`
`RETURNING part-object com.HTTPPart`
	Creates a new HTTPPart object based on given filename located on disk.

`CreateFromData(` `  b BYTE)`	Creates a new HTTPPart object based on given BYTE located in memory.

Name	Description
`RETURNING part-object com.HTTPPart`	
`CreateFromDomDocument( x xml.DomDocument) RETURNING part-object com.HTTPPart`	Creates a new HTTPPart object based on given XML document.
`CreateFromString( s STRING) RETURNING part-object com.HTTPPart`	Creates a new HTTPPart object based on given string.

Table 470: Object methods of com.HTTPPart

Name	Description
`getAttachment() RETURNING filename STRING`	Returns the absolute path to the HTTP part.
`getContentAsData( b BYTE )`	Returns the HTTP part as a BYTE.
`getContentAsDomDocument() RETURNING domDocument xml.DomDocument`	Returns the HTTP part as a XML document.
`getHeader( name STRING ) RETURNING value STRING`	Setter to handle HTTP multipart headers.
`getContentAsString() RETURNING str STRING`	Returns the HTTP part as a string.
`setHeader( name STRING, value STRING)`	Setter to handle HTTP multipart headers.

com.HTTPPart.CreateFromString

Creates a new HTTPPart object based on given string.

Syntax

```
CreateFromString(
   s STRING)
  RETURNING part-object com.HTTPPart
```

1. *s* is a string

Usage

Creates a new HTTPPart object based on given string. To be used via the addPart() method.

Defaults HTTP multipart headers:

- Content-Type: text/plain
- Content-Transfer-Encoding: 8bits

Notice that the string will be converted during request sending into ISO-8859-1 by default, unless a different charset has been set via setHeader("Content-Type","text/plain; charset=UTF-8") for instance.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPPart.CreateFromDomDocument

Creates a new HTTPPart object based on given XML document.

Syntax

```
CreateFromDomDocument(
   x xml.DomDocument)
  RETURNING part-object com.HTTPPart
```

1. *x* is an XML document.

Usage

Creates a new HTTPPart object based on given XML document. To be used via the addPart() method.

Defaults HTTP multipart headers:

- Content-Type: text/xml; charset=UTF-8
- Content-Transfer-Encoding: 8bits

A different charset can be set with the setHeader method. For example, `setHeader("Content-Type","text/plain; charset=ISO-8859-1")` sets the charset to ISO-8859-1.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPPart.CreateFromData

Creates a new HTTPPart object based on given BYTE located in memory.

Syntax

```
CreateFromData(
   b BYTE)
  RETURNING part-object com.HTTPPart
```

1. *b* is a BYTE object located in memory.

Usage

Creates a new HTTPPart object based on given BYTE located in memory. To be used via the addPart() method.

Defaults HTTP headers:

- Content-Type: application/octet-stream
- Content-Transfer-Encoding: base64

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPPart.CreateAttachment

Creates a new HTTPPart object based on given filename located on disk.

Syntax

```
CreateAttachment(
   filename STRING)
  RETURNING part-object com.HTTPPart
```

1. *filename* is the name of a file.

Usage

Creates a new HTTPPart object based on given filename located on disk. To be used via the addPart() method.

The com.HTTPPart.CreateAttachment() method automatically sets the following headers for the created HTTPPart object:

- Content-Type is defined according to the file name extension. If the file extension is not recognized, Content-Type defaults to application/octet-stream.

  Note: File extensions to Content-Type mapping can be customized in the file FGLDIR/lib/wse/mime.cfg.
- Content-Transfer-Encoding is set to "binary".
- Content-Disposition is set with the base name of the given *filename* as follows: attachment; filename="*basename*".

For example, when calling the method as follows:

```
LET part = com.HTTPPart.CreateAttachment( "/opt/myapp/resources/logo.jpg" )
```

The resulting HTTP part headers will look like:

```
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Disposition: attachment; filename="logo.jpg"
```

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPPart.getAttachment

Returns the absolute path to the HTTP part.

Syntax

```
getAttachment()
  RETURNING filename STRING
```

Usage

Returns the absolute path location of the received part file.

The file is created in the temporary directory used by the runtime system (DBTEMP). The name of the file will be the basename found in the HTTP Content-Disposition Header, if this basename is not specified, the

filename will be created with a UUID. If a file with the same name already exists in the temporary directory, the API prefixes the new file with a number. It is then of the form : `/tmp/ABC/filename_`*`index`*`.ext`, where *index* represents the number of files with the same name on disk.

If the file is encoded in base64, you can use the Genero Web Services `fglpass -dec64` command to convert it back to binary.

It is up to programmer to remove file from the disk when it is no longer needed.

To be used via methods: com.HTTPResponse.getPart on page 2075, com.HTTPResponse.getPartCount on page 2075, and com.HTTPResponse.getPartFromContentID on page 2075

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.HTTPPart.getContentAsData

Returns the HTTP part as a `BYTE`.

Syntax

```
getContentAsData(
   b BYTE )
```

1. *b* is a variable holding the `BYTE` data.

Usage

Returns the HTTP part as a `BYTE`. `BYTE` data cannot be returned from a function with a `RETURN` statement. Therefore, the `BYTE` parameter must be handled by reference.

To be used via methods: com.HTTPResponse.getPart on page 2075, com.HTTPResponse.getPartCount on page 2075, and com.HTTPResponse.getPartFromContentID on page 2075

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The error -15573 is raised if the part cannot be converted to a Genero `BYTE`.

com.HTTPPart.getContentAsDomDocument

Returns the HTTP part as a XML document.

Syntax

```
getContentAsDomDocument()
  RETURNING domDocument xml.DomDocument
```

Usage

Returns the HTTP part as a XML document.

To be used via methods: com.HTTPResponse.getPart on page 2075, com.HTTPResponse.getPartCount on page 2075, and com.HTTPResponse.getPartFromContentID on page 2075

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The error -15573 is raised if the part cannot be converted to a XML DomDocument.

com.HTTPPart.getContentAsString

Returns the HTTP part as a string.

Syntax

```
getContentAsString()
  RETURNING str STRING
```

Usage

Returns the HTTP part as a string.

To be used via methods: com.HTTPResponse.getPart on page 2075, com.HTTPResponse.getPartCount on page 2075, and com.HTTPResponse.getPartFromContentID on page 2075

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The error -15573 is raised if the part cannot be converted to a Genero string or if the charset is not supported.

com.HTTPPart.getHeader

Setter to handle HTTP multipart headers.

Syntax

```
getHeader(
   name STRING )
  RETURNING value STRING
```

1. *name* is the name of the header part.
2. *value* is the value for the header part specified by *name*.

Usage

Getter to handle HTTP multipart headers.

> **Note:** In case of related multipart (i.e., the part is multipart/related and set via the com.HTTPRequest.setMultipartType("related",NULL,NULL)), it is mandatory to set a unique Content-ID header. To set up a unique Content-ID header, you can use the security.RandomGenerator.CreateUUIDString on page 2280 method for that.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Example

```
LET val = req.getHeader("MyClientHeader")
```

com.HTTPPart.setHeader

Setter to handle HTTP multipart headers.

Syntax

```
setHeader(
   name STRING,
   value STRING)
```

1. *name* is the multipart header name.
2. *value* is the multipart header value (such as HTTP headers).

Usage

Setter to handle HTTP multipart headers.

For instance, when you send a multipart image, you should specify the image mime type with this header method. If the image is a `png`, you have to do `part.setHeader("Content-Type","image/png")`, which lets the peer know the format of the attached file it has to process.

> **Note:** In case of related multipart (i.e., the part is multipart/related and set via the `com.HTTPRequest.setMultipartType("related",NULL,NULL)`), it is mandatory to set a unique Content-ID header. To set up a unique Content-ID header, you can use the security.RandomGenerator.CreateUUIDString on page 2280 method for that.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

Example

```
CALL req.setHeader("MyClientHeader","Hello")
```

Example
Examples using methods of the `com.HTTPPart` class.

This example consists of two applications: a client and server exchanging an XML document in multipart with a image as an attachment.

Client Application

```
IMPORT com
IMPORT xml

CONSTANT SERVER_URL = "http://localhost:8090/MultipartMixed/Sample"


MAIN

  DEFINE req      com.HTTPRequest
  DEFINE resp     com.HTTPResponse
  DEFINE doc      xml.DomDocument
  DEFINE root     xml.DomNode
  DEFINE p        com.HTTPPart
  DEFINE type     STRING
  DEFINE count    INTEGER
  DEFINE ind      INTEGER

  LET req = com.HTTPRequest.Create(SERVER_URL)
  CALL req.setMethod("POST")
  CALL req.setHeader("MyClientHeader","Hello")
  TRY
    # Set multipart type
    CALL req.setMultipartType("mixed",NULL,NULL)
    # Add filename as part
    LET p = com.HTTPPart.CreateAttachment("my_picture.png")
    # Set attachment Content-Type
    CALL p.setHeader("Content-Type","image/png")
    # Add part to the request
    CALL req.addPart(p)
```

```
    # Perform XML request
    LET doc = xml.DomDocument.CreateDocument("MyXmlDocument")
    CALL req.doXmlRequest(doc)
    # Check response
    LET resp=req.getResponse()
    IF resp.getStatusCode() != 200 THEN
      DISPLAY  "HTTP Error ("||resp.getStatusCode()||") ",
        resp.getStatusDescription()
      EXIT PROGRAM (-1)
    END IF
    IF resp.getStatusDescription() != "OK" THEN
      DISPLAY  "HTTP Error ("||resp.getStatusCode()||") ",
        resp.getStatusDescription()
      EXIT PROGRAM (-1)
    END IF
    # Check whether multipart response or not
    LET type = resp.getMultipartType()
    IF type IS NULL THEN
      DISPLAY  "Failed : Expected multipart in response"
      EXIT PROGRAM (-1)
    ELSE
      DISPLAY "Response is multipart of :",type
    END IF
    # Check response
    LET doc = resp.getXmlResponse()
    IF doc IS NULL THEN
      DISPLAY  "Expected XML document as response"
      EXIT PROGRAM (-1)
    ELSE
      DISPLAY "Response is : ",doc.saveToString()
    END IF
    # Process additional parts
    FOR ind = 1 TO resp.getPartCount()
      LET p = resp.getPart(ind)
      IF p.getAttachment() IS NOT NULL THEN
        DISPLAY "Attached file at :",p.getAttachment()
      ELSE
        DISPLAY "Attached part is :",p.getContentAsString()
      END IF
    END FOR
  CATCH
    DISPLAY "unexpected exception :",STATUS," ("||SQLCA.SQLERRM||")"

    EXIT PROGRAM (-1)
  END TRY
END MAIN
```

Server Application

```
IMPORT com
IMPORT xml

MAIN

  DEFINE req        com.HTTPServiceRequest
  DEFINE url        STRING
  DEFINE method     STRING
  DEFINE txt        STRING
  DEFINE doc        xml.DomDocument
  DEFINE type       STRING
  DEFINE ind        INTEGER
  DEFINE p          com.HTTPPart
```

```
  CALL com.WebServiceEngine.Start()

  LET req = com.WebServiceEngine.getHTTPServiceRequest(-1)
  LET url = req.getURL()
  IF url IS NULL  THEN
    DISPLAY "Failed: url should not be null"
    EXIT PROGRAM (-1)
  END IF
  LET method = req.getMethod()
  IF method IS NULL OR method != "POST" THEN
    DISPLAY "Failed: method should be POST"
    EXIT PROGRAM (-1)
  END IF
  # Check multipart type
  LET type = req.getRequestMultipartType()
  IF type IS NULL THEN
    DISPLAY "Failed: expected multipart in request"
    EXIT PROGRAM (-1)
  END IF
  TRY
    LET doc = req.readXMLRequest()
    DISPLAY "Request is :", doc.saveToString()
  CATCH
    DISPLAY "Failed: unexpected error :", STATUS
    EXIT PROGRAM (-1)
  END TRY
  # Process additional parts
  FOR ind = 1 TO req.getRequestPartCount()
    LET p = req.getRequestPart(ind)
    IF p.getAttachment() IS NOT NULL THEN
      DISPLAY "Attached file at :",p.getAttachment()
    ELSE
      DISPLAY "Attached part is :",p.getContentAsString()
    END IF
  END FOR
  # Set multipart response type
  CALL req.setResponseMultipartType("mixed",NULL,NULL)
  # Add XML Part
  LET p = com.HTTPPart.CreateAttachment("my_other_picture.jpg")
  CALL p.setHeader("Content-Type","image/jpg")
  CALL req.addResponsePart(p)
  LET doc = xml.DomDocument.CreateDocument("MyResponse")
  CALL req.sendXmlResponse(200,NULL,doc)
END MAIN
```

TCP classes
The TCP classes manage TCP client network operations.

- CLASS TCPRequest
- CLASS TCPResponse

The TCPRequest class
The `com.TCPRequest` class provides an interface to perform asynchronous XML and TEXT requests over TCP, with additional XML streaming possibilities.

> **Important:** This Web Services class is not supported on GMI mobile devices.

com.TCPRequest methods
Methods of the `com.TCPRequest` class.

Table 471: Class methods of com.TCPRequest

Name	Description

`com.TCPRequest.Create(`
`  url STRING )`
`  RETURNING result com.TCPRequest`
	Creates a new TCP request object.

Table 472: Object methods of com.TCPRequest

Name	Description
`beginXmlRequest()` `  RETURNING writer xml.StaxWriter`	Starts a streaming XML request.
`doDataRequest(` `  data BYTE )`	Performs the request by sending binary data.
`doRequest()`	Performs a TCP request.
`doTextRequest(` `  data STRING )`	Persforms a request with a string.
`doXmlRequest(` `  document xml.DomDocument )`	Persforms a request with a DOM document.
`endXmlRequest(` `  writer xml.StaxWriter )`	Terminates a streaming TCP request.
`getAsyncResponse()` `  RETURNING response com.TCPResponse`	Returns the response after performing a TCP request, asynchronously.
`getResponse()` `  RETURNING response com.TCPResponse`	Returns the response after performing a TCP request.
`setConnectionTimeOut(` `  seconds INTEGER )`	Defines the connection time out.
`setKeepConnection(` `  on BOOLEAN )`	Defines if the TCP connection is kept open after sending a request.
`setMaximumResponseLength(`	Defines the time out for read/write operations.

Name	Description
`length INTEGER )`	
`setTimeOut( seconds INTEGER )`	Defines the time out for read/write operations.

com.TCPRequest.beginXmlRequest

Starts a streaming XML request.

Syntax

```
beginXmlRequest()
  RETURNING writer xml.StaxWriter
```

Usage

The `beginXmlRequest()` method begins a streaming HTTP request and returns an `xml.StaxWriter` object ready to send XML to the server.

After sending all the XML data to the server, you must call the `endXmlRequest()` method with the `xml.StaxWriter` object created by the `beginXmlRequest()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPRequest.Create

Creates a new TCP request object.

Syntax

```
com.TCPRequest.Create(
   url STRING )
  RETURNING result com.TCPRequest
```

1. *url* is the URL of the TCP request.

Usage

This class method creates a new `com.TCPRequest` object according to the URL passed as parameter.

The URL must use the TCP or TCPS protocol. Examples of valid URLs include:

- `tcp://localhost:4242/`
- `tcps://localhost:4343/`

The URL can be an identifier of an URL mapping with an optional **alias://** prefix. See FGLPROFILE configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.TCPRequest.doDataRequest

Performs the request by sending binary data.

Syntax

```
doDataRequest(
   data BYTE )
```

1. *data* is the binary data to be send for a TCP request. The `BYTE` variable must be located `IN MEMORY`.

Usage

Performs the TCP request by sending binary data contained in the `BYTE` variable.

> **Note:** The `BYTE` variable must be located `IN MEMORY`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.TCPRequest.doRequest

Performs a TCP request.

Syntax

```
doRequest()
```

Usage

The `doRequest()` method performs the TCP request.

The connection is shutdown for writing, to notify that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

Example

```
IMPORT com
IMPORT XML

MAIN
  DEFINE url STRING
  LET url = "tcp://localhost:4242"
  CALL an_example(url)
END MAIN

FUNCTION an_example(url)
DEFINE url STRING
DEFINE req com.TCPRequest
DEFINE resp com.TCPResponse
DEFINE ret xml.DomDocument

TRY
  LET  req = com.TCPRequest.create(url)
  CALL req.doRequest()
  LET  resp = req.getResponse()
  LET  ret = resp.getXmlResponse()
```

```
CATCH
  DISPLAY "ERROR : ", STATUS, SQLCA.SQLERRM
  EXIT PROGRAM(-1)
END TRY

END FUNCTION
```

com.TCPRequest.doXmlRequest

Persforms a request with a DOM document.

Syntax

```
doXmlRequest(
   document xml.DomDocument )
```

1. *document* is the DOM document describing the request.

Usage

The `doXmlRequest()` method performs the TCP request by using the information defined in the `xml.DomDocument` object passed as parameter.

The connection is shutdown for writing, to notify that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPRequest.doTextRequest

Persforms a request with a string.

Syntax

```
doTextRequest(
   data STRING )
```

1. *data* is a string describing the request.

Usage

The `doTextRequest()` method performs the TCP request by using the information defined in string passed as parameter.

The connection is shutdown for writing, to notify that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPRequest.endXmlRequest

Terminates a streaming TCP request.

Syntax

```
endXmlRequest(
```

```
  writer xml.StaxWriter )
```

1. *writer* is the Stax writer object used for streaming.

Usage

The `endXmlRequest()` method terminates a streaming TCP request performed with the `xml.StaxWriter` object that what created by the `beginXmlRequest()` method.

The connection is shutdown for writing, to notify that no data will be sent.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPRequest.getResponse

Returns the response after performing a TCP request.

Syntax

```
getResponse()
  RETURNING response com.TCPResponse
```

Usage

The `getResponse()` method returns a TCP response as a `com.TCPResponse` object, after a call to `doRequest()`, `doXmlRequest()`, `doTextRequest()`, or `beginXmlRequest() / endXmlRequest()` calls.

A call to this methid will stop the program flow until the response is received.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPRequest.getAsyncResponse

Returns the response after performing a TCP request, asynchronously.

Syntax

```
getAsyncResponse()
  RETURNING response com.TCPResponse
```

Usage

The `getAsyncResponse()` method returns a TCP response as a `com.TCPResponse` object, after a call to `doRequest()`, `doXmlRequest()`, `doTextRequest()`, or `beginXmlRequest() / endXmlRequest()` calls.

Unlike `getResponse()`, the `getAsyncResponse()` method does not stop the program flow: The method returns `NULL` if the response was not yet received.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPRequest.setTimeOut

Defines the time out for read/write operations.

Syntax

```
setTimeOut(
   seconds INTEGER )
```

1. *seconds* is the time out in seconds.

Usage

This method defines the time value in seconds to wait for a reading or writing operation, before a break.

If the time out is -1, waits infinitely.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.TCPRequest.setConnectionTimeOut

Defines the connection time out.

Syntax

```
setConnectionTimeOut(
   seconds INTEGER )
```

1. *seconds* is the time out in seconds.

Usage

This method defines the time value in seconds to wait for a connection, before a break.

If the time out is -1, waits infinitely.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.TCPRequest.setKeepConnection

Defines if the TCP connection is kept open after sending a request.

Syntax

```
setKeepConnection(
   on BOOLEAN )
```

1. *on* indicates if the TCP connection must be kept open.

Usage

This method can be used to force the TCP socket to remain open after a send operation, in order to perform subsequent `do*Request()` calls, without closing the connection (in write mode).

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

com.TCPRequest.setMaximumResponseLength

Defines the time out for read/write operations.

Syntax

```
setMaximumResponseLength(
   length INTEGER )
```

1. *length* is the max size of a reponse, in Kbytes.

Usage

This method sets the maximum authorized size in Kbyte of the whole response, before a break.

A length of -1 defines no limit.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The TCPResponse class

The com.TCPResponse class provides an interface to perform XML and TEXT responses over TCP, with additional XML streaming possibilities.

> **Important:** This Web Services class is not supported on GMI mobile devices.

com.TCPResponse methods

Methods of the com.TCPResponse class.

Table 473: Object methods

Name	Description
`beginXmlResponse()` `  RETURNING reader xml.StaxReader`	Starts a streaming TCP response.
`endXmlResponse(` `  reader xml.StaxReader )`	Ends a streaming TCP response.
`getDataResponse(` `  data BYTE )`	Returns a TCP response in binary format.
`getTextResponse()` `  RETURNING result STRING`	Returns a TCP response in string format.
`getXmlResponse()` `  RETURNING doc xml.DomDocument`	Returns an entire DOM document as TCP response.

com.TCPResponse.beginXmlResponse

Starts a streaming TCP response.

Syntax

```
beginXmlResponse()
```

```
   RETURNING reader xml.StaxReader
```

Usage

Begins the streaming TCP response and returns an `xml.StaxReader` object ready to read XML from the server.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPResponse.endXmlResponse

Ends a streaming TCP response.

Syntax

```
endXmlResponse(
   reader xml.StaxReader )
```

1. *reader* is the STAX reader object created with `beginXmlResponse()`.

Usage

Terminates the streaming TCP response identified by the `xml.StaxReader` object passed as parameter. This object must have been created with the `beginXmlResponse()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPResponse.getDataResponse

Returns a TCP response in binary format.

Syntax

```
getDataResponse(
   data BYTE )
```

1. *data* is the `BYTE` variable that will hold the response data in binary format. The `BYTE` variable must be located `IN MEMORY`.

Usage

This method retrieves the TCP reponse in binary format into the `BYTE` variable passed as parameter. The method will read the TCP stream, until the peer closes the connection.

> **Note:** The `BYTE` variable must be located `IN MEMORY`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPResponse.getTextResponse
Returns a TCP response in string format.

Syntax

```
getTextResponse()
  RETURNING result STRING
```

Usage

This method returns a complete streaming TCP response from the server as a string.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

com.TCPResponse.getXmlResponse
Returns an entire DOM document as TCP response.

Syntax

```
getXmlResponse()
  RETURNING doc xml.DomDocument
```

Usage

This method returns a complete xml.DomDocument as streaming TCP response from the server.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The INT_FLAG variable is checked during GWS API call to handle program interruptions, for more details, see Interruption handling in GWS calls (INT_FLAG) on page 2546

Helper classes
The Helper classes provide utility classes.

- The Util class on page 2094

The Util class
The com.Util class provides static helper methods.

This class does not have to be instantiated.

> **Important:** This Web Services class is not supported on GMI mobile devices.

com.Util methods
Methods of the com.Util class.

Table 474: Class methods

Name	Description
com.Util.UniqueApplicationInstance(path STRING)	Checks that the calling application is the only one to run.

Name	Description
RETURNING *result* INTEGER	

com.Util.UniqueApplicationInstance

Checks that the calling application is the only one to run.

Syntax

```
com.Util.UniqueApplicationInstance(
   path STRING)
  RETURNING result INTEGER
```

1. *path* is the path to the lock file.

Usage

This method checks that the calling application is the only one to run, by trying to get an exclusive lock on the given file.

If the lock could be set, the method returns 0. Otherwise, returns 1 and updates STATUS with an error code.

Specific classes

Several classes support specific features.

- The APNS class on page 2095

The APNS class

The com.APNS class implements Apple Push Notification Service APIs.

The com.APNS class implements a set of methods to build and handle push messages to be broadcasted by the Apple Push Notification service.

APNs SSL certificate

Get and configure an SSL certificate to establish secure connections to the Apple Push Notification service.

Basics

The Apple Push Notification Certificate identifies the push notification service for a given mobile app. This certificate will be created from an App ID (a.k.a. Bundle ID) and is used by the APNs system to dispatch the notification message to the registered devices.

You can create two type of APNs certificates for a given App ID:

- Sandbox (for development and test purpose)
- Production (for deployment)

An APNS push notification provider or an APNS feedback handler needs to establish a secure connection to Apple's APNs server.

Get an APNs certificate for your app

In this section we will produce the **myapp.cer** file and **myapp-key.p12** file.

To create an Apple Push Notification Certificate:

1. Log to Apple's Member Center with you iOS developer or enterprise account,
2. Select Certificates, Identifiers & Profiles,
3. Under Certificates, select the + symbol,

4. Select Apple Push Notification service SSL (Sandbox) for development, or Apple Push Notification service SSL (Production) for production,
5. You need to choose the explicit app ID you want to use for push notifications. Verify before that you enable Push Notification Service for this app ID (go to App IDs section and edit your app ID),
6. Follow the instructions on the page to create a CSR file then click Continue,
7. Your certificate request is now available. Now you can go back to the Development Certificate section still active in your browser and click Choose file,
8. Navigate to the file you just saved and choose that file,
9. Click Generate,
10. Once the certificate is generated, click Download. The certificate will download into your Downloads folder, as a `.cer` file (for ex: **`myapp.cer`**).
11. Double-click this file to install it into Keychain,
12. When done, your new certificate should be listed in the Certificates list,
13. Open your Keychain app and locate the certificate you created, export the private key in p12 format (for ex **`myapp-key.p12`**). Note that you will be asked for a password to encode the `.p12` file, and for your session password, to exported Keychain files.

Configure Genero to use the APNs certificate

On the Genero push provider server, you will need the public certificate (`myapp.crt` file) and the private key (`myapp-key.pem` file) for you app. These files will be referenced in the `security.global.certificate` and `security.global.privatekey` entries of FGLPROFILE.

In order to authenticate the APNs server, you will also need the root certificate authority (`apple_entrust_root_certification_authority.pem`), that can be downloaded from Apple's web site. This file will be referenced by the `security.global.ca` entry in FGLPROFILE.

> **Note:** When executing on a Mac, the root certificate (`security.global.ca` entry) is not required: The Web Services library reads the Keystore of the Mac computer, to authenticate the APNs server.

Create the `myapp.crt` file (public certificate) from the `myapp.cer` file, with the `openssl x509` command:

```
$ openssl x509 -in myapp.cer -inform der -out myapp.crt
```

Convert the `myapp-key.p12` file (containing the private key) to a `myapp-key.pem` format, with the `openssl pkcs12` command:

```
$ openssl pkcs12 -nocerts -in myapp-key.p12 -out myapp-key.pem
```

> **Note:** You need to enter the passphrase for the `.p12` file so that `openssl` can read it. Then you need to enter a new passphrase that will be used to encrypt the `.pem` file.

FGLPROFILE entries require encrypted private key files, therefore you need to remove the passphrase from the `myapp-key.pem` file, with the `openssl rsa` command:

```
$ openssl rsa -in myapp-key.pem -out myapp-key-noenc.pem
```

Set up your FGLPROFILE with the appropriate `security.*` entries:

```
security.global.ca           =
 "apple_entrust_root_certification_authority.pem"
security.global.certificate  = "myapp.crt"
security.global.privatekey   = "myapp-key-noenc.pem"
```

In the above example:

1. `apple_entrust_root_certification_authority.pem` is the HTTPS root certificate authenticating the APNs server (if the computer is not a Mac).
2. `myapp.crt` is the public certificate for your app.
3. `myapp-key-noenc.pem` is the private key for your app.

> **Note:** If you want to keep the private key encrypted, you need to configure a password agent, as described in Using the password agent

com.APNS methods
Methods of the `com.APNS` class.

Table 475: Class methods

Name	Description
`com.APNS.DecodeError( data BYTE) RETURNING uuid STRING, error INTEGER`	Decodes content of BYTE data returned from the APNS server in case of error.
`com.APNS.DecodeFeedback( data BYTE, unregs DYNAMIC ARRAY OF RECORD timestamp INTEGER, deviceToken STRING END RECORD )`	Decodes content of BYTE data returned from the APNS feedback service.
`com.APNS.EncodeMessage( data BYTE, deviceToken STRING, json STRING, uuid STRING, expiration INTEGER, priority SMALLINT) RETURNING result INTEGER`	Encodes an APNS specific push notification message into a BYTE.

com.APNS.DecodeError

Decodes content of BYTE data returned from the APNS server in case of error.

Syntax

```
com.APNS.DecodeError(
   data BYTE)
  RETURNING uuid STRING, error INTEGER
```

1. *data* is the `BYTE` variable containing the error data. This `BYTE` variable must be located `IN MEMORY`.
2. *uuid* is a Base64 encoded string containing the push notification identifier.
3. *error* is the APNS error code returned by the server.

Usage

This method decodes the content of the `BYTE` variable passed as a parameter and received as response for a push notification message in the event of an error from the APNs server.

> **Note:** This `BYTE` variable must be located `IN MEMORY`.

The *uuid* is a binary value that identifies the push notification message. It is returned as a Base64-encoded string.

The *error* returned value defines the APNs error code. For example, error will be set to 10 if the APNs server was shutdown. See the Apple Push Notification Service error reference for more details.

In the case of a decoding error, the method will raise the exception -15566, with details in the `SQLCA.SQLERRM` register.

Example

```
DEFINE error_data BYTE,
       uuid STRING,
       error INTEGER

    LOCATE error_data IN MEMORY

    -- Send push notification message TCP request
    ...
    CALL req.doDataRequest(data)
    LET resp = req.getResponse()
    TRY
        CALL resp.getDataResponse(error_data)
        CALL com.APNS.DecodeError(error_data)
             RETURNING uuid, ecode
    ...
```

For a complete example, see APNs push provider on page 2101.

com.APNS.DecodeFeedback

Decodes content of BYTE data returned from the APNS feedback service.

Syntax

```
com.APNS.DecodeFeedback(
   data BYTE,
   unregs DYNAMIC ARRAY OF RECORD
          timestamp INTEGER,
          deviceToken STRING
       END RECORD
   )
```

1. *data* is the `BYTE` variable containing the feedback data. This `BYTE` variable must be located `IN MEMORY`.
2. *unregs* is a structured dynamic array that will contain the list of unregistered device tokens.
   a. *timestamp* is the number of seconds since Unix Epoch (in UTC)
   b. *deviceToken* is a APNS device token that has been unregistered (encoded in Base-64)

Usage

Apple recommends to connect frequently to the APNS feedback server in order to verify that your applications are still registered for push notifications.

To get APNS feedback, you must perform a TCP request (using SSL), to the following specific URI:

```
tcps://feedback.push.apple.com:2196
```

The `DecodeFeedback()` method decodes the content of the `BYTE` variable, which was passed as a parameter and received as response for the TCP request to the APNS feedback server.

Note: This BYTE variable must be located IN MEMORY.

For the second parameter, this method takes a structured dynamic array that will be filled with the list of unregistered APNS device tokens. It is up to the push program to stop sending push notification messages for these unregistered device tokens.

The *timestamp* member of an *unregs* dynamic array element can be used to verify that device tokens have not been re-registered since the feedback entry was generated. This timestamp is returned as a number of seconds since the Unix epoch, in UTC. Use the util.Datetime.fromSecondsSinceEpoch on page 1950 utility API to convert timestamp to a DATETIME value in the current local time.

The *deviceToken* member of an *unregs* dynamic array element identifies iOS devices that have been unregistered from the APNS server. Note that these identifier is encoded in Base64.

In the event of a decoding error, the method will raise the exception -15566, with details in the SQLCA.SQLERRM register.

Example

```
DEFINE feedback_data BYTE,
       unregs DYNAMIC ARRAY OF RECORD
              timestamp   INTEGER,
              deviceToken STRING
           END RECORD,
       i INTEGER

LOCATE feedback_data IN MEMORY

... TCP request to APNS feedback server ...

CALL com.APNS.DecodeFeedback(feedback_data, unregs)

FOR i=1 TO unregs.getLength()
    DISPLAY i, " ", unrefs[i].deviceToken
END FOR
```

For a complete example, see APNs feedback handler on page 2102.

com.APNS.EncodeMessage

Encodes an APNS specific push notification message into a BYTE.

Syntax

```
com.APNS.EncodeMessage(
   data BYTE,
   deviceToken STRING,
   json STRING,
   uuid STRING,
   expiration INTEGER,
   priority SMALLINT)
  RETURNING result INTEGER
```

1. *data* is the BYTE variable holding the APNS message. This BYTE variable must be located IN MEMORY.
2. *deviceToken* is an APNS device token (encoded in Base-64).
3. *json* is a JSON string containing the APNS push message data.
4. *uuid* is the 4 bytes-long push message identifier (encoded in Base64).
5. *expiration* is a number of seconds since Unix Epoch defining the expiration date of the message.
6. *priority* is an integer defining the priority of the message.

Usage

This method builds the APNS push notification message into a BYTE variable, for a given device token.

> **Note:** This BYTE variable must be located IN MEMORY.

> **Note:** The size of an APNS notification payload cannot exceed 2 Kilobytes. Make sure that the resulting BYTE variable does not exceed this size limitation. If more information needs to be passed, after receiving the push message, apps must contact the server part to query for more information. However, this is only possible when network is available.

The APNS push notification message protocol requires some binary data to be encoded in the message content before it is sent to the APNS server with a TCP (over SSL) request, to specific URIs, namely:

- `"tcps://gateway.sandbox.apple.com:2195"` (for development)
- `"tcps://gateway.push.apple.com:2195"` (for production)

You need to provide several parameters in order to build the push notification message:

The *deviceToken* parameter is an APNS device token encoded in Base-64. It's used to identify the target device that must receive the push message. The device token identifies a single iOS device: If you have N devices registered to your push notification provider, you will have N different device tokens. If you want to send one push notification message to all the devices, you must send N different messages, where the only difference between the messages is the device token.

> **Note:** It's in your hands to handle the list of registered device tokens. A device token is assigned to a physical iOS device when the mobile app issues a registerForRemoteNotifications on page 1934 front call. The app must then provide its device token to the push provider program using a method such as a web service mechanism.

Fill the *json* parameter with a JSON string containing the APNS push message data. For example:

```
LET json = '{"aps":{"alert":"Hello,
 world","sound":"default","badge":1,"content-available":1}}'
```

See APNS documentation for more details about the JSON content of a message.

The *uuid* parameter is the 4 bytes-long push message identifier, encoded in Base64. This parameter can used later to identify the message in push notification errors (com.APNS.DecodeError on page 2097). This parameter can be NULL. To create the *uuid* parameter, use the security.RandomGenerator.CreateRandomString on page 2279 API, with a size of 4:

```
LET uuid = security.RandomGenerator.createRandomString(4)
```

The *expiration* parameter is a number of seconds since Unix Epoch. It defines the expiration date of the message if it can not be sent by the APNS server to the target devices. This parameter can be NULL, to indicate that there is no expiration date:

```
LET dt = CURRENT + INTERVAL (10) MINUTE TO MINUTE
LET expiration = util.Datetime.toSecondsSinceEpoch(dt)
```

The *priority* parameter can be used to define a priority for the push notification message. Typically, use a value of 10 for immediate, 5 for delayed. This parameter can be NULL. See APNS documentation for more details.

If there's an encoding error, the method will raise the exception -15566, with details in the SQLCA.SQLERRM register.

Example

```
DEFINE push_data BYTE,
       deviceTokenHexa STRING,
```

```
        dt DATETIME YEAR TO FRACTION(3),
        expiration INTEGER,
        json_data STRING,
        uuid STRING

LOCATE push_data IN MEMORY

LET deviceTokenHexa = "84e3................."

LET dt = CURRENT + INTERVAL (10) MINUTE TO MINUTE
LET expiration = util.Datetime.toSecondsSinceEpoch(dt)

LET json_date = util.JSON.stringify(...)

LET uuid = security.RandomGenerator.createRandomString(4)

CALL com.APNS.EncodeMessage(
             push_data,
             security.HexBinary.ToBase64(deviceTokenHexa),
             json_data,
             uuid,
             expiration,
             10
       )

IF LENGTH(push_data) > 2000 THEN
   -- Must reduce the message content...
   RETURN FALSE
END IF

-- Do the TCP request with the push_data variable
...
```

For a complete example, see APNs push provider on page 2101.

APNs examples

APNs push provider

The `com.APNS` class can be used to implement an APNs push provider.

The following code example implements a push program using the `com.APNS` API to send a notification message to devices by using the TCP request API. See also com.APNS methods on page 2097 for more details about the APNs API.

> **Note:** An SSL certificate needs to be defined in FGLPROFILE, as described in APNs SSL certificate on page 2095.
>
> **Note:** The size of the resulting BYTE variable containing the APNS payload cannot exceed 2 Kilobytes.
>
> **Important:** In order to check that the push message was properly handled by the APNs server, you need to define a TCP request timeout (2 seconds in this example). In case of error, the APNs server will return a response immediately. In case of success, there is not a response from the APNs server. For more details about this protocol, see Apple's APNs documentation.

```
IMPORT com
IMPORT security
IMPORT util

MAIN
  DEFINE json STRING
  DEFINE deviceTokenHexa STRING
  DEFINE req com.TCPRequest
  DEFINE resp com.TCPResponse
```

```
  DEFINE uuid STRING
  DEFINE ecode INTEGER
  DEFINE dt DATETIME YEAR TO SECOND
  DEFINE exp INTEGER
  DEFINE data, err BYTE

  LOCATE data IN MEMORY
  LOCATE err IN MEMORY

  LET deviceTokenHexa = "84e3................."
  LET dt = CURRENT + INTERVAL(10) MINUTE TO MINUTE
  LET exp = util.Datetime.toSecondsSinceEpoch(dt)
  TRY
    LET req = com.TCPRequest.create( "tcps://gateway.push.apple.com:2195" )
    CALL req.setKeepConnection(true)
    CALL req.setTimeout(2) # Wait 2 seconds for APNs to return an error code
    LET uuid = security.RandomGenerator.createRandomString(4)
    LET json = '{"aps":{"alert":"Hello,
 world","sound":"default","badge":1,"content-available":1}}'
    CALL com.APNS.EncodeMessage(
                  data,
                  security.HexBinary.ToBase64(deviceTokenHexa),
                  json,
                  uuid,
                  exp,
                  10
              )
    IF LENGTH(data) > 2000 THEN
       DISPLAY "APNS payload cannot exceed 2 kilobytes"
       EXIT PROGRAM 1
    END IF
    DISPLAY "Sending notif with ID:",uuid," and expiring at ",dt
    CALL req.doDataRequest(data)
    LET resp = req.getResponse()
    TRY
        CALL resp.getDataResponse(err)
        CALL com.APNS.DecodeError(err) RETURNING uuid, ecode
        DISPLAY "ERROR code :",ecode
        DISPLAY "ERROR uuid :",uuid
    CATCH
        CASE STATUS
          WHEN -15553 DISPLAY "Timeout Push sent without error"
          WHEN -15566 DISPLAY "Operation failed :", SQLCA.SQLERRM
          WHEN -15564 DISPLAY "Server has shutdown"
          OTHERWISE   DISPLAY "ERROR :",STATUS
        END CASE
    END TRY
  CATCH
    DISPLAY "ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
  END TRY

END MAIN
```

APNs feedback handler

The `com.APNS` class can be used to implement a server application to query the APNs feedback service.

Implement an APNs feedback handler to get a list of unregistered device tokens in order to stop sending push notification messages to these apps.

> **Note:** An SSL certificate needs to be defined in FGLPROFILE, as described in APNs SSL certificate on page 2095.

```
IMPORT com
```

```
IMPORT security
IMPORT util

MAIN
  DEFINE req com.TCPRequest
  DEFINE resp com.TCPResponse
  DEFINE feedback DYNAMIC ARRAY OF RECORD
                      timestamp INTEGER,
                      deviceToken  STRING
                  END RECORD
  DEFINE timestamp DATETIME YEAR TO SECOND
  DEFINE i INTEGER
  DEFINE data BYTE

  LOCATE data IN MEMORY

  TRY
    LET req = com.TCPRequest.create( "tcps://feedback.push.apple.com:2196" )
    CALL req.setKeepConnection(true)
    CALL req.setTimeout(2)
    CALL req.doRequest()
    LET resp = req.getResponse()
    CALL resp.getDataResponse(data)
    DISPLAY "Feedback service has responded"
    CALL com.APNS.DecodeFeedback(data,feedback)
    FOR i=1 TO feedback.getLength()
      LET timestamp =
 util.Datetime.fromSecondsSinceEpoch(feedback[i].timestamp)
      DISPLAY "Device Token :",feedback[i].deviceToken, " Timestamp :",
 timestamp
    END FOR
  CATCH
    CASE STATUS
      WHEN -15553 DISPLAY "Timeout: No feedback message"
      WHEN -15566 DISPLAY "Operation failed :", SQLCA.SQLERRM
      WHEN -15564 DISPLAY "Server has shutdown"
      OTHERWISE   DISPLAY "ERROR :",STATUS
    END CASE
  END TRY

END MAIN
```

The xml package

The Genero Web Services XML package provides classes and methods to handle any kind of XML documents, including documents with namespaces.

The library provides a W3C-compatible DOM API, integrating additional XML Schema and DTD validation methods. There is also an API compatible with StAX for writing or reading XML documents where performance and speed are important.

Use the `IMPORT` statement at the top of the module using this library:

```
IMPORT xml
```

> **Note:** The DOM API of the `om` package is designed to handle specific FGL files or to manipulate the user interface tree (the AUI tree). For all other cases/scenarios, we recommend that you use the DOM API of the Web Services `xml` package.

- The Document Object Modeling (DOM) classes on page 2104
- The streaming API for XML (StAX) classes on page 2170
- XML serialization classes on page 2202

- XML security classes on page 2208
- OM to XML Migration on page 2276

The Document Object Modeling (DOM) classes
The Document Object Modeling (DOM) classes manage XML documents entirely in memory with support of XML Schema and DTD validation.

- CLASS DomDocument
  - Features
- CLASS DomNode
  - Types
- CLASS DomNodeList

The DomDocument class
The `xml.DomDocument` class provides methods to manipulate a data tree, following the DOM standards.

The `STATUS` variable is set to zero after a successful method call.

xml.DomDocument methods
Methods for the `xml.DomDocument` class.

Table 476: Class methods: Creation

Name	Description
`xml.DomDocument.create()` `  RETURNING object xml.DomDocument`	Constructor of an empty DomDocument object.

`xml.DomDocument.createDocument(`
`  name STRING )`
`  RETURNING object xml.DomDocument`
	Constructor of a DomDocument with an XML root element.

`xml.DomDocument.createDocumentNS(`
`  prefix STRING,`
`  name STRING,`
`  ns STRING )`
`  RETURNING object xml.DomDocument`
	Constructor of a DomDocument with a root namespace-qualified XML root element

Table 477: Object methods: Navigation

Name	Description
`getDocumentElement()` `  RETURNING object xml.DomNode`	Returns the root XML Element DomNode object for this DomDocument object.
`getDocumentNodesCount()` `  RETURNING count INTEGER`	Returns the number of child DomNode objects for a DomDocument object.
`getDocumentNodeItem(` `  pos INTEGER )`	Returns the child DomNode object at a given position for this DomDocument object.

Name	Description
`RETURNING object xml.DomNode`	
`getElementById( id STRING ) RETURNING object xml.DomNode`	Returns the element that has an attribute of type ID with the given value
`getElementsByTagName( name STRING ) RETURNING object xml.DomNodeList`	Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document.
`getElementsByTagNameNS( name STRING, ns STRING ) RETURNING list xml.DomNodeList`	Returns a DomNodeList object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document
`getFirstDocumentNode() RETURNING object xml.DomNode`	Returns the first child DomNode object for a DomDocument object.
`getLastDocumentNode() RETURNING object xml.DomNode`	Returns the last child DomNode object for a DomDocument object.
`selectByXPath( expr STRING, nslist ... ) RETURNING list xml.DomNodeList`	Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression.

Table 478: Object methods: Management

Name	Description
`appendDocumentNode( node xml.DomNode )`	Adds a child DomNode object to the end of the DomNode children for this DomDocument object.
`clone() RETURNING object xml.DomDocument`	Returns a copy of a DomDocument object.
`declareNamespace( node xml.DomNode, alias STRING, ns STRING )`	Forces namespace declaration to an XML Element DomNode for a DomDocument object.
`insertAfterDocumentNode( node xml.DomNode,`	Inserts a child DomNode object after another child DomNode for a DomDocument object.

Name	Description
ref xml.DomNode)	
insertBeforeDocumentNode(*node* xml.DomNode, *ref* xml.DomNode)	Inserts a child DomNode object before another child DomNode for this DomDocument object.
importNode(*node* xml.DomNode *deep* INTEGER) RETURNING *object* xml.DomNode	Imports a DomNode from a DomDocument object into its new context (attached to a DomDocument object).
prependDocumentNode(*node* xml.DomNode)	Adds a child DomNode object to the beginning of the DomNode children for a DomDocument object
removeDocumentNode(*node* xml.DomNode)	Removes a child DomNode object from the DomNode children for this DomDocument object.

Table 479: Object methods: Node Creation

Name	Description
createAttribute(*name* STRING) RETURNING *object* xml.DomNode	Creates an XML Attribute DomNode object for a DomDocument object.
createAttributeNS(*prefix* STRING, *name* STRING, *ns* STRING) RETURNING *object* xml.DomNode	Creates an XML namespace-qualified Attribute DomNode object for a DomDocument object.
createCDATASection(*cdata* STRING) RETURNING *object* xml.DomNode	Creates an XML CData DomNode object for a DomDocument object.
createComment(*comment* STRING) RETURNING *object* xml.DomNode	Creates an XML Comment DomNode object for a DomDocument object.
createDocumentFragment() RETURNING *object* xml.DomNode	Creates an XML Document Fragment DomNode object for a DomDocument object.
createDocumentType(*name* STRING, *publicID* STRING, *systemID* STRING, *internalDTD* STRING)	Creates an XML Document Type (DTD) DomNode object for a DomDocument object.

Name	Description
RETURNING *object* xml.DomNode	
createElement(*name* STRING) RETURNING *object* xml.DomNode	Creates an XML Element DomNode object for a DomDocument object
createElementNS(*prefix* STRING, *name* STRING, *ns* STRING) RETURNING *object* xml.DomNode	Creates an XML namespace-qualified Element DomNode object for a DomDocument object.
createEntityReference(*ref* STRING) RETURNING *object* xml.DomNode	Creates an XML EntityReference DomNode object for a DomDocument object
createNode(*str* STRING) RETURNING *object* xml.DomNode	Creates an XML DomNode object from a string for a DomDocument object.
createProcessingInstruction(*target* STRING, *data* STRING) RETURNING *object* xml.DomNode	Creates an XML Processing Instruction DomNode object for this DomDocument object.
createTextNode(*text* STRING) RETURNING *object* xml.DomNode	Creates an XML Text DomNode object for a DomDocument object.

Table 480: Object methods: Load and Save

Name	Description
load(*url* STRING)	Loads an XML Document into a DomDocument object from a file or an URL.
loadFromPipe(*cmd* STRING)	Loads an XML Document into a DomDocument object from a PIPE.
loadFromString(*str* STRING)	Loads an XML Document into a DomDocument object from a string.
normalize()	Normalizes the entire Document.
save(	Saves a DomDocument object as an XML Document to a file or URL.

Name	Description
url STRING)	
saveToPipe(*cmd* STRING)	Saves a DomDocument object as an XML Document to a PIPE.
saveToString() RETURNING *result* STRING	Saves a DomDocument object as an XML Document to a string.

Table 481: Object methods: Configuration

Name	Description
getFeature(*feature* STRING) RETURNING *result* STRING	Gets a feature for a DomDocument object.
getXmlEncoding() RETURNING *result* STRING	Returns the document encoding as defined in the XML document declaration.
getXmlVersion() RETURNING *result* STRING	Returns the document version as defined in the XML document declaration.
isXmlStandalone() RETURNING *result* INTEGER	Returns whether the XML standalone attribute is set in the XML declaration.
setFeature(*feature* STRING, *value* STRING)	Sets a feature for a DomDocument object.
setXmlEncoding(*enc* STRING)	Sets the XML document encoding in the XML declaration.
setXmlStandalone(*alone* INTEGER)	Sets the XML standalone attribute in the XML declaration to yes or no in the XML declaration.

Table 482: Object methods: Validation

Name	Description
validate() RETURNING *result* INTEGER	Performs a DTD or XML Schema validation for a DomDocument object.
validateOneElement(*node* xml.DomNode)	Performs a DTD or XML Schema validation of an XML Element DomNode object.

Name	Description
`RETURNING result INTEGER`	

Table 483: Object methods: Error Management

Name	Description
`getErrorsCount()` `RETURNING count INTEGER`	Returns the number of errors encountered during the loading, saving or validation of an XML document.

`getErrorDescription(`
`pos INTEGER )`
`RETURNING desc STRING`
	Returns the error description at given position.

xml.DomDocument.appendDocumentNode

Adds a child DomNode object to the end of the DomNode children for this DomDocument object.

Syntax

```
appendDocumentNode(
   node xml.DomNode )
```

1. *node* is the node to add.

Usage

Adds a child DomNode object to the end of the DomNode children for this DomDocument object, where *node* is the node to add.

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed.

> **Note:** A fragment is a structure created to receive xml nodes that are not always valid. Once a fragment is added to a valid node, the fragment becomes empty as all nodes are moved from the fragment as a child to the valid node. So developpers can work on the fragment until it is added to another node. At that time developers should no more work on the fragment but rather on the valid node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.clone

Returns a copy of a DomDocument object.

Syntax

```
clone()
  RETURNING object xml.DomDocument
```

Usage

Returns a copy of this DomDocument object, or `NULL`.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.create

Constructor of an empty DomDocument object.

Syntax

```
xml.DomDocument.create()
  RETURNING object xml.DomDocument
```

Usage

Constructor of an empty DomDocument object.

Returns a DomDocument object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Example

Create a DomDocument without a root node:

```
xml.domDocument.create()
```

Create a DomDocument with an initial root node named **ARoot**:

```
xml.domDocument.create("ARoot")
```

xml.DomDocument.createAttribute

Creates an XML Attribute DomNode object for a DomDocument object.

Syntax

```
createAttribute(
   name STRING )
  RETURNING object xml.DomNode
```

1. *name* is the name of the XML attribute.

Usage

Creates an XML Attribute DomNode object for a DomDocument object, where *name* is the name of the XML attribute, cannot be NULL.

Returns a DomNode object, or NULL.

To create a default namespace declaration attribute use **xmlns** as the name. (Using declareNamespace instead is recommended)

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createAttributeNS

Creates an XML namespace-qualified Attribute DomNode object for a DomDocument object.

Syntax

```
createAttributeNS(
   prefix STRING,
   name STRING,
   ns STRING )
  RETURNING object xml.DomNode
```

1. *prefix* is the prefix of the XML attribute.
2. *name* is the name of the XML attribute.
3. *ns* is the namespace URI of the XML attribute.

Usage

Creates an XML namespace-qualified Attribute DomNode object for this DomDocument object, where *prefix* is the prefix of the XML attribute, cannot be `NULL`; *name* is the name of the XML attribute, cannot be `NULL`; *ns* is the namespace URI of the XML attribute, cannot be `NULL`.

Returns a DomNode object, or `NULL`.

To create a namespace declaration attribute use **xmlns** as the prefix and **http://www.w3.org/XML/1998/namespace** as the namespace. Using `declareNamespace` instead is recommended.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createCDATASection

Creates an XML CData DomNode object for a DomDocument object.

Syntax

```
createCDATASection(
   cdata STRING )
  RETURNING object xml.DomNode
```

1. *cdata* is the data of the XML CData node.

Usage

Creates an XML CData DomNode object for this DomDocument object, where *cdata* is the data of the XML CData node, or `NULL`. Returns a DomNode object, or `NULL`.

Only the characters **#x9**, **#xA**, **#xD**, [**#x20-#xD7FF**], [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML CDATASection node.

The character sequence (Double-Hyphen) '--' is not allowed in the content of an XML CDATASection node. The `saveToFile()` and `normalize()` methods will fail if this sequence or characters other than those allowed exist in a CDATASection node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createComment

Creates an XML Comment DomNode object for a DomDocument object.

Syntax

```
createComment(
   comment STRING )
  RETURNING object xml.DomNode
```

1. *comment* is the data of the XML Comment node.

Usage

Creates an XML Comment DomNode object for this DomDocument object, where *comment* is the data of the XML Comment node, or `NULL`.

Returns a DomNode object, or `NULL`.

Only the characters **#x9**, **#xA**, **#xD**, [**#x20-#xD7FF**], [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML Comment node.

The character sequence (Double-Hyphen) '--' is not allowed in the content of an XML Comment node. The `saveToFile()` and `normalize()` methods will fail if this sequence or characters other than those allowed exist in a Comment node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createDocument

Constructor of a DomDocument with an XML root element.

Syntax

```
xml.DomDocument.createDocument(
   name STRING )
  RETURNING object xml.DomDocument
```

1. *name* is the name of the XML Element.

Usage

Constructor of a xml.DomDocument with an XML root element; where *name* is the name of the XML Element.

Returns a DomDocument object or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createDocumentFragment

Creates an XML Document Fragment DomNode object for a DomDocument object.

Syntax

```
createDocumentFragment()
  RETURNING object xml.DomNode
```

Usage

Creates an XML Document Fragment DomNode object for this DomDocument object.

Returns a DomNode object, or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createDocumentNS

Constructor of a DomDocument with a root namespace-qualified XML root element

Syntax

```
xml.DomDocument.createDocumentNS(
   prefix STRING,
   name STRING,
   ns STRING )
  RETURNING object xml.DomDocument
```

1. *prefix* is the prefix of the XML Element or NULL.
2. *name* is the name of the XML Element.
3. *ns* is the namespace of the XML Element.

Usage

Constructor of a xml.DomDocument with a root namespace-qualified XML root element where *prefix* is the prefix of the XML Element or `NULL`, *name* is the name of the XML Element, and *ns* is the namespace of the XML Element. Returns a DomDocument object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

Example

Create a DomDocument with an initial root node named `"List"` with `abc` as the prefix and `http://www.mysite.com/xmlapi` as the namespace:

```
xml.domdocument.createDocumentNS("abc","List","http://
www.mysite.com/xmlapi")
```

Produces:

```
<abc:List xmlns:abc="http://www.mysite.com/xmlapi">
[...]
</abc:List>
```

xml.DomDocument.createDocumentType

Creates an XML Document Type (DTD) DomNode object for a DomDocument object.

Syntax

```
createDocumentType(
   name STRING,
   publicID STRING,
   systemID STRING,
   internalDTD STRING )
```

```
   RETURNING object xml.DomNode
```

1. *name* is the name of the document type.
2. *publicID* is the URI of the public identifier.
3. *systemID* is the URL of the system identifier (Specifies the file location of the external DTD subset).
4. *internalDTD* is the internal DTD subset.

Usage

Creates an XML Document Type (DTD) DomNode object for this DomDocument object; *name* is the name of the document type; *publicID* is the URI of the public identifier or `NULL`; *systemID* is the URL of the system identifier or `NULL` (Specifies the file location of the external DTD subset); *internalDTD* is the internal DTD subset or `NULL`.

Returns a DomNode object, or `NULL` if *internalDTD* is malformed.

Caution: Not part of W3C API.

Only internal DTDs are supported.

The public identifier cannot be set without the system identifier.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createElement

Creates an XML Element DomNode object for a DomDocument object

Syntax

```
createElement(
   name STRING )
  RETURNING object xml.DomNode
```

1. *name* is the name of the XML element.

Usage

Creates an XML Element DomNode object for this DomDocument object, where *name* is the name of the XML element, cannot be `NULL`.

Returns a DomNode object, or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createElementNS

Creates an XML namespace-qualified Element DomNode object for a DomDocument object.

Syntax

```
createElementNS(
   prefix STRING,
   name STRING,
   ns STRING )
  RETURNING object xml.DomNode
```

1. *prefix* is the prefix of the XML element, or `NULL` to use the default namespace.

2. *name* is the name of the XML element.
3. *ns* is the namespace URI of the XML element.

Usage

Creates an XML namespace-qualified Element DomNode object for this DomDocument object, where *prefix* is the prefix of the XML element, or `NULL` to use the default namespace; *name* is the name of the XML element, cannot be `NULL`; *ns* is the namespace URI of the XML element, cannot be `NULL`.

Returns a DomNode object, or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createEntityReference

Creates an XML EntityReference DomNode object for a DomDocument object

Syntax

```
createEntityReference(
   ref STRING )
  RETURNING object xml.DomNode
```

1. *ref* is the name of the entity reference.

Usage

Creates an XML EntityReference DomNode object for this DomDocument object, where *ref* is the name of the entity reference.

Returns a DomNode object, or `NULL`.

An Entity Reference node is read-only and cannot be modified.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createNode

Creates an XML DomNode object from a string for a DomDocument object.

Syntax

```
createNode(
   str STRING )
  RETURNING object xml.DomNode
```

1. *str* is the string representation of the DomNode to be created.

Usage

Creates an XML DomNode object from a string for this DomDocument object; *str* is the string representation of the DomNode to be created.

Returns a xml.DomNode object, or `NULL`.

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createProcessingInstruction

Creates an XML Processing Instruction DomNode object for this DomDocument object.

Syntax

```
createProcessingInstruction(
   target STRING,
   data STRING )
  RETURNING object xml.DomNode
```

1. *target* is the target part of the XML Processing Instruction.
2. *data* is the data part of the XML Processing Instruction.

Usage

Creates an XML Processing Instruction DomNode object for this DomDocument object, where *target* is the target part of the XML Processing Instruction, cannot be NULL; *data* is the data part of the XML Processing Instruction, or NULL.

Returns a DomNode object, or NULL.

Only the characters **#x9**, **#xA**, **#xD**, [**#x20-#xD7FF**], [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML Processing Instruction node.

The character sequence (Double-Hyphen) '--' is not allowed in the content of an XML Processing Instruction. The save() and normalize() methods will fail if this sequence or characters other than those allowed exist in a Processing Instruction node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.createTextNode

Creates an XML Text DomNode object for a DomDocument object.

Syntax

```
createTextNode(
   text STRING )
  RETURNING object xml.DomNode
```

1. *text* is the data of the XML Text node.

Usage

Creates an XML Text DomNode object for this DomDocument object, where *text* is the data of the XML Text node, or NULL.

Returns a DomNode object, or NULL.

Only the characters **#x9**, **#xA**, **#xD**, [**#x20-#xD7FF**], [**#xE000-#xFFFD**] and [**#x10000-#x10FFFF**] are allowed in the content of an XML Text node. The save() and normalize() methods will fail if characters other than those allowed exist in a Text node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.declareNamespace

Forces namespace declaration to an XML Element DomNode for a DomDocument object.

Syntax

```
declareNamespace(
   node xml.DomNode,
   alias STRING,
   ns STRING )
```

1. *node* is the XML Element DomNode that carries the namespace definition.
2. *alias* is the alias of the namespace to declare.
3. *ns* is the URI of the namespace to declare.

Usage

Forces namespace declaration to an XML Element DomNode for this DomDocument object ; *node* is the XML Element DomNode that carries the namespace definition; *alias* is the alias of the namespace to declare, or `NULL` to declare the default namespace; *ns* is the URI of the namespace to declare (can only be `NULL` if alias is `NULL`).

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getDocumentElement

Returns the root XML Element DomNode object for this DomDocument object.

Syntax

```
getDocumentElement()
  RETURNING object xml.DomNode
```

Usage

Returns the root XML Element DomNode object for this DomDocument object.

Returns a DomNode object, or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getDocumentNodesCount

Returns the number of child DomNode objects for a DomDocument object.

Syntax

```
getDocumentNodesCount()
  RETURNING count INTEGER
```

Usage

Returns the number of child DomNode objects for this DomDocument object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getDocumentNodeItem

Returns the child DomNode object at a given position for this DomDocument object.

Syntax

```
getDocumentNodeItem(
   pos INTEGER )
  RETURNING object xml.DomNode
```

1. *pos* is the position of the node to return (index starts at 1).

Usage

Returns the child DomNode object at a given position for this DomDocument object where *pos* is the position of the node to return (Index starts at 1), or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getElementById

Returns the element that has an attribute of type ID with the given value

Syntax

```
getElementById(
   id STRING )
  RETURNING object xml.DomNode
```

1. *id* is the Id value.

Usage

Returns the xml.DomNode element that has an attribute of type ID with the given value, or NULL if there is none.

Attributes with the name "ID" or "id" are not of type ID unless so defined with setIdAttribute or setIdAttributeNS. However, there is a specific attribute called **xml:id** and belonging to the namespace **http://www.w3.org/XML/1998/namespace** that is always of type ID even if not set with setIdAttributeNS.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getElementsByTagName

Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document.

Syntax

```
getElementsByTagName(
   name STRING )
  RETURNING object xml.DomNodeList
```

1. *name* is the name of the XML Element tag to match or "*" to match all tags.

Usage

Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name in the entire document; *name* is the name of the XML Element tag to match, or "*" to match all tags.

Returns a DomNodeList object, or `NULL`.

The returned list is ordered using a Depth-First pass algorithm.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getElementsByTagNameNS

Returns a DomNodeList object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document

Syntax

```
getElementsByTagNameNS(
   name STRING,
   ns STRING )
  RETURNING list xml.DomNodeList
```

1. *name* is the name of the XML Element tag to match or "*" to match all tags.
2. *ns* is the namespace URI of the XML Element tag to match, or "*" to match all namespaces.

Usage

Returns a xml.DomNodeList object containing all namespace qualified XML Element DomNode objects with the same tag name and namespace in the entire document; *name* is the name of the XML Element tag to match, or "*" to match all tags; *ns* is the namespace URI of the XML Element tag to match, or "*" to match all namespaces. Returns a DomNodeList object, or `NULL`.

The returned list is ordered using a Depth-First pass algorithm.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getErrorDescription

Returns the error description at given position.

Syntax

```
getErrorDescription(
   pos INTEGER )
  RETURNING desc STRING
```

1. *pos* is the position of the error description (index starts at 1).

Usage

Returns the error description at given position. *pos* is the position of the error description (index starts at 1). Returns a string with an error description.

Caution: Not part of W3C API

Example

```
FOR i=1 TO doc.getErrorsCount()
```

```
  DISPLAY "[", i, "] ", doc.getErrorDescription(i)
END FOR
```

Displays all the errors encountered in the save, load or validate of doc DomDocument.

To display other errors, use the global variable STATUS to get the error code and err_get(status) or sqlca.sqlerrm to get the description of the error. See error code for more details.

xml.DomDocument.getErrorsCount

Returns the number of errors encountered during the loading, saving or validation of an XML document.

Syntax

```
getErrorsCount()
  RETURNING count INTEGER
```

Usage

Returns the number of errors encountered during the loading, the saving or the validation of an XML document.

Returns the number of errors, or zero if there are none.

Caution: Not part of W3C API

Example

```
FOR i=1 TO doc.getErrorsCount()
  DISPLAY "[", i, "] ", doc.getErrorDescription(i)
END FOR
```

Displays all the errors encountered in the save, load or validate of doc DomDocument.

To display other errors, use the global variable STATUS to get the error code and err_get(status) or sqlca.sqlerrm to get the description of the error. See error code for more details.

xml.DomDocument.getFirstDocumentNode

Returns the first child DomNode object for a DomDocument object.

Syntax

```
getFirstDocumentNode()
  RETURNING object xml.DomNode
```

Usage

Returns the first child DomNode object for this DomDocument object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getFeature

Gets a feature for a DomDocument object.

Syntax

```
getFeature(
   feature STRING)
```

```
  RETURNING result STRING
```

1. *feature* is the name of the DomDocument feature.

Usage

Gets a feature for the DomDocument object, where *feature* is the name of the DomDocument feature.

Returns the value of the feature.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getLastDocumentNode

Returns the last child DomNode object for a DomDocument object.

Syntax

```
getLastDocumentNode()
  RETURNING object xml.DomNode
```

Usage

Returns the last child DomNode object for this DomDocument object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getXmlEncoding

Returns the document encoding as defined in the XML document declaration.

Syntax

```
getXmlEncoding()
  RETURNING result STRING
```

Usage

Returns the document encoding as defined in the XML document declaration, or NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.getXmlVersion

Returns the document version as defined in the XML document declaration.

Syntax

```
getXmlVersion()
  RETURNING result STRING
```

Usage

Returns the document version as defined in the XML document declaration, which is **1.0**. No other versions are supported.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.importNode

Imports a DomNode from a DomDocument object into its new context (attached to a DomDocument object).

Syntax

```
importNode(
   node xml.DomNode
   deep INTEGER )
  RETURNING object xml.DomNode
```

1. *node* is the node to import.
2. *deep* is a boolean identifying whether to import the node only or the node and all its child nodes.

Usage

Imports a DomNode from a DomDocument object into its new context (attached to this DomDocument object), where *node* is the node to import. When *deep* is FALSE only the node is imported; when TRUE the node and all its child nodes are imported.

Returns the DomNode object that has been imported to this DomDocument, or NULL.

Document and Document Type nodes cannot be imported.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.insertBeforeDocumentNode

Inserts a child DomNode object before another child DomNode for this DomDocument object.

Syntax

```
insertBeforeDocumentNode(
   node xml.DomNode,
   ref xml.DomNode )
```

1. *node* is the node to insert.
2. *ref* is the reference node (the node before which the new node must be inserted).

Usage

Inserts a child DomNode object before another child DomNode for this DomDocument object; *node* is the node to insert, *ref* is the reference node (the node before which the new node must be inserted).

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.insertAfterDocumentNode

Inserts a child DomNode object after another child DomNode for a DomDocument object.

Syntax

```
insertAfterDocumentNode(
   node xml.DomNode,
   ref xml.DomNode )
```

1. *node* is the node to insert.
2. *ref* is the reference node (the node after which the new node must be inserted).

Usage

Inserts a child DomNode object after another child DomNode for this DomDocument object ; *node* is the node to insert; *ref* is the reference node (the node after which the new node must be inserted).

Caution: Not part of W3C API.

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.isXmlStandalone

Returns whether the XML standalone attribute is set in the XML declaration.

Syntax

```
isXmlStandalone()
  RETURNING result INTEGER
```

Usage

Returns whether the XML standalone attribute is set in the XML declaration.

Returns TRUE if the standalone attribute in the XML declaration is set to yes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.load

Loads an XML Document into a DomDocument object from a file or an URL.

Syntax

```
load(
   url STRING )
```

1. *url* is a valid URL or the name of the file.

Usage

Loads an XML Document into a DomDocument object from a file or an URL, where *url* is a valid URL or the name of the file.

Only the following kinds of URLs are supported: **http://** , **https://** , **tcp://** , **tcps://** , **file:///** and **alias://** . See Web services configuration on page 2509 for more details about URL mapping with aliases, and for proxy and security configuration.

See `setFeature()` to specify how the document can be loaded. **HTML** parsing is possible when enable-html-compliancy is enabled.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document loading.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

> **Important:** On Mac OS X versions prior to 10.9, the libxml library (used by Genero Web Services classes) has a bug when parsing HTML documents. If you set the HTML compliancy option with `CALL doc.setFeature("enable-html-compliancy",TRUE)`, loading an HTML document with `xml.DomDocument.load()` may produce additional blank TEXT nodes, because the libxml library loads some ignorable whitespace nodes from the HTML document. Starting with Mac OS X 10.9, the libxml library of the system has fixed this bug.

xml.DomDocument.loadFromPipe

Loads an XML Document into a DomDocument object from a PIPE.

Syntax

```
loadFromPipe(
   cmd STRING )
```

1. *cmd* is the command to read from the PIPE.

Usage

Loads an XML Document into a DomDocument object from a PIPE where *cmd* is the command to read from the PIPE.

> **Caution:** Not part of W3C API.

See `setFeature()` to specify how the document can be loaded. **HTML** parsing is possible when enable-html-compliancy is enabled.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document loading.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.loadFromString

Loads an XML Document into a DomDocument object from a string.

Syntax

```
loadFromString(
   str STRING )
```

1. *str* is the string to load.

Usage

Loads an XML Document into a DomDocument object from a string, where *str* is the string to load.

Caution: Not part of W3C API.

See `setFeature()` to specify how the document can be loaded. **HTML** parsing is possible when enable-html-compliancy is enabled.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document loading.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.normalize

Normalizes the entire Document.

Syntax

```
normalize()
```

Usage

Normalizes the entire Document. This method merges adjacent Text nodes, removes empty Text nodes and sets namespace declarations as if the document had been saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document normalization.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.prependDocumentNode

Adds a child DomNode object to the beginning of the DomNode children for a DomDocument object

Syntax

```
prependDocumentNode(
   node xml.DomNode )
```

1. *node* is the node to add.

Usage

Adds a child DomNode object to the beginning of the DomNode children for this DomDocument object; *node* is the node to add.

Caution: Not part of W3C API.

Only Text nodes, Processing Instruction nodes, Document Fragment nodes, one Element node and one Document Type node allowed.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.removeDocumentNode

Removes a child DomNode object from the DomNode children for this DomDocument object.

Syntax

```
removeDocumentNode(
   node xml.DomNode )
```

1. *node* is the node to remove.

Usage

Removes a child DomNode object from the DomNode children for this DomDocument object, where *node* is the node to remove.

Only Text nodes, Processing Instruction nodes, Element nodes and Document Type nodes allowed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.save

Saves a DomDocument object as an XML Document to a file or URL.

Syntax

```
save(
   url STRING )
```

1. *url* is a valid URL or the name of a file.

Usage

Saves a DomDocument object as an XML Document to a file or URL, where *url* is a valid URL or the name of the file.

Only the following kinds of URLs are supported: **http://** , **https://** , **tcp://** , **tcps://** , **file:///** and **alias://** . See Web services configuration on page 2509 for more details about URL mapping with aliases, and for proxy and security configuration.

See `setFeature()` to specify how the document can be saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document saving.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.saveToPipe

Saves a DomDocument object as an XML Document to a PIPE.

Syntax

```
saveToPipe(
   cmd STRING )
```

1. *cmd* is the command to start the pipe.

Usage

Saves a DomDocument object as an XML Document to a PIPE, where *cmd* is the command to start the pipe.

See `setFeature()` to specify how the document can be saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document saving.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.saveToString

Saves a DomDocument object as an XML Document to a string.

Syntax

```
saveToString()
  RETURNING result STRING
```

Usage

Saves a DomDocument object as an XML Document to a string. Returns the string that will contain the resulting document.

Caution: Not part of W3C API.

See `setFeature()` to specify how the document can be saved.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to XML document saving.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.selectByXPath

Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression.

Syntax

```
selectByXPath(
   expr STRING,
   nslist ... )
  RETURNING list xml.DomNodeList
```

1. *expr* is the XPath1.0 expression
2. *nslist* is a list of prefixes bounded to namespaces in order to resolve qualified names in the XPath expression.

Usage

Returns a xml.DomNodeList object containing all DomNode objects matching an XPath 1.0 expression.*expr* is the XPath1.0 expression, *nslist* is a list of prefixes bounded to namespaces in order to resolve qualified names in the XPath expression. This list must be filled with an even number of arguments, representing the prefix and its corresponding namespace.

Caution: Not part of W3C API.

Valid example:

```
selectByXPath(
  "//d:Record",
  "d",
  "http://defaultnamespace")
selectByXPath(
  "//ns1:Record",
  NULL)
selectByXPath(
  "//ns1:Records/ns2:Record",
  "ns1",
  "http://namespace1",
  "ns2",
  "http://namespace2")
```

Invalid example:

```
selectByXPath(
  "//ns1:Record",
  "ns1")
```

This example is invalid because the namespace definition is missing.

If the namespaces list is NULL, the prefixes and namespaces defined in the document itself are used if available.

A namespace must be an absolute URI (ex 'http://', 'file://').

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.setFeature

Sets a feature for a DomDocument object.

Syntax

```
setFeature(
   feature STRING,
   value STRING)
```

1. *feature* is the name of a DomDocument feature.
2. *value* is the value of a feature.

Usage

Sets a feature for the DomDocument object, where *feature* is the name of a DomDocument feature, and *value* is the value of a feature.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.setXmlEncoding

Sets the XML document encoding in the XML declaration.

Syntax

```
setXmlEncoding(
   enc STRING )
```

1. *enc* is the XML document encoding.

Usage

Sets the XML document encoding in the XML declaration, or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.setXmlStandalone

Sets the XML standalone attribute in the XML declaration to yes or no in the XML declaration.

Syntax

```
setXmlStandalone(
   alone INTEGER )
```

1. *alone* is a boolean flag.

Usage

Sets the XML standalone attribute in the XML declaration to yes or no in the XML declaration, or `NULL`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.validate

Performs a DTD or XML Schema validation for a DomDocument object.

Syntax

```
validate()
  RETURNING result INTEGER
```

Usage

Performs a DTD or XML Schema validation for this DomDocument object. Returns the number of validation errors, or zero if there are none.

Caution: Not part of W3C API.

See `setFeature()` to specify what kind of validation to do.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to validation errors.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomDocument.validateOneElement

Performs a DTD or XML Schema validation of an XML Element DomNode object.

Syntax

```
validateOneElement(
   node xml.DomNode )
  RETURNING result INTEGER
```

1. *node* is the XML Element DomNode to validate.

Usage

Performs a DTD or XML Schema validation of an XML Element DomNode object; *node* is the XML Element DomNode to validate.

Returns the number of validation errors, or zero if there are none.

> **Caution:** Not part of W3C API.

See `setFeature()` to specify what kind of validation to do.

See `getErrorsCount()` and `getErrorDescription()` to retrieve error messages related to validation errors.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

Navigation methods usage examples
Examples using the navigation methods of the `xml.DomDocument` class.

DomDocument navigation functions deal with nodes immediately under the DomDocument object, except for search features. To navigate through all the nodes, you can refer to the navigation functions of the class xml.DomNode.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="card.xsl"?>
<!-- demo card -->
<CardList xml:id="1" >[...]
</CardList>
```

The first node of the document is `xml-stylesheet`. Use **getFirstDocumentNode** to get the node. The element at position 2 is the comment `<!-- demo card -->`. Use **getDocumentNodeItem** function to get the node.

The last node of the document is `CardList`. Use **getLastDocumentNode** to get the node.

The number of node of the document is 3. This is result of function **getDocumentNodeCount**. This function only count the number of children immediately under the DomDocument.

Note that the first line of the example, `<?xml version="1.0" encoding="ISO-8859-1"?>`, is not considered as a node. To access to the information of the first line, use `getXmlVersion()` and `getXmlEncoding` functions.

Caution, if the example is in pretty printed format, the results are not the same. There are addition text nodes representing the carriage returns.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="card.xsl"?>
<!-- demo card -->
<CardList xml:id="1" > [...]
</CardList>
```

See Cautions section for more details.

You can select nodes using their tag names, by XPath, or by their attributes value (if of type ID, xml:id for example). The `getElementsbyTagName` and `getElementsbyTagNameNS` methods return a DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is

restricted to contain objects with the same tag name and/or namespace. The`selectByXPath` method also returns a DomNodeList object, but each node can have a different name.

```
getElementsByTagNameNS("message","http://schemas.xmlsoap.org/wsdl/")
```

Get the *message* nodes that have *http://schemas.xmlsoap.org/wsdl/* as the namespace.

```
getElementsByTagNameNS("message","*")
```

Get all the *message* nodes, regardless of the namespace they have.

```
getElementsByTagName("message")
```

Get all the *message* nodes that do not have any namespace.

```
selectByXPath("//xs:element",NULL)
```

Get all the xs:element nodes that has a namespace corresponding to prefix *xs* .

```
selectByXPath("//Card",NULL)
```

Get all the Card nodes that do not have any namespace.

```
getElementById("1")
```

Get the unique node whose attribute of type ID has a value of "1".

Node creation methods usage examples
Node creation methods usage examples for the `xml.DomDocument` class.

Creating a node for the DomDocument is done in two steps:

- Create the node.
- Add the node to the DomDocument.

Each time you create a node, you need to append it at the right place in the DomDocument. To add a node the document use the DomDocument management methods or the DomNode manipulation methods.

```
createNode("<LastName>PATTERSON</LastName><FirstName>Andrew</FirstName>")
```

Creates a structure of nodes.

```
createElement("CardList")
```

Produces

```
<CardList>
```

```
createElementNS("cny", "Company", "http://www.mysite.com/")
```

Produces `<cny:Company xmlns:cny="http://www.mysite.com/"/>` or `<cny:Company />`. See Cautions for more details.

```
createAttribute("Country")
```

Creates a *Country* attribute node.

- To set a value to the attribute, use the method setNodeValue of the xml.DomNode class.

- To add the attribute to an element node, use the method setAttributeNode of the xml.DomNode class.

```
createAttributeNS("tw","Town","http://www.mysite.com/cities")
```

Produces `xmlns:tw="http://www.mysite.com/cities" tw:Town=""`

- To set a value to the attribute use the method setNodeValue of the xml.DomNode class.
- To add the attribute to an element node use the method setAttributeNodeNS of the xml.DomNode class.
- For optimization reasons, the namespace is not written aside the attribute until the saving of the DomDocument.
- When accessing the element node, the namespace is not listed in the list of children. In the example above, `tw:Town=""` is in the list of children, not `xmlns:tw="http://www.mysite.com/cities"`.
- To access the namespace during the DomDocument building use the method normalize first. Normalize write the namespace declaration at the appropriate place. If there is no previous declaration, it will be accessible as an attribute of this element, otherwise it will be an attribute of one of the ancestors of the element.

```
createTextNode("My Company")
```

Creates a text node.

```
createComment("End of the card")
```

Produces`<!--End of the card-->`

```
createCDATASection("<website><a href=\"www.mysite.com\">My
Company</a></website>")
```

Produces`<![CDATA[<website><a href="www.mysite.com">My Company</a></website>]]>`

```
createEntityReference("title")
```

Creates the entity reference `&title`.

```
createProcessingInstruction("xml-stylesheet", "type=\"text/xsl\"
href=\"card.xsl\"")
```

Produces`<?xml-stylesheet type="text/xsl"href="card.xsl"?>`

```
createDocumentType("Card", NULL, NULL,"<!ELEMENT
Card (lastname, firstname, company, location)>")
```

Produces `<!DOCTYPE Card [ <!ELEMENT Card (lastname , firstname , company , location)>]>`

- Only inline DTD are supported. The DTD has to been inserted in the DomDocument at an appropriate place.

```
createDocumentFragment
```

Is a method that creates a lightweight DomDocument. It represents a subtree of nodes that do not need to conform to well-formed XML rules. This makes DocumentFragment easier to manipulate than a DomDocument.

```
for i=1 to 5
  let node = doc.createelement("Card")
  call root.appendchild(node) end for
```

This produces a subtree with 5 Card nodes that do not have any root node. Once the subtree is completed, it can be added to the DomDocument object like any other node.

HTML document usage example
The HTML language provides tags that allow the user to provide an embedded style sheet (the "`style`" tag) and to write embedded client side script (the "`script`" tag). According to the HTML 4.0 specification, the content of these tags must be managed as CDATA section.

> **Note:** For more information, see the HTML 4.0 specification.

Because HTML document management via the xml.DomDocument object provides HTML compliancy only (and not strict HTML management), there is a specific way to add these nodes inside a loaded HTML document:

1. Create an element node with the name of the tag to be created.
2. Append that element node to its parent.
3. Create a CDATASection node with the wanted embedded piece of style sheet or piece of script content.
4. Append the CDATASection to the previously created element node.

By following this procedure, the "`script`" and "`style`" tags content are recognized as `CDATA` section content and not `TEXT` section content and will be preserved. Other methods for adding nodes to the document manage text and therefore will not treat these types of content properly, resulting in invalid HTML code.

Example

```
IMPORT XML

MAIN

  DEFINE myDoc XML.DomDocument
  DEFINE myEltNode, myAttrNode, bodyNode, myCdataNode
 XML.DomNode
  DEFINE nodeLst XML.DomNodeList
  DEFINE i INTEGER

  TRY
    LET myDoc = XML.DomDocument.create()
    CALL myDoc.setFeature("enable-html-compliancy", 1)
    CALL myDoc.load("testHtml.html")

    LET myEltNode = myDoc.CreateElement("script")
    LET myCdataNode =
 myDoc.CreateCDATASection("document.write(\"CDATA\");")
    LET myAttrNode = myDoc.CreateAttribute("type")
    CALL myAttrNode.setNodeValue("text/javascript")

    LET nodeLst = myDoc.getElementsByTagName("body")
    LET docNode = nodeLst.getItem(1)

    CALL docNode.appendChild(myEltNode)
    CALL myEltNode.setAttributeNode(myAttrNode)
    CALL myEltNode.appendChild(myCdataNode)

  CATCH
    DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
    EXIT PROGRAM(-1)
  END TRY

END MAIN
```

Load and save methods usage examples
Load and save methods usage examples for the `xml.DomDocument` class.

You can load an existing xml document. Before loading an xml document you need to create the DomDocument object.

A DomDocument can load files using different URI: http://, https://, tcp://, tcps://, file://and alias://. Use `getErrorsCount()` and `getErrorDescription()` to display errors about the document loading.

```
load("data.xml")
load("http://www.w3schools.com/xml/cd_catalog.xml")
load("https://localhost:6394/ws/r/calculator?WSDL")
load("file:///data/cd_catalog.xml")
load("tcp://localhost:4242/")
load("tcps://localhost:4243/")
load("alias://demo")
```

where demo alias is defined in fglprofile as `ws.demo.url = "http://www.w3schools.com/xml/cd_catalog.xml"`

```
loadfromstring("<List> <elt>First element</elt>
 <elt>Second element</elt> <elt>Third element</elt> </List>")
```

Produces a subtree with a root node `List` and three nodes `elt` and three `textnode`.

A DomDocument can be saved at different URI beginning with: http://, https://, tcp://, tcps://,file:// and alias://. Use `getErrorsCount()` and `getErrorDescription()`to display errors about the document saving.

```
save("myfile.xml")
save("http://myserver:8080/data/save1.xml")
save("file:///data/save.xml")
save("tcp://localhost:4242/")
save("alias://test")
```

where `test` alias is defined in fglprofile as `ws.test.url = "http://localhost:8080/data/save3.xml"`

`saveToString` saves the DomDocument in a string. Use `getErrorsCount()` and `getErrorDescription()` to display errors about the document saving

`normalize` function emulates a DomDocument save and load. It can be called at any stage of the DomDocument building. This removes empty Text nodes and sets namespace declarations as if the document had been saved.

Cautions
Cautions when working with the `xml.DomDocument` class.

Whitespaces, line feeds and carriage returns between elements are represented as text nodes in memory. An XML document written in a single line and a human readable (pretty printed format) do not have the same representation in the DomDocument. Take this under account when navigating in the document.

If a DomNode is not attached to a DomDocument and not referenced by any variable it can be destroyed. If one child of this node is still referenced, this child is not destroyed but its parent and the others node of the subtree are destroyed. To check if a node is attached to a DomDocument use isAttached method.

DomDocument remains in memory if any of its node is still referenced in a variable.

DomDocument Features
A list of features for the `xml.DomDocument` class.

DomDocument features

Table 484: DomDocument Features

Name	Description
format-pretty-print	Formats the output by adding white space to produce a pretty-printed, indented, human-readable form. Possible values are TRUE or FALSE. Default value is FALSE.
comments	Defines whether the XML comments are kept during the load of a document into a DomDocument object. Possible values are TRUE or FALSE. Default value is TRUE.
whitespace-in-element-content	Defines whether XML Text nodes that can be considered "Ignorable" are kept during the load of an XML document into a DomDocument object. Possible values are TRUE or FALSE. Default value is TRUE.
cdata-sections	Defines whether XML CData nodes are kept or replaced by XML Text nodes during the load of an XML document into a DomDocument object. Possible values are TRUE or FALSE. Default value is TRUE.
expand-entity-references	Defines whether XML EntityReference nodes are kept or replaced during the load of an XML document into a DomDocument object. Possible values are TRUE or FALSE. Default value is FALSE. **Note:** See security issues with expand-entity-references.
validation-type	Defines what kind of validation should be performed. Possible values are: DTD, Schema. Default is Schema.
external-schemaLocation	Defines a list of namespace-qualified XML schemas to use for validation on a DomDocument object.

Name	Description
	Value is a space-separated string of one or several pairs of strings representing the namespace URI of the schema, followed by its location. Example: "http://tempuri/org/NS mySchema1.xsd http://www.mycompany.com mySchema2.xsd"
external-noNamespaceSchemaLocation	Defines a list of XML schemas to use for validation on a DomDocument object. Value is a space-separated string of one or several strings representing the location of a schema. Example: "mySchema1.xsd mySchema2.xsd"
schema-uriRecovery	Changes the schema location of an XML schema referenced by import tags in other schemas. Value is a space-separated string of one or several pairs of strings representing the original schema location followed by the new schema location Example: "http://www.w3.org/2001/xml.xsd myXML.xsd http://www.mycompany.com/GWS.xsd myGWS.xsd"
load-save-base64-string	Changes methods loadFromString() and saveToString() to handle Base64 strings. Parsing an XML document is done from a BASE64 encoded string, and saving an XML document results in a BASE64 encoded string. Possible values are TRUE or FALSE. Default is FALSE.
auto-id-attribute	Changes the parsing of an XML document in order to set all unqualified attributes named ID, Id, iD or id to be of type ID. They can then be retrieved with method getElementById() or with an XPath expression without calling setIdAttribute(). Possible values are TRUE or FALSE. Default is FALSE.
auto-id-qualified-attribute	Changes the parsing of an XML document in order to set all qualified attributes named ID, Id, iD or id to be of type ID. They can then be retrieved with method getElementById() or with an XPath expression without calling setIdAttributeNS().

Name	Description
	Possible values are TRUE or FALSE. Default is FALSE.
enable-html-compliancy	Changes methods to parse, normalize and save HTML document via the DomDocument object. Possible values are TRUE or FALSE. Default value is FALSE. The HTML parsing isn't namespace qualified, and document is considered as an XML document after loading. **Note:** This feature works only for HTML 4, it is not supported for HTML 5.

Security issues with expand-entity-references

When the `expand-entity-references` document feature is set to TRUE, XML entities referencing sensitive data may be included when loading the XML document with xml.DomDocument.load on page 2123, xml.DomDocument.loadFromPipe on page 2124, xml.DomDocument.loadFromString on page 2124, or xml.DomDocument.normalize on page 2125.

For example, in its DTD, the following XML file defines the `myref ENTITY` element referencing the `/etc/passwd` file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
      <!ELEMENT foo ANY >
      <!ENTITY myref SYSTEM "file:///etc/passwd" >
   ]>
<foo>&myref;</foo>
```

When loading this XML file with `expand-entity-references` set to TRUE, the resulting DOM document will have a `<foo>` node containing a text node with the content of `/etc/passwd`.

Examples
Examples involving the `xml.DomDocument` class.

Example 1 : Create a namespace qualified document with processing instructions

To create the following XML document on disk:

```
<?Target1 This is my first PI ?>
<MyPre:RootNode xmlns:MyPre="http://www.tempuri.org" >
<MyPre:Element />
</MyPre:RootNode>
<?Target2 This is my last PI ?>
```

Write the following code:

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE pi xml.DomNode
  DEFINE node xml.DomNode
  DEFINE elt xml.DomNode
```

```
  # Create a document with an initial namespace qualified root node
  LET doc = xml.DomDocument.CreateDocumentNS("MyPre", "RootNode", "http://
www.tempuri.org")
  # Create a Processing instruction
  LET pi = doc.createProcessingInstruction("Target1", "This is my first PI")
  # And add it at the begining of the document
  CALL doc.prependDocumentNode(pi)
  # Create another Processing instruction
  LET pi = doc.createProcessingInstruction("Target2", "This is my last PI")
  # And add it at the end of the document
  CALL doc.appendDocumentNode(pi)
  # Retrieve initial root node of the document
  LET elt = doc.getDocumentElement()
  # Create a new Element node
  LET node = doc.createElementNS("MyPre", "Element", "http://
www.tempuri.org")
  # And add it as child of the RootNode
  CALL elt.appendChild(node)
  # Then save the document on disk
  CALL doc.save("MyFile.xml")
END MAIN
```

Example 2 : Validating a document against XML schemas or a DTD

This code example loads one or more XML schemas or uses an embedded DTD to validate against a XML document:

```
IMPORT xml

MAIN
  DEFINE location STRING
  DEFINE xmlfile STRING
  DEFINE doc xml.DomDocument
  DEFINE ind INTEGER

  IF num_args()<2 THEN
    # Checks the number of arguments
    CALL ExitHelp()
  ELSE
    LET doc = xml.DomDocument.Create()
    LET xmlfile = arg_val(num_args())
    IF num_args() == 2 AND arg_val(1) == "-dtd" THEN
      # User choosed DTD validation
      CALL doc.setFeature("validation-type", "DTD")
    ELSE
      # User choosed XML Schema validation
      IF arg_val(1) == "-ns" THEN
        # Handle namespace qualified XML schemas
        IF num_args() MOD 2 != 0 THEN
          CALL ExitHelp()
        END IF
        FOR ind = 2 TO num_args()-1 STEP 2
          IF location IS NULL THEN
            LET location = arg_val(ind) || " " || arg_val(ind+1)
          ELSE
            LET location = location || " " || arg_val(ind) ||
                 " " || arg_val(ind+1)
          END IF
        END FOR
        TRY
          CALL doc.setFeature("external-schemaLocation", location)
        CATCH
          FOR ind = 1 TO doc.getErrorsCount()
```

```
            DISPLAY "Schema error ("||
ind||") :",doc.getErrorDescription(ind)
          END FOR
          EXIT PROGRAM (-1)
        END TRY
      ELSE
        # Handle unqualified XML schemas
        FOR ind = 1 TO num_args()-1
          IF location IS NULL THEN
            LET location = arg_val(ind)
          ELSE
            LET location = location || " " || arg_val(ind)
          END IF
        END FOR
        TRY
          CALL doc.setFeature("external-noNamespaceSchemaLocation",
 location)
        CATCH
          FOR ind = 1 TO doc.getErrorsCount()
            DISPLAY "Schema error ("||
ind||") :",doc.getErrorDescription(ind)
          END FOR
          EXIT PROGRAM (-1)
        END TRY
      END IF
    END IF
  END IF
  TRY
    # Load XML document from disk
    CALL doc.load(xmlfile)
  CATCH
    # Display errors if loading failed
    IF doc.getErrorsCount()>0 THEN
      FOR ind = 1 TO doc.getErrorsCount()
        DISPLAY "LOADING ERROR #"||ind||" :",doc.getErrorDescription(ind)
      END FOR
      EXIT PROGRAM(-1)
    ELSE
      DISPLAY "Unable to load file :",xmlfile
      EXIT PROGRAM(-1)
    END IF
  END TRY
  TRY
    # Validate loaded document
    LET ind = doc.validate()
    IF ind == 0 THEN
      # Successful validation
      DISPLAY "OK"
    ELSE
      # Display validation errors
      FOR ind = 1 TO doc.getErrorsCount()
        DISPLAY "VALIDATING ERROR #"||ind||" :",doc.getErrorDescription(ind)
      END FOR
      EXIT PROGRAM(-1)
    END IF
  CATCH
    DISPLAY "Unable to validate file :",xmlfile
    EXIT PROGRAM(-1)
  END TRY
END MAIN

# Display help

FUNCTION ExitHelp()
```

```
  DISPLAY "Validator < -dtd | -ns [namespace schema]+ | [schema]+ > xmlfile"
  EXIT PROGRAM
END FUNCTION
```

Example

```
$ fglrun Validator -dtd MyFile.xml
```

Validates XML file using DTD embedded in the XML file.

```
$ fglrun Validator Schema1.xsd Schema2.xsd MyFile.xml
```

Validates unqualified XML file using two unqualified XML schemas.

```
$ fglrun Validator -ns http://tempuri.org/one Schema1.xsd
   http://tempuri.org/two Schema2.xsd MyFile.xml
```

Validates namespace qualified XML file using two namespace qualified XML schemas.

The DomNode class

The `xml.DomNode` class provides methods to manipulate a node of a DomDocument object.

You can create a DomNode object using creation methods in the DomDocument class.

The `STATUS` variable is set to zero after a successful method call.

xml.DomNode methods
Methods for the `xml.DomNode` class.

Table 485: Object methods: Navigation

Name	Description
`getChildrenCount()` `  RETURNING cnt INTEGER`	Returns the number of child DomNode objects for a DomNode object.

`getChildNodeItem(`
`  pos INTEGER )`
`  RETURNING object xml.DomNode`
	Returns the child DomNode object at a given position for a DomNode object.

`getFirstChild()` `  RETURNING object xml.DomNode`	Returns the first child DomNode object for this XML Element DomNode object.
`getFirstChildElement()` `  RETURNING object xml.DomNode`	Returns the first XML Element child DomNode object for this DomNode object.
`getLastChild()` `  RETURNING object xml.DomNode`	Returns the last child DomNode object for a XML Element DomNode object.
`getLastChildElement()`	Returns the last child XML element DomNode object for this DomNode object.

Name	Description
RETURNING *object* xml.DomNode	
getNextSibling() RETURNING *object* xml.DomNode	Returns the DomNode object immediately following a DomNode object.
getNextSiblingElement() RETURNING *object* xml.DomNode	Returns the XML Element DomNode object immediately following a DomNode object.
getParentNode() RETURNING *object* xml.DomNode	Returns the parent DomNode object for this DomNode object.
getOwnerDocument() RETURNING *object* xml.DomDocument	Returns the DomDocument object containing this DomNode object.
getPreviousSibling() RETURNING *object* xml.DomNode	Returns the DomNode object immediately preceding a DomNode object.
getPreviousSiblingElement() RETURNING *object* xml.DomNode	Returns the XML Element DomNode object immediately preceding a DomNode object.
hasChildNodes() RETURNING *flag* INTEGER	Returns TRUE if a node has child nodes.

Table 486: Object methods: Manipulation

Name	Description
clone(*deep* INTEGER) RETURNING *object* xml.DomNode	Returns a duplicate DomNode object of a node.
appendChild(*node* xml.DomNode)	Adds a child DomNode object to the end of the child list for a DomNode object
appendChildElement(*name* STRING) RETURNING *object* xml.DomNode	Creates and adds a child XML Element node to the end of the list of child nodes for an XML Element DomNode object.
appendChildElementNS(*prefix* STRING, *name* STRING, *ns* STRING)	Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for an XML Element DomNode object.

Name	Description
RETURNING *object* xml.DomNode	
addNextSibling(*node* xml.DomNode)	Adds a DomNode object as the next sibling of a DomNode object.
addPreviousSibling(*node* xml.DomNode)	Adds a DomNode object as the previous sibling of a DomNode object.
insertBeforeChild(*node* xml.DomNode, *ref* xml.DomNode)	Inserts a DomNode object before an existing child DomNode object.
insertAfterChild(*node* xml.DomNode, *ref* xml.DomNode)	Inserts a DomNode object after an existing child DomNode object.
prependChild(*node* xml.DomNode)	Adds a child DomNode object to the beginning of the child list for a DomNode object.
prependChildElement(*name* STRING) RETURNING *object* xml.DomNode	Creates and adds a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object.
prependChildElementNS(*prefix* STRING, *name* STRING, *ns* STRING) RETURNING *object* xml.DomNode	Creates and adds a child namespace-qualified XML Element node to the beginning of the list of child nodes for an XML Element DomNode object.
removeAllChildren()	Removes all child DomNode objects from a DomNode object.
removeChild(*node* xml.DomNode)	Removes a child DomNode object from the list of child DomNode objects.
replaceChild(*new* xml.DomNode, *old* xml.DomNode)	Replaces an existing child DomNode with another child DomNode object.

Table 487: Object methods: Access

Name	Description
getLocalName()	Gets the local name for a DomNode object.

Name	Description
RETURNING *str* STRING	
getNamespaceURI() RETURNING *str* STRING	Returns the namespace URI for a DomNode object.
getNodeName() RETURNING *str* STRING	Gets the name for a DomNode object.
getNodeType() RETURNING *str* STRING	Gets the XML type for this DomNode object.
getNodeValue() RETURNING *str* STRING	Returns the value for a DomNode object.
getPrefix() RETURNING *str* STRING	Returns the prefix for a DomNode object.
isAttached() RETURNING *num* INTEGER	Returns whether the node is attached to the XML document.

Table 488: Object methods: Modifier

Name	Description
setNodeValue(*val* STRING)	Sets the node value for a DomNode object.
setPrefix(*prefix* STRING)	Sets the prefix for a DomNode object.
toString() RETURNING STRING	Returns a string representation of a DomNode object.

Table 489: Object methods: Attribute

Name	Description
hasAttribute(*name* STRING) RETURNING *flag* INTEGER	Checks whether an XML Element DomNode object has the XML Attribute specified by a specified name.
hasAttributeNS(*name* STRING, *ns* STRING)	Checks whether a namespace qualified XML Attribute of a given name is carried by an XML Element DomNode object.

Name	Description
`RETURNING flag INTEGER`	
`getAttributesCount()` `RETURNING num INTEGER`	Returns the number of XML Attribute DomNode objects on this XML Element DomNode object.

`getAttributeNode(`
`name STRING )`
`RETURNING object xml.DomNode`
	Returns an XML Attribute DomNode object for an XML Element DomNode object

`getAttributeNodeItem(`
`pos INTEGER )`
`RETURNING object xml.DomNode`
	Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object.

`getAttributeNodeNS(`
`name STRING,`
`ns STRING )`
`RETURNING object xml.DomNode`
	Returns a namespace-qualified XML Attribute DomNode object for an XML Element DomNode object

`getAttribute(`
`name STRING )`
`RETURNING value STRING`
	Returns the value of an XML Attribute for an XML Element DomNode object

`getAttributeNS(`
`name STRING,`
`ns STRING )`
`RETURNING value STRING`
	Returns the value of a namespace qualified XML Attribute for an XML Element DomNode object

`hasAttributes()` `RETURNING flag INTEGER`	Identifies whether a node has XML Attribute nodes.

`setAttribute(`
`name STRING,`
`value STRING )`
	Sets (or resets) an XML Attribute for an XML Element DomNode object.

`setAttributeNode(` `node xml.DomNode )`	Sets (or resets) an XML Attribute DomNode object to an XML Element DomNode object.
`setAttributeNodeNS(` `node xml.DomNode )`	Sets (or resets) a namespace-qualified XML Attribute DomNode object to an XML Element DomNode object.

`setAttributeNS(`
`prefix STRING,`
`name STRING,`
`ns STRING,`
	Sets (or resets) a namespace-qualified XML Attribute for an XML Element DomNode object.

Name	Description
value STRING)	
setIdAttribute(*name* STRING, *isId* INTEGER)	Declare (or undeclare) the XML Attribute of given name to be of type ID.
setIdAttributeNS(*name* STRING, *ns* STRING, *isId* INTEGER)	Declare (or undeclare) the namespace-qualified XML Attribute of given name and namespace to be of type ID.
removeAttribute(*name* STRING)	Removes an XML Attribute for an XML Element DomNode object.
removeAttributeNS(*name* STRING, *ns* STRING)	Removes a namespace qualified XML Attribute for an XML Element DomNode object

Table 490: Object methods: Search

Name	Description
getElementsByTagName(*name* STRING) RETURNING *list* xml.DomNodeList	Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name.
getElementsByTagNameNS(*name* STRING, *ns* STRING) RETURNING *list* xml.DomNodeList	Returns a DomNodeList object containing all namespace-qualified XML Element DomNode objects with the same tag name and namespace.
isDefaultNamespace(*ns* STRING) RETURNING *flag* INTEGER	Checks whether the specified namespace URI is the default namespace.
lookupNamespaceURI(*prefix* STRING) RETURNING *ns* STRING	Looks up the namespace URI associated to a prefix, starting from a specified node.
lookupPrefix(*ns* STRING) RETURNING *prefix* STRING	Looks up the prefix associated to a namespace URI, starting from the specified node.
selectByXPath(*expr* STRING, *NamespacesList* ...)	Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression.

Name	Description
RETURNING *list* xml.DomNodeList	

xml.DomNode.addPreviousSibling

Adds a DomNode object as the previous sibling of a DomNode object.

Syntax

```
addPreviousSibling(
   node xml.DomNode )
```

1. *node* is the node to add.

Usage

Adds a DomNode object as the previous sibling of this DomNode object; *node* is the node to add.

Caution: Not part of W3C API.

The DomNode object node must be the child of an element or document node; otherwise the operation fails.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.addNextSibling

Adds a DomNode object as the next sibling of a DomNode object.

Syntax

```
addNextSibling(
  node xml.DomNode )
```

1. *node* is the node to add.

Usage

Adds a DomNode object as the next sibling of this DomNode object; *node* is the node to add.

Caution: Not part of W3C API.

The DomNode object node must be the child of an element or document node, otherwise the operation fails.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.appendChild

Adds a child DomNode object to the end of the child list for a DomNode object

Syntax

```
appendChild(
  node xml.DomNode )
```

1. *node* is the node to add

Usage

Adds a child DomNode object to the end of the child list for this DomNode object.

The DomNode object node must be the child of an element or document node; otherwise the operation fails.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.appendChildElement

Creates and adds a child XML Element node to the end of the list of child nodes for an XML Element DomNode object.

Syntax

```
appendChildElement(
   name STRING )
  RETURNING object xml.DomNode
```

1. *name* is the XML Element name.

Usage

Creates and adds a child XML Element node to the end of the list of child nodes for this XML Element DomNode object.

> **Caution:** Not part of W3C API.

Returns the XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.appendChildElementNS

Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for an XML Element DomNode object.

Syntax

```
appendChildElementNS(
   prefix STRING,
   name STRING,
   ns STRING )
  RETURNING object xml.DomNode
```

1. *prefix* is the prefix of the XML Element to add.
2. *name* is the name of the XML Element to add.
3. *ns* is the namespace URI of the XML Element to add.

Usage

Creates and adds a child namespace qualified XML Element node to the end of the list of child nodes for this XML Element DomNode object.

> **Caution:** Not part of W3C API.

Returns the XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.clone

Returns a duplicate DomNode object of a node.

Syntax

```
clone(
   deep INTEGER )
  RETURNING object xml.DomNode
```

1. *deep* is a boolean. If *deep* is TRUE, child DomNode objects are cloned too; otherwise only the DomNode itself is cloned.

Usage

Returns a duplicate DomNode object of this node. If *deep* is TRUE, child DomNode objects are cloned too; otherwise only the DomNode itself is cloned.

Returns a copy of this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getAttribute

Returns the value of an XML Attribute for an XML Element DomNode object

Syntax

```
getAttribute(
   name STRING )
  RETURNING value STRING
```

1. *name* is the name of the XML attribute to retrieve.

Usage

Returns the value of an XML Attribute for this XML Element DomNode object. where *name* is the name of the XML attribute to retrieve.

Returns the XML Attribute value, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getAttributeNode

Returns an XML Attribute DomNode object for an XML Element DomNode object

Syntax

```
getAttributeNode(
   name STRING )
  RETURNING object xml.DomNode
```

1. *name* is the name of the attribute to retrieve.

Usage

Returns an XML Attribute DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getAttributeNodeItem

Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object.

Syntax

```
getAttributeNodeItem(
   pos INTEGER )
  RETURNING object xml.DomNode
```

1. *pos* is the position of the node to return.

Usage

Returns the XML Attribute DomNode object at a given position on this XML Element DomNode object, where *pos* is the position of the node to return (Index starts at 1).

Returns the XML Attribute DomNode object at the given position, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getAttributeNodeNS

Returns a namespace-qualified XML Attribute DomNode object for an XML Element DomNode object

Syntax

```
getAttributeNodeNS(
   name STRING,
   ns STRING )
  RETURNING object xml.DomNode
```

1. *name* is the name of the XMLAttribute to retrieve.
2. *ns* is the namespace URI of the XML Attribute to retrieve.

Usage

Returns a namespace-qualified XML Attribute DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getAttributeNS

Returns the value of a namespace qualified XML Attribute for an XML Element DomNode object

Syntax

```
getAttributeNS(
   name STRING,
   ns STRING )
```

```
   RETURNING value STRING
```

1. *name* is the name.
2. *ns* is the namespace URI of the XML Attribute to retrieve

Usage

Returns the value of a namespace qualified XML Attribute for this XML Element DomNode object, where *name* is the name and *ns* is the namespace URI of the XML Attribute to retrieve.

Returns the XML Attribute value, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getAttributesCount

Returns the number of XML Attribute DomNode objects on this XML Element DomNode object.

Syntax

```
getAttributesCount()
  RETURNING num INTEGER
```

Usage

Returns the number of XML Attribute DomNode objects on this XML Element DomNode object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getChildNodeItem

Returns the child DomNode object at a given position for a DomNode object.

Syntax

```
getChildNodeItem(
   pos INTEGER )
  RETURNING object xml.DomNode
```

1. *pos* is the position of the child node in the collection.

Usage

Returns the child DomNode object at a given position for this DomNode object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getChildrenCount

Returns the number of child DomNode objects for a DomNode object.

Syntax

```
getChildrenCount()
  RETURNING cnt INTEGER
```

Usage

Returns the number of child DomNode objects for this DomNode object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getElementsByTagName

Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name.

Syntax

```
getElementsByTagName(
   name STRING )
  RETURNING list xml.DomNodeList
```

1. *name* is the name of the XML Element tag to match or "*" to match all tags.

Usage

Returns a DomNodeList object containing all XML Element DomNode objects with the same tag name, or NULL; *name* is the name of the XML Element tag to match, or "*" to match all tags.

The `getElementsByTagName` and `getElementsByTagNameNS` methods return a DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is restricted to contain objects with the same tag name and/or namespace.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getElementsByTagNameNS

Returns a DomNodeList object containing all namespace-qualified XML Element DomNode objects with the same tag name and namespace.

Syntax

```
getElementsByTagNameNS(
   name STRING,
   ns STRING )
  RETURNING list xml.DomNodeList
```

1. *name* is the name of the XML Element tag to match or "*" to match all tags.
2. *ns* is the namespace URI of the XML Element tag to match or "*" to match any namespace.

Usage

Returns a DomNodeList object containing all namespace-qualified XML Element DomNode objects with the same tag name and namespace, or NULL. *name* is the name of the XML Element tag to match, or "*" to match all tags; *ns* is the namespace URI of the XML Element tag to match, or "*" to match any namespace.

The `getElementsByTagName` and `getElementsByTagNameNS` methods return a DomNodeList object, unlike the other methods that return a DomNode object. The DomNodeList is restricted to contain objects with the same tag name and/or namespace.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getFirstChild

Returns the first child DomNode object for this XML Element DomNode object.

Syntax

```
getFirstChild()
  RETURNING object xml.DomNode
```

Usage

Returns the first child DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getFirstChildElement

Returns the first XML Element child DomNode object for this DomNode object.

Syntax

```
getFirstChildElement()
  RETURNING object xml.DomNode
```

Usage

Returns the first XML Element child DomNode object for this DomNode object, or NULL.

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getLastChild

Returns the last child DomNode object for a XML Element DomNode object.

Syntax

```
getLastChild()
  RETURNING object xml.DomNode
```

Usage

Returns the last child DomNode object for this XML Element DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getLastChildElement

Returns the last child XML element DomNode object for this DomNode object.

Syntax

```
getLastChildElement()
  RETURNING object xml.DomNode
```

Usage

Returns the last child XML element DomNode object for this DomNode object, or NULL.

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getLocalName

Gets the local name for a DomNode object.

Syntax

```
getLocalName()
  RETURNING str STRING
```

Usage

Gets the local name for this DomNode object. If DomNode has a qualified name, only the local part is returned.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getNamespaceURI

Returns the namespace URI for a DomNode object.

Syntax

```
getNamespaceURI()
  RETURNING str STRING
```

Usage

Returns the namespace URI for this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getNextSibling

Returns the DomNode object immediately following a DomNode object.

Syntax

```
getNextSibling()
  RETURNING object xml.DomNode
```

Usage

Returns the DomNode object immediately following this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getNextSiblingElement

Returns the XML Element DomNode object immediately following a DomNode object.

Syntax

```
getNextSiblingElement()
  RETURNING object xml.DomNode
```

Usage

Returns the XML Element DomNode object immediately following this DomNode object, or NULL.

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getNodeName

Gets the name for a DomNode object.

Syntax

```
getNodeName()
  RETURNING str STRING
```

Usage

Gets the name for this DomNode object; returns the qualified name of this DomNode object, or NULL. If DomNode does not have a qualified name, the local part is returned.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getNodeType

Gets the XML type for this DomNode object.

Syntax

```
getNodeType()
  RETURNING str STRING
```

Usage

Gets the XML type for this DomNode object; returns one of the XML DomNode types, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getNodeValue

Returns the value for a DomNode object.

Syntax

```
getNodeValue()
  RETURNING str STRING
```

Usage

Returns the value for this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getOwnerDocument

Returns the DomDocument object containing this DomNode object.

Syntax

```
getOwnerDocument()
  RETURNING object xml.DomDocument
```

Usage

Returns the DomDocument object containing this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getParentNode

Returns the parent DomNode object for this DomNode object.

Syntax

```
getParentNode()
  RETURNING object xml.DomNode
```

Usage

Returns the parent DomNode object for this DomNode object, or NULL. In the case of a DomDocument node, this method will return NULL (parent is not a DomNode object) but isAttached() will return TRUE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getPrefix

Returns the prefix for a DomNode object.

Syntax

```
getPrefix()
  RETURNING str STRING
```

Usage

Returns the prefix for this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getPreviousSibling

Returns the DomNode object immediately preceding a DomNode object.

Syntax

```
getPreviousSibling()
  RETURNING object xml.DomNode
```

Usage

Returns the DomNode object immediately preceding this DomNode object, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.getPreviousSiblingElement

Returns the XML Element DomNode object immediately preceding a DomNode object.

Syntax

```
getPreviousSiblingElement()
  RETURNING object xml.DomNode
```

Usage

Returns the XML Element DomNode object immediately preceding this DomNode object, or NULL.

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.hasAttribute

Checks whether an XML Element DomNode object has the XML Attribute specified by a specified name.

Syntax

```
hasAttribute(
   name STRING )
  RETURNING flag INTEGER
```

1. *name* is the object name to check.

Usage

Checks whether this XML Element DomNode object has the XML Attribute specified by *name*. Returns TRUE if an XML Attribute with the given name is carried by this XML Element DomNode object, otherwise returns FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.hasAttributeNS

Checks whether a namespace qualified XML Attribute of a given name is carried by an XML Element DomNode object.

Syntax

```
hasAttributeNS(
   name STRING,
   ns STRING )
  RETURNING flag INTEGER
```

1. *name* the name of the XMLAttribute to check
2. *ns* the namespace URI of the XML Attribute to check.

Usage

Checks whether a namespace qualified XML Attribute of a given name is carried by this XML Element DomNode object, where *name* the name of the XMLAttribute to check; *ns* the namespace URI of the XML Attribute to check. Returns TRUE if an XML Attribute with the given name and namespace URI is carried by this XML Element DomNode object, otherwise returns FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.hasAttributes

Identifies whether a node has XML Attribute nodes.

Syntax

```
hasAttributes()
  RETURNING flag INTEGER
```

1. *flag* acts as a boolean.

Usage

Returns TRUE if this node has XML Attribute nodes; otherwise returns FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.hasChildNodes

Returns TRUE if a node has child nodes.

Syntax

```
hasChildNodes()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if this node has child nodes;otherwise, returns FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.insertAfterChild

Inserts a DomNode object after an existing child DomNode object.

Syntax

```
insertAfterChild(
   node xml.DomNode,
   ref xml.DomNode )
```

1. *node* is the node to insert.
2. *ref* is the reference node (the node before which the new node must be inserted).

Usage

Inserts a DomNode object after an existing child DomNode object; *node* is the node to insert, *ref* is the reference node (the node before which the new node must be inserted).

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.insertBeforeChild

Inserts a DomNode object before an existing child DomNode object.

Syntax

```
insertBeforeChild(
   node xml.DomNode,
   ref xml.DomNode )
```

1. *node* is the node to insert.
2. *ref* is the reference node (the node before which the new node must be inserted).

Usage

Inserts a DomNode object before an existing child DomNode object; *node* is the node to insert, *ref* is the reference node (the node before which the new node must be inserted).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.isAttached

Returns whether the node is attached to the XML document.

Syntax

```
isAttached()
  RETURNING num INTEGER
```

Usage

Returns whether the node is attached to the XML document.

Caution: Not part of W3C API.

Returns TRUE if this DomNode object is attached to a DomDocument object as a child and was not removed later on; otherwise it returns FALSE.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.isDefaultNamespace

Checks whether the specified namespace URI is the default namespace.

Syntax

```
isDefaultNamespace(
   ns STRING )
  RETURNING flag INTEGER
```

1. *ns* is the namespace URI to look for.

Usage

Checks whether the specified namespace URI is the default namespace, where *ns* is the namespace URI to look for. Returns TRUE if the given namespace is the default namespace, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.lookupNamespaceURI

Looks up the namespace URI associated to a prefix, starting from a specified node.

Syntax

```
lookupNamespaceURI(
   prefix STRING )
  RETURNING ns STRING
```

1. *prefix* is the prefix to look for.

Usage

Looks up the namespace URI associated to a prefix, starting from this node, where *prefix* is the prefix to look for; if NULL, the default namespace URI will be returned. Returns a namespace URI, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.lookupPrefix

Looks up the prefix associated to a namespace URI, starting from the specified node.

Syntax

```
lookupPrefix(
   ns STRING )
  RETURNING prefix STRING
```

1. *ns* is the namespace URI to look for.

Usage

Looks up the prefix associated to a namespace URI, starting from this node, where *ns* is the namespace URI to look for. Returns the prefix associated to this namespace URI, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.prependChild

Adds a child DomNode object to the beginning of the child list for a DomNode object.

Syntax

```
prependChild(
   node xml.DomNode )
```

1. *node* is the node to add.

Usage

Adds a child DomNode object to the beginning of the child list for this DomNode object ; *node* is the node to add.

Caution: Not part of W3C API.

The DomNode object node must be the child of an element or document node, otherwise the operation fails.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.prependChildElement

Creates and adds a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object.

Syntax

```
prependChildElement(
   name STRING )
  RETURNING object xml.DomNode
```

1. *name* is the name of the XML element to add.

Usage

Creates and adds a child XML Element node to the beginning of the list of child nodes for this XML Element DomNode object; *name* is the name of the XML element to add.

Returns the XML Element DomNode object, or NULL.

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.prependChildElementNS

Creates and adds a child namespace-qualified XML Element node to the beginning of the list of child nodes for an XML Element DomNode object.

Syntax

```
prependChildElementNS(
```

```
    prefix STRING,
    name STRING,
    ns STRING )
  RETURNING object xml.DomNode
```

1. *prefix* is the prefix of the XML Element to add.
2. *name* is the name of the XML Element to add.
3. *ns* is the namespace URI of the XML Element to add.

Usage

Creates and adds a child namespace-qualified XML Element node to the beginning of the list of child nodes for this XML Element DomNode object.

Returns the XML Element DomNode object, or NULL.

Note: Not part of W3C API.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.removeAllChildren

Removes all child DomNode objects from a DomNode object.

Syntax

```
removeAllChildren()
```

Usage

Removes all child DomNode objects from this DomNode object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.removeAttribute

Removes an XML Attribute for an XML Element DomNode object.

Syntax

```
removeAttribute(
  name STRING )
```

1. *name* is the name of the XML attribute to remove.

Usage

Removes an XML Attribute for this XML Element DomNode object, where *name* is the name of the XML attribute to remove. Status is updated with an error code.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.removeAttributeNS

Removes a namespace qualified XML Attribute for an XML Element DomNode object

Syntax

```
removeAttributeNS(
   name STRING,
   ns STRING )
```

1. *name* is the name of the XML Attribute to remove.
2. *ns* is the namespace URI of the XML Attribute to remove.

Usage

Removes a namespace qualified XML Attribute for this XML Element DomNode object, where *name* is the name and *ns* is the namespace URI of the XML Attribute to remove.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.removeChild

Removes a child DomNode object from the list of child DomNode objects.

Syntax

```
removeChild(
   node xml.DomNode )
```

1. *node* is the node to remove.

Usage

Removes a child DomNode object from the list of child DomNode objects, where *node* is the node to remove.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.replaceChild

Replaces an existing child DomNode with another child DomNode object.

Syntax

```
replaceChild(
   new xml.DomNode,
   old xml.DomNode )
```

1. *new* is the replacement child.
2. *old* is the child to be replaced.

Usage

Replaces an existing child DomNode with another child DomNode object, where *old* is the child to be replaced and *new* is the replacement child.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.selectByXPath

Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression.

Syntax

```
selectByXPath(
   expr STRING,
   NamespacesList ...)
  RETURNING list xml.DomNodeList
```

1. *expr* is the XPath 1.0 expression.
2. *NamespacesList* is a list of prefixes bounded to namespaces in order to resolve qualified names in the XPath expression.

Usage

Returns a DomNodeList object containing all DomNode objects matching an XPath 1.0 expression; *expr* is the XPath 1.0 expression, *NamespacesList* is a list of prefixes bounded to namespaces in order to resolve qualified names in the XPath expression. This list must be filled with an even number of arguments, representing the prefix and it corresponding namespace.

Caution: Not part of W3C API.

Example

```
selectByXPath(
  "../../d:Record/*[last()]",
  "d",
  "http://defaultnamespace")
selectByXPath(
  "ns:Record",
  NULL)
selectByXPath(
  "ns1:Records/ns2:Record",
  "ns1",
   "http://namespace1",
   "ns2",
   "http://namespace2")
```

`selectByXPath("ns1:Record", "ns1")` is invalid because the namespace definition is missing.

If the namespaces list is NULL, the prefixes and namespaces defined in the document itself are used if available.

A namespace must be an absolute URI (ex 'http://', 'file://').

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setAttribute

Sets (or resets) an XML Attribute for an XML Element DomNode object.

Syntax

```
setAttribute(
   name STRING,
   value STRING )
```

1. *name* is the name of the XML Attribute.
2. *val* is the value of the XML Attribute.

Usage

Sets (or resets) an XML Attribute for this XML Element DomNode object, where *name* is the name of the XML Attribute and *val* is the value of the XML Attribute.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setAttributeNode

Sets (or resets) an XML Attribute DomNode object to an XML Element DomNode object.

Syntax

```
setAttributeNode(
   node xml.DomNode )
```

1. *node* is the XML Attribute DomNode object to set.

Usage

Sets (or resets) an XML Attribute DomNode object to an XML Element DomNode object, where *node* is the XML Attribute DomNode object to set.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setAttributeNodeNS

Sets (or resets) a namespace-qualified XML Attribute DomNode object to an XML Element DomNode object.

Syntax

```
setAttributeNodeNS(
   node xml.DomNode )
```

1. *node* is the XML Attribute DomNode object to set.

Usage

Sets (or resets) a namespace-qualified XML Attribute DomNode object to an XML Element DomNode object, where *node* is the XML Attribute DomNode object to set.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setAttributeNS

Sets (or resets) a namespace-qualified XML Attribute for an XML Element DomNode object.

Syntax

```
setAttributeNS(
   prefix STRING,
   name STRING,
   ns STRING,
```

```
   value STRING )
```

1. *prefix* is the prefix of the XMLAttribute.
2. *name* is the name of the XML Attribute.
3. *ns* is the namespace URI of the XML Attribute.
4. *val* is the value of the XML Attribute.

Usage

Sets (or resets) a namespace-qualified XML Attribute for this XML Element DomNode object, where *prefix* is the prefix of the XMLAttribute, *name* is the name of the XML Attribute, *ns* is the namespace URI of the XML Attribute, and *val* is the value of the XML Attribute.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setIdAttribute

Declare (or undeclare) the XML Attribute of given name to be of type ID.

Syntax

```
setIdAttribute(
   name STRING,
   isId INTEGER )
```

1. *name* is the name of the XML Attribute to set.
2. *isId* declares whether the attribute is a user-determined ID attribute.

Usage

Declare (or undeclare) the XML Attribute of given name to be of type ID. Use the value TRUE for the parameter *isID* to declare that attribute for being a user-determined ID attribute, otherwise returns FALSE.

This affects the behavior of getElementById.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setIdAttributeNS

Declare (or undeclare) the namespace-qualified XML Attribute of given name and namespace to be of type ID.

Syntax

```
setIdAttributeNS(
   name STRING,
   ns STRING,
   isId INTEGER )
```

1. *name* is the name of the XML Attribute to set.
2. *ns* is the namespace URI of the XML Attribute to set.
3. *isId* declares whether the attribute is a user-determined ID attribute.

Usage

Declare (or undeclare) the namespace-qualified XML Attribute of given name and namespace to be of type ID. Use the value TRUE for the parameter *isID* to declare that attribute for being a user-determined ID attribute, otherwise returns FALSE.

This affects the behavior of getElementById.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setNodeValue

Sets the node value for a DomNode object.

Syntax

```
setNodeValue(
   val STRING )
```

1. *val* is the node value.

Usage

Sets the node value for this DomNode object, where *val* is the node value.

This method should only be used for nodes that are not parent of other nodes, which means it can be sued for a node of type:

- ATTRIBUTE_NODE
- TEXT_NODE
- CDATA_SECTION_NODE
- PROCESSING_INSTRUCTION_NODE
- COMMENT_NODE

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.setPrefix

Sets the prefix for a DomNode object.

Syntax

```
setPrefix(
  prefix STRING )
```

1. *prefix* is the prefix for this DomNode object.

Usage

Sets the *prefix* for this DomNode object.

This method is only valid on namespace qualified Element or Attribute nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNode.toString

Returns a string representation of a DomNode object.

Syntax

```
toString()
  RETURNING STRING
```

Usage

Returns a string representation of this DomNode object, or NULL

Caution: Not part of W3C API.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

DomNode types
List of types for the `xml.DomNode` class.

Table 491: DomNode types

Type	Description
ELEMENT_NODE	The DomNode is an XML Element node.
ATTRIBUTE_NODE	The DomNode is an XML Attribute node.
TEXT_NODE	The DomNode is an XML Text node.
CDATA_SECTION_NODE	The DomNode is an XML CData Section node.
ENTITY_REFERENCE_NODE	The DomNode is an XML Entity Reference node.
PROCESSING_INSTRUCTION_NODE	The DomNode is an XML Processing Instruction node.
COMMENT_NODE	The DomNode is an XML Comment node.
DOCUMENT_TYPE_NODE	The DomNode is an XML DTD node.
DOCUMENT_FRAGMENT_NODE	The DomNode is an XML Document Fragment node.

Examples
Examples involving the `xml.DomNode` class.

Example Counting the number of nodes in an XML document

This code example counts the number of nodes of each type.

```
IMPORT XML

DEFINE nbElt INTEGER
DEFINE nbAttr INTEGER
DEFINE nbComment INTEGER
DEFINE nbPI INTEGER
DEFINE nbTxt INTEGER
DEFINE nbCData INTEGER

MAIN
  DEFINE document xml.DomDocument
  DEFINE ind INTEGER
```

```
  # Handle arguments
  IF num_args() !=1 THEN
    CALL ExitHelp()
  END IF
  # Create document, load it, and count the nodes
  LET document = xml.DomDocument.Create()
  CALL document.load(arg_val(1))
  CALL CountDoc(document)
  # Display result
  DISPLAY "Results: "
  DISPLAY " Elements: ",nbElt
  DISPLAY " Attributes:",nbAttr
  DISPLAY " Comments: ",nbComment
  DISPLAY " PI: ",nbPI
  DISPLAY " Texts: ",nbTxt
  DISPLAY " CData: ",nbCData
END MAIN

FUNCTION CountDoc(d)
  DEFINE d xml.DomDocument
  DEFINE n xml.DomNode
  LET n = d.getFirstDocumentNode()
  WHILE (n IS NOT NULL )
    CALL Count(n)
    LET n = n.getNextSibling()
  END WHILE
END FUNCTION

FUNCTION Count(n)
  DEFINE n xml.DomNode
  DEFINE child xml.DomNode
  DEFINE next xml.DomNode
  DEFINE node xml.DomNode
  DEFINE ind INTEGER
  DEFINE name STRING
  IF n IS NOT NULL THEN
    IF n.getNodeType() == "COMMENT_NODE" THEN
      LET nbComment = nbComment + 1
    END IF
    IF n.getNodeType() == "ATTRIBUTE_NODE" THEN
      LET nbAttr = nbAttr + 1
    END IF
    IF n.getNodeType() == "PROCESSING_INSTRUCTION_NODE " THEN
      LET nbPI = nbPI + 1
    END IF
    IF n.getNodeType() == "ELEMENT_NODE" THEN
      LET nbElt = nbElt + 1
    END IF
    IF n.getNodeType() == "TEXT_NODE" THEN
      LET nbTxt = nbTxt +1
    END IF
    IF n.getNodeType() == "CDATA_SECTION_NODE" THEN
      LET nbCData = nbCData + 1
    END IF
    IF n.hasChildNodes() THEN
      LET name = n.getLocalName()
      LET child = n.getFirstChild()
      WHILE (child IS NOT NULL )
        CALL Count(child)
        LET child = child.getNextSibling()
      END WHILE
    END IF
    IF n.hasAttributes() THEN
      FOR ind = 1 TO n.getAttributesCount()
```

```
        LET node = n.getAttributeNodeItem(ind)
        CALL Count(node)
      END FOR
    END IF
  END IF
END FUNCTION

FUNCTION ExitHelp()
  DISPLAY "DomCount <xml>"
  EXIT PROGRAM
END FUNCTION
```

The DomNodeList class

The `xml.DomNodeList` class provides methods to manipulate a list of DomNode objects.

You can create a DomNodeList object using selection methods in the DomDocument and DomNode classes. The relationship between the DomNode objects in the list depends on the method used to create the DomNodeList object.

The `STATUS` variable is set to zero after a successful method call.

xml.DomNodeList methods

Methods for the `xml.DomNodeList` class.

Table 492: Object methods

Name	Description
`getCount()` `  RETURNING num INTEGER`	Returns the number of DomNode objects in a DomNodeList object.

`getItem(`
`  pos INTEGER)`
`  RETURNING object xml.DomNode`
	Returns the DomNode object at a given position in a DomNodeList object.

xml.DomNodeList.getCount

Returns the number of DomNode objects in a DomNodeList object.

Syntax

```
getCount()
  RETURNING num INTEGER
```

Usage

Returns the number of DomNode objects in a DomNodeList object.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.DomNodeList.getItem

Returns the DomNode object at a given position in a DomNodeList object.

Syntax

```
getItem(
  pos INTEGER)
```

```
  RETURNING object xml.DomNode
```

1. *pos* is the position of the DomNode object to return (index starts at 1).

Usage

Returns the DomNode object at the given position in this DomNodeList object, where *pos* is the position of the DomNode object to return (Index starts at 1).

Returns NULL when no DomNode object is at the given position.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The streaming API for XML (StAX) classes

The streaming API for XML (StAX) classes use streaming while managing XML documents.

- CLASS StaxWriter
  - Features
  - Example
- CLASS StaxReader
  - Event types
  - Features
  - Example

The StaxWriter class

The xml.StaxWriter class provides methods compatible with StAX (Streaming API for XML) for writing XML documents.

The STATUS variable is set to zero after a successful method call.

xml.StaxWriter methods

Methods for the xml.StaxWriter class.

Table 493: Class methods: Creation

Name	Description
`xml.StaxWriter.create()` `  RETURNING object xml.StaxWriter`	Constructor of a StaxWriter object.

Table 494: Object methods: Configuration

Name	Description

`getFeature(`
`  feature STRING )`
`  RETURNING str STRING`
	Gets a feature of a StaxWriter object.

`setFeature(`
`  feature STRING,`
`  value STRING )`
	Sets a feature of a StaxWriter object.

Table 495: Object methods: Output

Name	Description
`close()`	Closes the StaxWriter streaming, and releases all associated resources.
`writeTo(` `  url STRING )`	Sets the output stream of the StaxWriter object to a file or an URL, and starts the streaming.
`writeToDocument(` `  doc xml.DomDocument )`	Sets the output stream of the StaxWriter object to an xml.DomDocument object, and starts the streaming.
`writeToPipe(` `  cmd STRING )`	Sets the output stream of the StaxWriter object to a PIPE, and starts the streaming.
`writeToText(` `  txt TEXT )`	Sets the output stream of the StaxWriter object to a TEXT large object, and starts the streaming.

Table 496: Object methods: Document

Name	Description
`dtd(` `  data STRING )`	Writes a DTD to the StaxWriter stream.
`endDocument()`	Closes any open tags and writes corresponding end tags.

`startDocument(`
`  encoding STRING,`
`  version STRING,`
`  standalone INTEGER )`
	Writes an XML declaration to the StaxWriter stream.

Table 497: Object methods: Namespace

Name	Description
`declareDefaultNamespace(` `  defaultNS STRING )`	Binds a namespace URI to the default namespace, and forces the output of the default XML namespace definition to the StaxWriter stream.

`declareNamespace(`
`  prefix STRING,`
`  ns STRING )`
	Binds a namespace URI to a prefix, and forces the output of the XML namespace definition to the StaxWriter stream.

`setDefaultNamespace(`	Binds a namespace URI to the default namespace.

Name	Description
defaultNS STRING)	
setPrefix(*prefix* STRING, *ns* STRING)	Binds a namespace URI to a prefix.

Table 498: Object methods: Node

Name	Description
attribute(*name* STRING, *value* STRING)	Writes an XML attribute to the StaxWriter stream.
attributeNS(*name* STRING, *ns* STRING, *value* STRING)	Writes an XML namespace qualified attribute to the StaxWriter stream.
cdata(*data* STRING)	Writes an XML CData to the StaxWriter stream.
characters(*text* STRING)	Writes an XML text to the StaxWriter stream.
comment(*data* STRING)	Writes an XML comment to the StaxWriter stream.
emptyElement(*name* STRING)	Writes an empty XML element to the StaxWriter stream.
emptyElementNS(*name* STRING, *ns* STRING)	Writes an empty namespace qualified XML element to the StaxWriter stream.
endElement()	Writes an end tag to the StaxWriter stream relying on the internal state to determine the prefix and local name of the last START_ELEMENT.
entityRef(*name* STRING)	Writes an XML EntityReference to the StaxWriter stream.
processingInstruction(*target* STRING,	Writes an XML ProcessingInstruction to the StaxWriter stream

Name	Description
data STRING)	
startElement(*name* STRING)	Writes an XML start element to the StaxWriter stream.
startElementNS(*name* STRING, *ns* STRING)	Writes a namespace-qualified XML start element to the StaxWriter stream.

xml.StaxWriter.attribute

Writes an XML attribute to the StaxWriter stream.

Syntax

```
attribute(
   name STRING,
   value STRING )
```

1. *name* is the local name of the XML attribute. It cannot be NULL.
2. *value* is the value of the XML attribute. It cannot be NULL.

Usage

Attributes can only be written on the StaxWriter stream if it points to a `START_ELEMENT` or an `EMPTY_ELEMENT`, otherwise the operation fails with an exception; that is, this method can only be called after a `startElement`, `startElementNS`, `emptyElement`, `emptyElementNS`, or `attribute` and `attributeNS`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.attributeNS

Writes an XML namespace qualified attribute to the StaxWriter stream.

Syntax

```
attributeNS(
   name STRING,
   ns STRING,
   value STRING )
```

1. *name* is the local name of the XML attribute, cannot be NULL.
2. *ns* is the namespace URI of the XML attribute, cannot be NULL.
3. *value* is the value of the XML attribute, cannot be NULL.

Usage

Attributes can only be written on the StaxWriter stream if it points to a `START_ELEMENT` or an `EMPTY_ELEMENT`, otherwise the operation fails with an exception; that is, this method can only be called after a `startElement`, `startElementNS`, `emptyElement`, `emptyElementNS`, or `attribute` and `attributeNS`.

If namespace URI has not been bound to a prefix with one of the methods `setPrefix`, `declareNamespace`, `setDefaultNamespace` or `declareDefaultNamespace`, the operation fails with an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.cdata

Writes an XML CData to the StaxWriter stream.

Syntax

```
cdata(
   data STRING )
```

1. *data* is the data contained in the CData section, or NULL.

Usage

This method writes XML character data passed as parameter as a CData.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.characters

Writes an XML text to the StaxWriter stream.

Syntax

```
characters(
   text STRING )
```

1. *text* is the value to write.

Usage

This method writes the character string passed as parameter as a text element.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.close

Closes the StaxWriter streaming, and releases all associated resources.

Syntax

```
close()
```

Usage

This method closes the stream.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.comment

Writes an XML comment to the StaxWriter stream.

Syntax

```
comment(
   data STRING )
```

1. *data* is the data in the XML comment, or NULL.

Usage

This method write and XML comment to the stream.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.create

Constructor of a StaxWriter object.

Syntax

```
xml.StaxWriter.create()
  RETURNING object xml.StaxWriter
```

Usage

Returns a new StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.declareDefaultNamespace

Binds a namespace URI to the default namespace, and forces the output of the default XML namespace definition to the StaxWriter stream.

Syntax

```
declareDefaultNamespace(
   defaultNS STRING )
```

1. *defaultNS* is the URI to bind to the default namespace. It cannot be NULL.

Usage

The stream must point to a START_ELEMENT, and the prefix scope is the current START_ELEMENT / END_ELEMENT pair.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.declareNamespace

Binds a namespace URI to a prefix, and forces the output of the XML namespace definition to the StaxWriter stream.

Syntax

```
declareNamespace(
   prefix STRING,
   ns STRING )
```

1. *prefix* is the prefix to be bind to the URI, cannot be NULL.
2. *ns* is the URI to bind to the default namespace, cannot be NULL.

Usage

The stream must point to a `START_ELEMENT`, and the prefix scope is the current `START_ELEMENT` / `END_ELEMENT` pair.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.dtd

Writes a DTD to the StaxWriter stream.

Syntax

```
dtd(
   data STRING )
```

1. *data* is a string representing a valid DTD, cannot be NULL.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.emptyElement

Writes an empty XML element to the StaxWriter stream.

Syntax

```
emptyElement(
   name STRING )
```

1. *name* is the local name of the XML empty element, cannot be NULL.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.emptyElementNS

Writes an empty namespace qualified XML element to the StaxWriter stream.

Syntax

```
emptyElementNS(
```

```
   name STRING,
   ns STRING )
```

1. *name* is the local name of the XML empty element, cannot be NULL.
2. *ns* is the namespace URI of the XML empty element, cannot be NULL.

Usage

If namespace URI has not been bound to a prefix with one of the functions `setPrefix`, `declareNamespace`, `setDefaultNamespace` or `declareDefaultNamespace`, the operation fails with an exception.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.endDocument

Closes any open tags and writes corresponding end tags.

Syntax

```
endDocument()
```

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.endElement

Writes an end tag to the StaxWriter stream relying on the internal state to determine the prefix and local name of the last `START_ELEMENT`.

Syntax

```
endElement()
```

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.entityRef

Writes an XML EntityReference to the StaxWriter stream.

Syntax

```
entityRef(
   name STRING )
```

1. *name* is the name of the entity, cannot be NULL.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.getFeature

Gets a feature of a StaxWriter object.

Syntax

```
getFeature(
   feature STRING )
  RETURNING str STRING
```

1. *feature* is the name of a feature.

Usage

Returns the feature value. See StaxWriter Features on page 2182.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.processingInstruction

Writes an XML ProcessingInstruction to the StaxWriter stream

Syntax

```
processingInstruction(
   target STRING,
   data STRING )
```

1. *target* is the target of the Processing Instruction, cannot be NULL.
2. *data* is the data of the Processing Instruction, or NULL.

Usage

Writes an XML ProcessingInstruction to the StaxWriter stream, where *target* is the target of the Processing Instruction, cannot be NULL; *data* is the data of the Processing Instruction, or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.setDefaultNamespace

Binds a namespace URI to the default namespace.

Syntax

```
setDefaultNamespace(
   defaultNS STRING )
```

1. *defaultNS* is the URI to bind to the default namespace, cannot be NULL.

Usage

Binds a namespace URI to the default namespace. The default namespace scope is the current START_ELEMENT / END_ELEMENT pair; *defaultNS* is the URI to bind to the default namespace, cannot be NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.setFeature

Sets a feature of a StaxWriter object.

Syntax

```
setFeature(
   feature STRING,
   value STRING )
```

1. *feature* is the name of a feature.
2. *value* is the value of the feature.

Usage

Sets a feature of a StaxWriter object, where *feature* is the name of a feature, and *value* is the value of the feature. The features can be changed at any time, but will only be taken into account at the beginning of a new stream (see `writeTo` or `writeToDocument`).

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.setPrefix

Binds a namespace URI to a prefix.

Syntax

```
setPrefix(
   prefix STRING,
   ns STRING )
```

1. *prefix* is the prefix to be bind to the URI, cannot be NULL.
2. *ns* is the namespace URI to be bind to the prefix, cannot be NULL.

Usage

The prefix scope is the current `START_ELEMENT` / `END_ELEMENT` pair.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.startDocument

Writes an XML declaration to the StaxWriter stream.

Syntax

```
startDocument(
   encoding STRING,
   version STRING,
   standalone INTEGER )
```

1. *encoding* is the encoding of the XML declaration, or NULL to use the default UTF-8 encoding.
2. *version* is the XML version of the XML declaration, or NULL to use the default 1.0 version
3. *standalone* when TRUE sets the standalone of the XML declaration to "yes", when FALSE sets it to "no" or NULL.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Example

This call:

```
startDocument("utf-8","1.0",true)
```

Produces:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
dtd("note [<!ENTITY writer \"Donald Duck.\">]")
```

xml.StaxWriter.startElement

Writes an XML start element to the StaxWriter stream.

Syntax

```
startElement(
   name STRING )
```

1. *name* is the local name of the XML start element, cannot be NULL.

Usage

All startElement methods open a new scope and set the stream to a START_ELEMENT. Writing the corresponding endElement causes the scope to be closed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.startElementNS

Writes a namespace-qualified XML start element to the StaxWriter stream.

Syntax

```
startElementNS(
   name STRING,
   ns STRING )
```

1. *name* is the local name of the XML start element, cannot be NULL.
2. *ns* is the namespace URI of the XML start element, cannot be NULL.

Usage

All startElementNS methods open a new scope and set the stream to a START_ELEMENT. Writing the corresponding endElement causes the scope to be closed.

If namespace URI has not been bound to a prefix with one of the functions setPrefix, declareNamespace, setDefaultNamespace or declareDefaultNamespace, the operation fails with an exception.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.writeTo

Sets the output stream of the StaxWriter object to a file or an URL, and starts the streaming.

Syntax

```
writeTo(
   url STRING )
```

1. *url* is a valid URL or the name of the file that will contain the resulting XML document.

Usage

Only the following kinds of URLs are supported:

- http://
- https://
- tcp://
- tcps://
- file:///
- alias://

See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Examples

```
writeTo("printerList.xml")
```

```
writeTo("http://myserver:1100/documents/printerList.xml")
```

```
writeTo("https://myserver:1100/documents/printerList.xml")
```

```
writeTo("alias://printerlist")
```

where printerlist alias is defined in fglprofile as ws.printerlist.url = "http://myserver:1100/documents/ptinterList.xml".

xml.StaxWriter.writeToDocument

Sets the output stream of the StaxWriter object to an xml.DomDocument object, and starts the streaming.

Syntax

```
writeToDocument(
   doc xml.DomDocument )
```

1. *doc* is the empty xml.DomDocument object that will contain the resulting XML document.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.writeToPipe

Sets the output stream of the StaxWriter object to a PIPE, and starts the streaming.

Syntax

```
writeToPipe(
   cmd STRING )
```

1. *cmd* is the command to start the PIPE that will get the resulting XML document.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxWriter.writeToText

Sets the output stream of the StaxWriter object to a TEXT large object, and starts the streaming.

Syntax

```
writeToText(
   txt TEXT )
```

1. *txt* must be a TEXT lob located in memory that will contain the resulting XML document.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

StaxWriter Features

Features of the `xml.StaxWriter` class.

Table 499: StaxWriter features

Feature	Description
format-pretty-print	Formats the output by adding whitespace to produce a pretty-printed, indented, human-readable form. Default value is FALSE.
smart-ending-elements	Outputs each tag closed with an `endElement()` call as empty elements if they have no children. Default value is FALSE.

Example
This example uses methods from the `xml.StaxWriter` class.

```
IMPORT xml

FUNCTION save(file)
  DEFINE file STRING
  DEFINE writer xml.StaxWriter
  TRY
    LET writer = xml.StaxWriter.Create()
    CALL writer.setFeature("format-pretty-print",TRUE)
    CALL writer.writeTo(file)
    CALL writer.startDocument("utf-8","1.0",true)
    CALL writer.comment("This is my first comment using a stax writer")
    CALL writer.setPrefix("c","http://www.mycompany.com/c")
    CALL writer.setPrefix("d","http://www.mycompany.com/d")
    CALL writer.setDefaultNamespace("http://www.mycompany.com/d")
    CALL writer.startElementNS("root", "http://www.mycompany.com/d")
    CALL writer.attribute("attr1","value1")
    CALL writer.attribute("attr2","value2")
    CALL writer.attributeNS("attr3", "http://www.mycompany.com/d","value3")
    CALL writer.comment("This is a comment using a stax writer")
    CALL writer.startElementNS("eltA", "http://www.mycompany.com/d")
    CALL writer.CData("<this is a CData section>")
    CALL writer.endElement()
    CALL writer.startElementNS("eltB", "http://www.mycompany.com/c")
    CALL writer.characters("Hello world, I'm from the development team")
    CALL writer.entityRef("one")
    CALL writer.endElement()
    CALL writer.processingInstruction("command1","do what you want")
    CALL writer.endElement()
    CALL writer.comment("This is my last comment using a stax writer")
    CALL writer.endDocument()
    CALL writer.close()
    RETURN TRUE
  CATCH
    DISPLAY "StaxWriter ERROR :",STATUS, SQLCA.SQLERRM
    RETURN FALSE
  END TRY
END FUNCTION
```

The StaxReader class
The StaxReader class provides methods compatible with StAX (Streaming API for XML) for reading XML documents.

The `STATUS` variable is set to zero after a successful method call.

Syntax

```
xml.StaxReader
```

xml.StaxReader methods
Methods for the `xml.StaxReader` class.

Table 500: Class methods: Creation

Name	Description
`xml.StaxReader.Create()` `  RETURNING` *`object`* `xml.StaxReader`	Constructor of a StaxReader object.

Table 501: Object methods: Configuration

Name	Description
`setFeature( feature STRING, value STRING )`	Sets a feature of a StaxReader object.
`getFeature( feature STRING ) RETURNING str STRING`	Gets a feature of a StaxReader object.

Table 502: Object methods: Input

Name	Description
`readFrom( url STRING )`	Sets the input stream of the StaxReader object to a file or an URL and starts the streaming
`readFromDocument( doc xml.DomDocument )`	Sets the input stream of the StaxReader object to a DomDocument object and starts the streaming
`readFromText( txt TEXT )`	Sets the input stream of the StaxReader object to a TEXT large object and starts the streaming.
`readFromPipe( cmd STRING )`	Sets the input stream of the StaxReader object to a PIPE and starts the streaming.
`close()`	Closes the StaxReader streaming and releases all associated resources.

Table 503: Object methods: Access

Name	Description
`getEventType() RETURNING eventtype STRING`	Returns a string that indicates the type of event the cursor of the StaxReader object is pointing to.
`hasName() RETURNING flag INTEGER`	Checks whether the StaxReader cursor points to a node with a name.
`hasText() RETURNING flag INTEGER`	Checks whether the StaxReader cursor points to a node with a text value.
`isEmptyElement()`	Checks whether the StaxReader cursor points to an empty element node.

Name	Description
RETURNING *flag* INTEGER	
isStartElement() RETURNING *flag* INTEGER	Checks whether the StaxReader cursor points to a start element node.
isEndElement() RETURNING *flag* INTEGER	Checks whether the StaxReader cursor points to an end element node.
isCharacters() RETURNING *flag* INTEGER	Checks whether the StaxReader cursor points to a text node.
isIgnorableWhitespace() RETURNING *flag* INTEGER	Checks whether the StaxReader cursor points to ignorable whitespace.

Table 504: Object methods: Document

Name	Description
getEncoding() RETURNING *docenc* STRING	Returns the document encoding defined in the XML Document declaration, or NULL.
getVersion() RETURNING *version* STRING	Returns the document version defined in the XML Document declaration, or NULL.
isStandalone() RETURNING *flag* STRING	Checks whether the document standalone attribute defined in the XML Document declaration is set to yes.
standaloneSet() RETURNING *flag* STRING	Checks whether the document standalone attribute is defined in the XML Document declaration.

Table 505: Object methods: Nodes

Name	Description
getPrefix() RETURNING *prefix* STRING	Returns the prefix of the current XML node, or NULL.
getLocalName() RETURNING *localname* STRING	Returns the local name of the current XML node, or NULL.
getName()	Returns the qualified name of the current XML node, or NULL.

Name	Description
`RETURNING name STRING`	
`getNamespace()` `RETURNING nsuri STRING`	Returns the namespace URI of the current XML node, or NULL.
`getText()` `RETURNING value STRING`	Returns as a string the value of the current XML node, or NULL.

Table 506: Object methods: Processing Instructions

Name	Description
`getPITarget()` `RETURNING target STRING`	Returns the target part of an XML ProcessingInstruction node, or NULL.
`getPIData()` `RETURNING data STRING`	Returns the data part of an XML ProcessingInstruction node, or NULL.

Table 507: Object methods: Attributes

Name	Description
`getAttributeCount()` `RETURNING num INTEGER`	Returns the number of XML attributes defined on the current XML node, or zero.

`getAttributeLocalName(`
`pos INTEGER )`
`RETURNING localname STRING`
	Returns the local name of an XML attribute defined at a given position on the current XML node, or NULL.

`getAttributeNamespace(`
`pos INTEGER )`
`RETURNING nsuri STRING`
	Returns the namespace URI of an XML attribute defined at a given position on the current XML node, or NULL.

`getAttributePrefix(`
`pos INTEGER )`
`RETURNING prefix STRING`
	Returns the prefix of an XML attribute defined at a given position on the current XML node, or NULL.

`getAttributeValue(`
`pos INTEGER )`
`RETURNING value STRING`
	Returns the value of an XML attribute defined at a given position on the current XML node, or NULL.

`findAttributeValue(`
`name STRING,`
`ns STRING )`
`RETURNING value STRING`
	Returns the value of an XML attribute of a given name and/or namespace on the current XML node, or NULL.

Table 508: Object methods: Namespace

Name	Description
`lookupNamespace( prefix STRING ) RETURNING nsuri STRING`	Looks up the namespace URI associated with a given prefix starting from the current XML node the StaxReader cursor is pointing to.
`lookupPrefix( ns STRING ) RETURNING prefix STRING`	Looks up the prefix associated with a given namespace URI, starting from the current XML node the StaxReader cursor is pointing to.
`getNamespaceCount() RETURNING num INTEGER`	Returns the number of namespace declarations defined on the current XML node, or zero.
`getNamespacePrefix( pos INTEGER ) RETURNING prefix STRING`	Returns the prefix of a namespace declaration defined at a given position on the current XML node, or NULL.
`getNamespaceURI( pos INTEGER ) RETURNING nsuri STRING`	Returns the URI of a namespace declaration defined at a given position on the current XML node, or NULL.

Table 509: Object methods: Navigation

Name	Description
`hasNext() RETURNING flag INTEGER`	Checks whether the StaxReader cursor can be moved to a XML node next to it.
`next()`	Moves the StaxReader cursor to the next XML node.
`nextTag()`	Moves the StaxReader cursor to the next XML open or end tag
`nextSibling()`	Moves the StaxReader cursor to the immediate next sibling XML Element of the current node, skipping all its child nodes.

xml.StaxReader.close

Closes the StaxReader streaming and releases all associated resources.

Syntax

```
close()
```

Usage

This method closes the stream.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.Create

Constructor of a StaxReader object.

Syntax

```
xml.StaxReader.Create()
  RETURNING object xml.StaxReader
```

Usage

Returns a StaxReader object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.findAttributeValue

Returns the value of an XML attribute of a given name and/or namespace on the current XML node, or NULL.

Syntax

```
findAttributeValue(
   name STRING,
   ns STRING )
  RETURNING value STRING
```

1. *name* is the name of the attribute to retrieve. It cannot be NULL.
2. *ns* is the namespace URI of the attribute to retrieve, or NULL if the attribute is not namespace-qualified.

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getAttributeCount

Returns the number of XML attributes defined on the current XML node, or zero.

Syntax

```
getAttributeCount()
  RETURNING num INTEGER
```

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getAttributeLocalName

Returns the local name of an XML attribute defined at a given position on the current XML node, or NULL.

Syntax

```
getAttributeLocalName(
   pos INTEGER )
  RETURNING localname STRING
```

1. *pos* is the position of the attribute to return (Index starts at 1).

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getAttributeNamespace

Returns the namespace URI of an XML attribute defined at a given position on the current XML node, or NULL.

Syntax

```
getAttributeNamespace(
   pos INTEGER )
  RETURNING nsuri STRING
```

1. *pos* is the position of the attribute to return (Index starts at 1).

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getAttributePrefix

Returns the prefix of an XML attribute defined at a given position on the current XML node, or NULL.

Syntax

```
getAttributePrefix(
   pos INTEGER )
  RETURNING prefix STRING
```

1. *pos* is the position of the attribute to return (Index starts at 1).

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getAttributeValue

Returns the value of an XML attribute defined at a given position on the current XML node, or NULL.

Syntax

```
getAttributeValue(
   pos INTEGER )
  RETURNING value STRING
```

1. *pos* is the position of the attribute to return (Index starts at 1).

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getEncoding

Returns the document encoding defined in the XML Document declaration, or NULL.

Syntax

```
getEncoding()
  RETURNING docenc STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getEventType

Returns a string that indicates the type of event the cursor of the StaxReader object is pointing to.

Syntax

```
getEventType()
  RETURNING eventtype STRING
```

Usage

See StaxReader Event Types on page 2200 for the full list of StaxReader event types.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getFeature

Gets a feature of a StaxReader object.

Syntax

```
getFeature(
   feature STRING )
  RETURNING str STRING
```

1. *feature* is the name of a feature.

Usage

See StaxReader Features on page 2200 for the full list of StaxReader features.

Returns the feature value.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getLocalName

Returns the local name of the current XML node, or NULL.

Syntax

```
getLocalName()
  RETURNING localname STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getName

Returns the qualified name of the current XML node, or NULL.

Syntax

```
getName()
  RETURNING name STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getNamespace

Returns the namespace URI of the current XML node, or NULL.

Syntax

```
getNamespace()
  RETURNING nsuri STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getNamespaceCount

Returns the number of namespace declarations defined on the current XML node, or zero.

Syntax

```
getNamespaceCount()
```

```
  RETURNING num INTEGER
```

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getNamespacePrefix

Returns the prefix of a namespace declaration defined at a given position on the current XML node, or NULL.

Syntax

```
getNamespacePrefix(
   pos INTEGER )
  RETURNING prefix STRING
```

1. *pos* is the position of the namespace declaration (Index starts at 1).

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getNamespaceURI

Returns the URI of a namespace declaration defined at a given position on the current XML node, or NULL.

Syntax

```
getNamespaceURI(
   pos INTEGER )
  RETURNING nsuri STRING
```

1. *pos* is the position of the namespace declaration (Index starts at 1).

Usage

This method is only valid on a START_ELEMENT node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getPIData

Returns the data part of an XML ProcessingInstruction node, or NULL.

Syntax

```
getPIData()
  RETURNING data STRING
```

Usage

This method is only valid on a `PROCESSING_INSTRUCTION` node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getPITarget

Returns the target part of an XML ProcessingInstruction node, or NULL.

Syntax

```
getPITarget()
  RETURNING target STRING
```

Usage

This method is only valid on a `PROCESSING_INSTRUCTION` node.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getPrefix

Returns the prefix of the current XML node, or NULL.

Syntax

```
getPrefix()
  RETURNING prefix STRING
```

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getText

Returns as a string the value of the current XML node, or NULL.

Syntax

```
getText()
  RETURNING value STRING
```

Usage

This method is only valid on `CHARACTERS`, `CDATA`, `SPACE`, `COMMENT`, `DTD` and `ENTITY_REFERENCE` nodes. For an `ENTITY_REFERENCE`, this method returns the replacement value, or NULL if none.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.getVersion

Returns the document version defined in the XML Document declaration, or NULL.

Syntax

```
getVersion()
  RETURNING version STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.hasName

Checks whether the StaxReader cursor points to a node with a name.

Syntax

```
hasName()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML node has a name, FALSE otherwise. This method returns TRUE for START_ELEMENT and END_ELEMENT, FALSE for all other nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.hasNext

Checks whether the StaxReader cursor can be moved to a XML node next to it.

Syntax

```
hasNext()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if there is still an XML node in the stream, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.hasText

Checks whether the StaxReader cursor points to a node with a text value.

Syntax

```
hasText()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML node has a text value, FALSE otherwise. This method returns TRUE for `CHARACTERS`, `SPACE`, `CDATA`, `COMMENT`, `ENTITY_REFERENCE` and `DTD`, FALSE for all other nodes.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.isCharacters

Checks whether the StaxReader cursor points to a text node.

Syntax

```
isCharacters()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML node is a text node, FALSE otherwise.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.isEmptyElement

Checks whether the StaxReader cursor points to an empty element node.

Syntax

```
isEmptyElement()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML element node has no children, FALSE otherwise.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.isEndElement

Checks whether the StaxReader cursor points to an end element node.

Syntax

```
isEndElement()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML node is an end element node, FALSE otherwise.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.isIgnorableWhitespace

Checks whether the StaxReader cursor points to ignorable whitespace.

Syntax

```
isIgnorableWhitespace()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML node is an ignorable text node, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.isStandalone

Checks whether the document standalone attribute defined in the XML Document declaration is set to yes.

Syntax

```
isStandalone()
  RETURNING flag STRING
```

Usage

Returns TRUE if the standalone attribute in the XML declaration is set to yes, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.isStartElement

Checks whether the StaxReader cursor points to a start element node.

Syntax

```
isStartElement()
  RETURNING flag INTEGER
```

Usage

Returns TRUE if the current XML node is a start element node, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.lookupNamespace

Looks up the namespace URI associated with a given prefix starting from the current XML node the StaxReader cursor is pointing to.

Syntax

```
lookupNamespace(
   prefix STRING )
  RETURNING nsuri STRING
```

1. *prefix* is the prefix to look for; if NULL the default namespace URI will be returned..

Usage

Returns the namespace URI associated with the prefix, or NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.lookupPrefix

Looks up the prefix associated with a given namespace URI, starting from the current XML node the StaxReader cursor is pointing to.

Syntax

```
lookupPrefix(
   ns STRING )
  RETURNING prefix STRING
```

1. *ns* is the namespace URI to look for. It cannot be NULL.

Usage

Returns the prefix associated with this namespace URI, or NULL if there is none.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.next

Moves the StaxReader cursor to the next XML node.

Syntax

```
next()
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.nextSibling

Moves the StaxReader cursor to the immediate next sibling XML Element of the current node, skipping all its child nodes.

Syntax

```
nextSibling()
```

Usage

The cursor points to the parent end tag if there are no siblings.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.nextTag

Moves the StaxReader cursor to the next XML open or end tag

Syntax

```
nextTag()
```

Usage

The cursor points to the end of the document if there is no next XML open or end tag.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.readFrom

Sets the input stream of the StaxReader object to a file or an URL and starts the streaming

Syntax

```
readFrom(
   url STRING )
```

1. *url* is a valid URL or the name of the file to read.

Usage

Only the following kinds of URLs are supported:

- http://
- https://
- tcp://
- tcps://
- file:///
- alias://

See FGLPROFILE Configuration for more details about URL mapping with aliases, and for proxy and security configuration.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.readFromDocument

Sets the input stream of the StaxReader object to a DomDocument object and starts the streaming

Syntax

```
readFromDocument(
   doc xml.DomDocument )
```

1. *doc* is an XML/DomDocument object that contains an XML document.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.readFromPipe

Sets the input stream of the StaxReader object to a PIPE and starts the streaming.

Syntax

```
readFromPipe(
   cmd STRING )
```

1. *cmd* is the command to start the PIPE and where the reader will get the XML from.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.readFromText

Sets the input stream of the StaxReader object to a TEXT large object and starts the streaming.

Syntax

```
readFromText(
   txt TEXT )
```

1. *txt* must be a TEXT lob located in memory and containing the XML to read.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.setFeature

Sets a feature of a StaxReader object.

Syntax

```
setFeature(
   feature STRING,
   value STRING )
```

1. *feature* is the name of a feature.
2. *value* is the value of the feature.

Usage

See StaxReader Features on page 2200 for the full list of StaxReader features.

The features can be changed at any time, but will only be taken into account at the beginning of a new stream (see `readFrom` or `readFromDocument`).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.StaxReader.standaloneSet

Checks whether the document standalone attribute is defined in the XML Document declaration.

Syntax

```
standaloneSet()
  RETURNING flag STRING
```

Usage

Returns TRUE if the standalone attribute in the XML declaration is set, FALSE otherwise.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

StaxReader Features
Features of the xml.StaxReader class.

Table 510: StaxReader Features

Feature	Description
expand-entity-references	Defines whether XML EntityReference nodes are kept or replaced during the parsing of an XML document. Default value is TRUE.

StaxReader Event Types
Event types of the xml.StaxReader class.

Table 511: StaxReader event types

Type	Description	XML sample
START_DOCUMENT	StaxReader cursor points to the beginning of the XML document.	<?xml version="1.0" standalone="no"?>
END_DOCUMENT	StaxReader cursor has reached the end of the XML document. No additional parsing operation will succeed.	
START_ELEMENT	StaxReader cursor points to an XML start element or empty element node.	<p:elt attr="val"> or <p:elt attr="val"/>
END_ELEMENT	StaxReader cursor points to an XML end element node.	</p:elt>
CHARACTERS	StaxReader cursor points to an XML text node.	... eltA/>This is text<eltB ...

Type	Description	XML sample
CDATA	StaxReader cursor points to an XML CData node.	`<![CDATA[<Hello, world!>]]>`
SPACE	StaxReader cursor points to an XML text node containing only whitespaces.	`... eltA/> <eltB ...`
COMMENT	StaxReader cursor points to an XML comment node.	`<!-- a comment -->`
DTD	StaxReader cursor points to a DTD string.	`<!DOCTYPE A [ <!ELEMENT B (C +)> ]>`
ENTITY_REFERENCE	StaxReader cursor points to an XML entity reference node.	`&ref;`
PROCESSING_INSTRUCTION	StaxReader cursor points to an XML processing instruction node.	`<?target data?>`
ERROR	StaxReader cursor points to an unexpected XML node.	

Example
Example using methods of the `xml.StaxReader` class.

```
IMPORT  xml

FUNCTION  parse(file  )
  DEFINE  file     STRING
  DEFINE  event    STRING
  DEFINE  ret      INTEGER
  DEFINE  ind      INTEGER
  DEFINE  reader   xml.StaxReader
  TRY
    LET  reader=xml.StaxReader.Create()
    CALL  reader.readFrom(file)
    WHILE  (true)
      LET  event=reader.getEventType()
      CASE  event
        WHEN  "START_DOCUMENT"
          DISPLAY  "Document reading started"
          DISPLAY  "XML Version  : ",reader.getVersion()
          DISPLAY  "XML Encoding : ",reader.getEncoding()
          IF  reader.standaloneSet() THEN
            IF  reader.isStandalone() THEN
              DISPLAY  "Standalone   : yes"
            ELSE
              DISPLAY  "Standalone   : no"
            END IF
          END IF
        WHEN  "END_DOCUMENT"
          DISPLAY  "Document reading finished"
        WHEN  "START_ELEMENT"
```

```
          IF  reader.isEmptyElement() THEN
            DISPLAY  "<"||reader.getName()||"/>"
          ELSE
            DISPLAY  "<"||reader.getName()||">"
          END IF
          FOR  ind=1 TO  reader.getNamespaceCount()
            DISPLAY  "xmlns:"||reader.getNamespacePrefix(ind)||"="
                        ||reader.getNamespaceURI(ind)
          END FOR
          FOR  ind=1 TO reader.getAttributeCount()
            IF  reader.getAttributePrefix(ind) THEN
              DISPLAY  reader.getAttributePrefix(ind)||":"
                          ||reader.getAttributeLocalName(ind)||"="
                          ||reader.getAttributeValue(ind)
            ELSE
              DISPLAY  reader.getAttributeLocalName(ind)||"="
                          ||reader.getAttributeValue(ind)
            END IF
          END FOR
        WHEN  "END_ELEMENT"
          DISPLAY  "</"||reader.getName()||">"
        WHEN  "CHARACTERS"
          IF  reader.hasText() AND NOT reader.isIgnorableWhitespace() THEN
            DISPLAY  "CHARACTERS :",reader.getText()
          END IF
        WHEN  "COMMENT"
          IF  reader.hasText() THEN
            DISPLAY  "Comment :",reader.getText()
          END IF
        WHEN  "CDATA"
          IF  reader.hasText() THEN
            DISPLAY  "CDATA :", reader.getText()
          END IF
        WHEN  "PROCESSING_INSTRUCTION"
          DISPLAY  "PI :",reader.getPITarget(),reader.getPIData()
        WHEN  "ENTITY_REFERENCE"
          DISPLAY  "Entity name :",reader.getName()
        OTHERWISE
          DISPLAY  "Unknown "||event||" node"
      END CASE
      IF  reader.hasNext() THEN
        CALL  reader.next()
      ELSE
        CALL  reader.close()
        EXIT WHILE
      END IF
    END WHILE
  CATCH
    DISPLAY  "StaxReader ERROR :",STATUS||" ("||SQLCA.SQLERRM||")"
  END TRY
END FUNCTION
```

XML serialization classes

The XML serialization classes convert BDL variables to XML and XML to BDL variables.

- CLASS Serializer
  - Option flags

The Serializer class

The `xml.Serializer` class provides methods to manage options for the serializer engine, and to use the serializer engine to serialize variables and XML element nodes.

This class is a static class and does not have to be instantiated.

The STATUS variable is set to zero after a successful method call.

xml.Serializer methods
Methods for the `xml.Serializer` class.

Table 512: Class methods

Name	Description
`xml.Serializer.CreateXmlSchemas( var fgl-type, ar DYNAMIC ARRAY OF xml.DomDocument )`	Creates XML schemas corresponding to the given variable *var*, and fills the dynamic array *ar* with xml.DomDocument objects each representing an XML schema.
`xml.Serializer.DomToStax( node xml.DomNode, stax xml.StaxWriter )`	Serializes an XML node object to a StaxWriter object.
`xml.Serializer.DomToVariable( node xml.DomNode, var fgl-type )`	Serializes an XML element node into a BDL variable using a DomNode object.
`xml.Serializer.GetOption( flag STRING ) RETURNING value STRING`	Gets a global option value from the serializer engine.
`xml.Serializer.SetOption( flag STRING, value STRING )`	Sets a global option value for the serializer engine
`xml.Serializer.SoapSection5ToVariable( node xml.DomNode, var fgl-type )`	Serializes an XML element node into a BDL variable in Soap Section 5 encoding.
`xml.Serializer.StaxToVariable( stax xml.StaxReader, var fgl-type )`	Serializes an XML element node into a BDL variable using a StaxReader object.
`xml.Serializer.StaxToDom( stax xml.StaxReader, node xml.DomNode )`	Serializes an XML element node into a DomNode object using a StaxReader object.
`xml.Serializer.VariableToDom( var fgl-type, node xml.DomNode )`	Serializes a BDL variable into an XML element node using a DomNode object.
`xml.Serializer.VariableToSoapSection5( var fgl-type,`	Serializes a BDL variable into an XML element node in Soap Section 5 encoding.

Name	Description
node xml.DomNode)	
xml.Serializer.VariableToStax(*var fgl-type*, *stax* xml.StaxWriter)	Serializes a BDL variable into an XML element node using a StaxWriter object.

xml.Serializer.CreateXmlSchemas

Creates XML schemas corresponding to the given variable *var*, and fills the dynamic array *ar* with xml.DomDocument objects each representing an XML schema.

Syntax

```
xml.Serializer.CreateXmlSchemas(
   var fgl-type,
   ar DYNAMIC ARRAY OF xml.DomDocument )
```

1. *var* is a given variable.
2. *ar* is a dynamic array of xml.DomDocument objects, each representing an XML schema.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.DomToStax

Serializes an XML node object to a StaxWriter object.

Syntax

```
xml.Serializer.DomToStax(
   node xml.DomNode,
   stax xml.StaxWriter )
```

1. *node* is an XML DomNode object.
2. *stax* is a StaxWriter object.

Usage

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.DomToVariable

Serializes an XML element node into a BDL variable using a DomNode object.

Syntax

```
xml.Serializer.DomToVariable(
   node xml.DomNode,
   var fgl-type )
```

1. *node* is a DomNode object of type `ELEMENT_NODE`.
2. *var* is any BDL variable with optional XML mapping attributes.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.getOption

Gets a global option value from the serializer engine.

Syntax

```
xml.Serializer.GetOption(
   flag STRING )
  RETURNING value STRING
```

1. *flag* is the option flag.

Usage

Returns the value of the flag.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.setOption

Sets a global option value for the serializer engine

Syntax

```
xml.Serializer.SetOption(
   flag STRING,
   value STRING )
```

1. *flag* is the option flag.
2. *value* is the value of the flag.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.SoapSection5ToVariable

Serializes an XML element node into a BDL variable in Soap Section 5 encoding.

Syntax

```
xml.Serializer.SoapSection5ToVariable(
   node xml.DomNode,
   var fgl-type )
```

1. *node* is a DomNode object of type `ELEMENT_NODE`.
2. *var* is any BDL variable with optional XML mapping attributes.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.StaxToDom

Serializes an XML element node into a DomNode object using a StaxReader object.

Syntax

```
xml.Serializer.StaxToDom(
   stax xml.StaxReader,
   node xml.DomNode )
```

1. *stax* is a StaxReader object where the cursor points to an XML Element node,
2. *node* is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.

Usage

The resulting XML element node of the serialization process will be appended to the last child of the given node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.StaxToVariable

Serializes an XML element node into a BDL variable using a StaxReader object.

Syntax

```
xml.Serializer.StaxToVariable(
   stax xml.StaxReader,
   var fgl-type )
```

1. *stax* is a StaxReader object where the cursor points to an XML Element node.
2. *var* is any BDL variable with optional XML mapping attributes.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.VariableToDom

Serializes a BDL variable into an XML element node using a DomNode object.

Syntax

```
xml.Serializer.VariableToDom(
   var fgl-type,
   node xml.DomNode )
```

1. *var* is any BDL variable with optional XML mapping attributes.
2. *node* is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.

Usage

The resulting XML element node of the serialization process will be appended to the last child of the given node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.VariableToSoapSection5

Serializes a BDL variable into an XML element node in Soap Section 5 encoding.

Syntax

```
xml.Serializer.VariableToSoapSection5(
   var fgl-type,
   node xml.DomNode )
```

1. *var* is any BDL variable with optional XML mapping attributes.
2. *node* is a DomNode object of type ELEMENT_NODE or DOCUMENT_FRAGMENT_NODE.

Usage

The resulting XML element node of the serialization process will be appended to the last child of the given node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Serializer.VariableToStax

Serializes a BDL variable into an XML element node using a StaxWriter object.

Syntax

```
xml.Serializer.VariableToStax(
   var fgl-type,
   stax xml.StaxWriter )
```

1. *var* is any BDL variable with optional XML mapping attributes.
2. *stax* is a StaxWriter object.

Usage

The resulting XML element node of the serialization process will be added at the current cursor position of the StaxWriter object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Serialization option flags

Serialization option flags for the xml.Serializer class.

Table 513: Serialization option flags

Flag	Description
xml_ignoretimezone	Defines whether, during the marshalling and un-marshalling process of a BDL DATETIME data

Flag	Description
	type, the Serializer should ignore the time zone information. A value of zero means FALSE. The default is FALSE. Throws an exception in case of errors, and updates status with an error code.
xml_usetypedefinition	Defines whether the Serializer must specify the type of data during serialization. This will add an "xsi:type" attribute to each XML data type. A value of zero means FALSE. The default is FALSE. Throws an exception in case of errors, and updates status with an error code.
xml_useutctime	Defines whether, during the marshalling process of a BDL DATETIME data type, the Serializer should convert it into UTC time. A value of zero means FALSE. The default is FALSE. Throws an exception in case of errors, and updates status with an error code.
xs_processcontents	Defines the way to generate wildcard elements and attributes in XML schemas via the XML schema **processContents** tag. See Table 514: Values for xs_processcontents on page 2208 Throws an exception in case of errors, and updates status with an error code.

Table 514: Values for xs_processcontents

Value	Description
0	No processContents tag will be generated. **(default)**
1	Generation of **processContents="skip"**.
2	Generation of **processContents="lax"**.
3	Generation of **processContents="strict"**.

XML security classes

XML Security classes handle encryption and signature of XML documents entirely in memory with keys and certificates.

Important: The XML security classes are not supported on GMI mobile devices.

- CLASS CryptoKey
  - Keys
- CLASS CryptX509

- CLASS Encryption
- CLASS Signature
  - Digests
  - Transformations
- CLASS KeyStore

The CryptoKey class
The `xml.CryptoKey` class provides methods to manipulate HMAC, symmetric and asymmetric keys needed for signing, verifying, encrypting and decrypting XML documents or document fragments.

It follows the XML-Signature and XML-Encryption specifications.

The `STATUS` variable is set to zero after a successful method call.

> **Important:** This class is not supported on GMI mobile devices.

xml.CryptoKey methods
Methods for the `xml.CryptoKey` class.

Table 515: Class methods: Creation

Name	Description

`xml.CryptoKey.Create(`
`  url STRING )`
`RETURNING object xml.CryptoKey`
	Initializes an xml.CryptoKey object. Constructor of an empty CryptoKey object depending on a url.

`xml.CryptoKey.CreateDerivedKey(`
`  url STRING )`
`RETURNING object xml.CryptoKey`
	Constructor of an empty CryptoKey object intended to be derived before use, and depending on a url.

`xml.CryptoKey.CreateFromNode(`
`  url STRING,`
`  node xml.DomNode )`
`RETURNING object xml.CryptoKey`
	Constructor of a new CryptoKey object depending on a url and from a XML node, according to the XML-Signature and XML-Encryption specification.

Table 516: Object methods: Access

Name	Description

`compareTo(`
`  secondKey xml.CryptoKey )`
`RETURNING flag INTEGER`
	Compares a CryptoKey object to a second key.

`getSHA1()` `RETURNING keyId STRING`	Returns the SHA1 encoded key identifier in a base64 encoded STRING.
`getSize()` `RETURNING size INTEGER`	Returns the size of the key in bits.
`getType()`	Returns the type of key.

Name	Description
RETURNING *type* STRING	
getUsage() RETURNING *usage* STRING	Returns the usage of the key.
getUrl() RETURNING *keyId* STRING	Returns the key identifier as an URL, as defined in the XML-Signature and XML-Encryption specification.

See also The Diffie-Hellman key agreement algorithm on page 2447.

Table 517: Object methods: Modify

Name	Description
deriveKey(*method* STRING, *label* STRING, *seed* STRING, *created* STRING, *offset* INTEGER, *size* INTEGER)	Derives the symmetric or HMAC CryptoKey object using the given *method* identifier and concatenating the optional *label*, the mandatory *seed* value and the optional *created* date as initial random value.
generateKey(*size* INTEGER)	Generates a random key of given size (in bits).
setKey(*key* STRING)	Defines the value of a HMAC or Symmetric key.

Table 518: Object methods: Load, save, and compute

Name	Description
computeKey(*otherPubKey* xml.CryptoKey, *url* STRING) RETURNING *sharedSecret* xml.CryptoKey	Computes the shared secret based on the given modulus, generator, the private key and the other peer's public key. The returned key can be any of symmetric/HMAC or symmetric/encryption key type. It can be used for symmetric signature or symmetric encryption.
loadBIN(*file* STRING)	Loads a symmetric or HMAC key from a file in raw format.
loadDER(*file* STRING)	Loads an asymmetric DSA key, an asymmetric RSA key or Diffie-Hellman parameters from a file in DER format.
loadFromString(	Loads the given key in BASE64 string format into a CryptoKey object.

Name	Description
str STRING)	
loadPEM(*file* STRING)	Loads an asymmetric DSA key, an asymmetric RSA key or Diffie-Hellman parameters from a file in PEM format.
loadPrivate(*xml* xml.DomDocument)	Loads the private asymmetric RSA key in the given XML document into the private part of this CryptoKey object, according to the XKMS2.0 specification.
loadPublic(*xml* xml.DomDocument)	Loads the public asymmetric RSA or DSA key in the given XML document into the public part of this CryptoKey object, according to the XML-Signature specification for DSA and RSA key value.
loadPublicFromString(*pubKeyStr* STRING)	Populate the current CryptoKey object with the passed public key.
savePrivate() RETURNING *object* xml.DomDocument	Saves the private part of an asymmetric RSA CryptoKey object into a XML document according to the XKMS2.0 specification.
savePublic() RETURNING *object* xml.DomDocument	Saves the public part of an asymmetric RSA or DSA CryptoKey object or the parameters and the public key of the Diffie-Hellman object into a XML document according to the XML-Signature specification for DSA and RSA and Diffie-Hellman key values.
savePublicToString() RETURNING *str* STRING	Save the current `xml.CryptoKey`'s public part in the returned base64 string.
saveToString() RETURNING *str* STRING	Saves the CryptoKey object into a BASE64 string format.

Table 519: Object methods: Feature

Name	Description
getFeature(*feature* STRING) RETURNING *value* STRING	Returns the value of the given feature for this CryptoKey object, or NULL.
setFeature(*feature* STRING, *value* STRING)	Sets or resets the value of a feature for a CryptoKey object.

Compares a CryptoKey object to a second key.

Syntax

```
compareTo(
   secondKey xml.CryptoKey )
  RETURNING flag INTEGER
```

1. *secondKey* is the xml.CryptoKey object to use for comparison to the current CryptoKey object.

Usage

The method verifies if the keys URL, type, size, usage and value are the same. If they are the same, the two identical keys will produce the same encryption cipher.

The key features are not taken into account during comparison.

Returns TRUE if they are identical, FALSE if they are not identical.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.computeKey

Computes the shared secret based on the given modulus, generator, the private key and the other peer's public key. The returned key can be any of symmetric/HMAC or symmetric/encryption key type. It can be used for symmetric signature or symmetric encryption.

Syntax

```
computeKey(
   otherPubKey xml.CryptoKey,
   url STRING )
  RETURNING sharedSecret xml.CryptoKey
```

1. *otherPubKey* is the other peer's public key (xml.CryptoKey).
2. *url* is the shared secret key type as an url identifier (STRING)..

Usage

> **Important:** This method is for Diffie-Hellman key-agreement algorithm only.

Returns an xml.CryptoKey sharedSecret: An xml.CryptoKey object of the specified type.

In the 3DES case, no key weakness test is done. If the compound shared secret is week, the other peer inovled in the communication may raise an error. It depends on the language used on the other side.

In order to be able to compute an AES256 shared secret of the Java™ side, you need to add or relace the files local_policy.jar and US_export_policy.jar located in $JDK_HOME/jre/lib/security by the Java™ Cryptographic Extension correspoinding to your JDK version. You can find this extension at http://www.oracle.com/technetwork/java/javase/downloads/index.html.

If the shared secret key length is less than the Diffie-Hellman key length, only the first needed bytes will be taken. For example, if the Diffie-Hellman is 512 bits length and the shared secret is a 3DES key, then only the first 192 bits will be used by the computation. In a 3DES shared secret case, xml.CryptoKey.computeKey() is calculated, where in AES shared secret case, the Diffie-Hellman key is truncated.

If the shared secret key length is bigger than the Diffie-Hellman key length, an error is raised.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.Create

Initializes an xml.CryptoKey object. Constructor of an empty CryptoKey object depending on a url.

Syntax

```
xml.CryptoKey.Create(
   url STRING )
  RETURNING object xml.CryptoKey
```

1. *url* defines a key identifier according to the XML-Signature and XML-Encryption specification or the Diffie-Hellman specification.

Usage

Returns a CryptoKey object or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.CreateDerivedKey

Constructor of an empty CryptoKey object intended to be derived before use, and depending on a url.

Syntax

```
xml.CryptoKey.CreateDerivedKey(
   url STRING )
  RETURNING object xml.CryptoKey
```

1. *url* defines a key identifier according to the XML-Signature and XML-Encryption specification.

Usage

Returns a CryptoKey object or NULL. Only symmetric and HMAC keys can be derived.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.CreateFromNode

Constructor of a new CryptoKey object depending on a url and from a XML node, according to the XML-Signature and XML-Encryption specification.

Syntax

```
xml.CryptoKey.CreateFromNode(
   url STRING,
   node xml.DomNode )
  RETURNING object xml.CryptoKey
```

1. *url* defines a key identifier restricted to PUBLIC/PRIVATE keys.
2. *node* is an ELEMENT node whose local name is either:
   - **DSAKeyValue** or **RSAKeyValue** and belonging to the XML-Signature namespace **http://www.w3.org/2000/09/xmldsig#**
   - **RSAKeyPair** and belonging to the XKMS 2.0 namespace**http://www.w3.org/2002/03/xkms#**

Usage

Returns a CryptoKey object or NULL.

If the local name is RSAKeyValue or RSAKeyPair, the URL must be a RSA key. If the local name is DSAKeyValue, the URL must be a DSA key.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.deriveKey

Derives the symmetric or HMAC CryptoKey object using the given *method* identifier and concatenating the optional *label*, the mandatory *seed* value and the optional *created* date as initial random value.

Syntax

```
deriveKey(
   method STRING,
   label STRING,
   seed STRING,
   created STRING,
   offset INTEGER,
   size INTEGER )
```

1. *method* is the identifier of the algorithm to apply to the password and its inputs.
2. *label* is the optional label input.
3. *seed*, the mandatory seed input, is the a valid Base64 string representing a random binary data you can obtain with the security.RandomGenerator.CreateRandomNumber on page 2279 helper method.
4. *created* is the optional created date input.
5. *offset* is the number of bytes the resulting octet stream must be shifted to obtain the derived key.
6. *size* defines the number of bytes of the resulting derived key.

Usage

If it is a symmetric key, the size can be 0, or must match the original key according to key identifier.

See Derived keys on page 2224 for more details.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.generateKey

Generates a random key of given size (in bits).

Syntax

```
generateKey(
   size INTEGER )
```

1. *size* is the size of the key to generate.

Usage

For symmetric keys, the size is fixed by the key identifier and cannot be changed. The only authorized values are the real key size or NULL.

For Diffie-Hellman, the input parameter (size INTEGER) is the size of the Diffie-Hellman modulus. If the given size is greater than zero (0), it populates the Diffie-Hellman object by randomly generating a modulus

of the given size and a private key, and computes the public key. The used generator is two (2). If the given size is zero (0), it completes the Diffie-Hellman object by choosing a private key and computing the public key according to the previously loaded parameters. For more details on loading parameters, see Table 526: Object methods: Load and save on page 2228.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.getFeature

Returns the value of the given feature for this CryptoKey object, or NULL.

Syntax

```
getFeature(
   feature STRING )
  RETURNING value STRING
```

1. *feature* is the CryptoKey feature.

Usage

Returns NULL if the feature is not set.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.getSHA1

Returns the SHA1 encoded key identifier in a base64 encoded STRING.

Syntax

```
getSHA1()
  RETURNING keyId STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.getSize

Returns the size of the key in bits.

Syntax

```
getSize()
  RETURNING size INTEGER
```

Usage

For a Diffie-Hellman key, returns the size of the key; the size of a Diffie-Hellman key is actually the size of the modulus. If the modulus is not available (null or equal to zero), the method returns zero. In this situation, a return of zero does NOT mean the key is corrupt or unusable.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.getType
Returns the type of key.

Syntax

```
getType()
  RETURNING type STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.getUrl
Returns the key identifier as an URL, as defined in the XML-Signature and XML-Encryption specification.

Syntax

```
getUrl()
  RETURNING keyId STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.getUsage
Returns the usage of the key.

Syntax

```
getUsage()
  RETURNING usage STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadBIN
Loads a symmetric or HMAC key from a file in raw format.

Syntax

```
loadBIN(
   file STRING )
```

1. *file* is the file name or an entry in the FGLPROFILE file.

Usage

Raw format means that the data in the file are read without any transformation, and will be stored as it in the key.

For instance, if you file contains **hello**, it has the same effect as calling setKey() with **hello** as parameter.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadDER

Loads an asymmetric DSA key, an asymmetric RSA key or Diffie-Hellman parameters from a file in DER format.

Syntax

```
loadDER(
   file STRING )
```

1. *file* is the file name or an entry in the FGLPROFILE file.

Usage

If the DSA or RSA private key or Diffie-Hellman parameters is protected with a password, the recommended way is to unprotect it with the `openssl` tool and to put the key file on a restricted file system. However, you can use a script or the `fglpass` agent to provide the password to the application.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadFromString

Loads the given key in BASE64 string format into a CryptoKey object.

Syntax

```
loadFromString(
   str STRING )
```

1. *str* is the string to load.

Usage

For Diffie-Hellman, the input parameter is a base64 encoded string containing the Diffie-Hellman parameters. This method populates the Diffie-Hellman key with the modulus and generator in the base64 encoded string. This is useful for the parameters exchange step between two peers.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadPEM

Loads an asymmetric DSA key, an asymmetric RSA key or Diffie-Hellman parameters from a file in PEM format.

Syntax

```
loadPEM(
   file STRING )
```

1. *file* is the file name or an entry in the FGLPROFILE file.

Usage

If the DSA or RSA private key or Diffie-Hellman parameters is protected with a password, the recommended way is to unprotect it with the `openssl` tool and to put the key file on a restricted file system. However, you can use a script or the `fglpass` agent to provide the password to the application.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadPrivate

Loads the private asymmetric RSA key in the given XML document into the private part of this CryptoKey object, according to the XKMS2.0 specification.

Syntax

```
loadPrivate(
   xml xml.DomDocument )
```

1. *xml* is a DomDocument object.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadPublic

Loads the public asymmetric RSA or DSA key in the given XML document into the public part of this CryptoKey object, according to the XML-Signature specification for DSA and RSA key value.

Syntax

```
loadPublic(
   xml xml.DomDocument )
```

Usage

For Diffie-Hellman, the input parameter is an xml.DomDocument object containing a representation of the Diffie-Hellman key. This method populates the Diffie-Hellman object with the parameters and the public key contained in the given xml.DomDocument according to the XML-Signature specification for the Diffie-Hellman key values. If the public key node exists in the xml document but is empty, it won't be possible to use the key unless the document contains valid modulus and generator parameters and you call `generateKey` with a size of zero (0). In this case, you won't be in possession of the other peer's public key.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.loadPublicFromString

Populate the current CryptoKey object with the passed public key.

Syntax

```
loadPublicFromString(
   pubKeyStr STRING )
```

1. *pubKeyStr* is the public part of the key in base64 form.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.savePrivate

Saves the private part of an asymmetric RSA CryptoKey object into a XML document according to the XKMS2.0 specification.

Syntax

```
savePrivate()
  RETURNING object xml.DomDocument
```

Usage

Returns an DomDocument object containing the private part of an asymmetric RSA key.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.savePublic

Saves the public part of an asymmetric RSA or DSA CryptoKey object or the parameters and the public key of the Diffie-Hellman object into a XML document according to the XML-Signature specification for DSA and RSA and Diffie-Hellman key values.

Syntax

```
savePublic()
  RETURNING object xml.DomDocument
```

Usage

For Diffie-Hellman, it is useful for the public key exchange between the two peers.

See also the RetrievalMethod feature.

Returns an DomDocumentobject.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.savePublicToString

Save the current xml.CryptoKey's public part in the returned base64 string.

Syntax

```
savePublicToString()
  RETURNING str STRING
```

Usage

Returns the public part of the key in base64 form (STRING).

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.saveToString

Saves the CryptoKey object into a BASE64 string format.

Syntax

```
saveToString()
  RETURNING str STRING
```

Usage

For Diffie-Hellman, returns the Diffie-Hellman key's modulus and generator in a base64 encoded string. This is useful for the parameters exchange step between the two peers.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.setFeature

Sets or resets the value of a feature for a CryptoKey object.

Syntax

```
setFeature(
   feature STRING,
   value STRING )
```

1. *feature* is the name of the feature.
2. *value* is the value to set for the named feature.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoKey.setKey

Defines the value of a HMAC or Symmetric key.

Syntax

```
setKey(
   key STRING )
```

1. *key* is the value.

Usage

The value can be a password and must be of the size corresponding to the key identifier.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

Supported kind of keys
Supported kind of keys for the `xml.CryptoKey` class.

Table 520: Supported kind of keys

Identifier	Description	Usage	Type
http://www.w3.org/2000/09/xmldsig#dsa-sha1	Asymmetric DSA key with SHA1 for signature purposes. Uses a private DSA key for signature and needs an associated public DSA key or X509 certificate containing it, to verify it. See specification for details.	SIGNATURE	PUBLIC or PRIVATE
http://www.w3.org/2000/09/xmldsig#rsa-sha1	Asymmetric RSA key with SHA1 for signature purposes. Uses a private RSA key for signature and needs an associated public RSA key or X509 certificate containing it, to verify it. See specification for details.	SIGNATURE	PUBLIC or PRIVATE
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256	Asymmetric RSA key with SHA256 for signature purposes. Uses a private RSA key for signature and needs an associated public RSA key or X509 certificate containing it, to verify it. See specification for details.	SIGNATURE	PUBLIC or PRIVATE
http://www.w3.org/2000/09/xmldsig#hmac-sha1	Message Authentication Code key with SHA1 for signature purposes. Uses a same password for signature and to verify it, and key size is free. See specification for details.	SIGNATURE	HMAC

Identifier	Description	Usage	Type
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256	Message Authentication Code key with SHA256 for signature purposes. Uses a same password for signature and to verify it, and key size is free. See specification for details.	SIGNATURE	HMAC
http://www.w3.org/2001/04/xmlenc#aes128-cbc	Symmectric AES128 key for encryption purposes. Uses a common key of 128bits for encrypting and decrypting XML documents. See specification for details.	ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#aes192-cbc	Symmetric AES192 key for encryption purposes. Uses a common key of 192bits for encrypting and decrypting XML documents. See specification for details.	ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#aes256-cbc	Symmetric AES256 key for encryption purposes. Uses a common key of 256bits for encrypting and decrypting XML documents. See specification for details.	ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#tripledes-cbc	Symmetric TripleDes key for encryption purposes. Uses a common key of 192bits for encrypting and decrypting XML documents. See specification for details.	ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#kw-aes128	Symmetric AES128 key wrap for key encryption purposes.	KEY ENCRYPTION	SYMMETRIC

Identifier	Description	Usage	Type
	Uses a common key of 128bits for encrypting and decrypting a symmetric key. See specification for details.		
http://www.w3.org/2001/04/xmlenc#kw-aes192	Symmetric AES192 key wrap for key encryption purposes. Uses a common key of 192bits for encrypting and decrypting a symmetric key. See specification for details.	KEY ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#kw-aes256	Symmetric AES256 key wrap for key encryption purposes. Uses a common key of 256bits for encrypting and decrypting a symmetric key. See specification for details.	KEY ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#kw-tripledes	Symmetric TripleDes key wrap for key encryption purposes. Uses a common key of 192bits for encrypting and decrypting a symmetric key. See specification for details.	KEY ENCRYPTION	SYMMETRIC
http://www.w3.org/2001/04/xmlenc#rsa-1_5	Asymmetric RSA key for key encryption purposes. Uses a public RSA key or a X509 certificate containing it to encrypt a symmetric key, and needs the associated private RSA key to decrypt it. See specification for details.	KEY ENCRYPTION	PUBLIC or PRIVATE

Identifier	Description	Usage	Type
http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p	Asymmetric RSA key for key encryption purposes. Uses a public RSA key or a X509 certificate containing it to encrypt a symmetric key, and needs the associated private RSA key to decrypt it. See specification for details.	KEY ENCRYPTION	PUBLIC or PRIVATE
Diffie-Hellman identifier: http://www.w3.org/2001/04/xmlenc#DHKeyValue	Diffie-Hellman key agreement algorithm. Derives a shared secret. The resulting shared secret is a HMAC or symmetric key for encryption purposes.	KEY AGREEMENT	PUBLIC or PRIVATE

Derived keys

Key derivation is used on symmetric or HMAC keys to avoid the direct usage of a shared secret password in secured operations. If two parties share a secret password that is successfully hacked by a third party, any future operation becomes unsecure, and the initial two parties do not even realize that their exchanges are unsafe. However, if a different password based on that shared secret password is used for each new secured operation, even if one operation is compromised, it will only unsecure that operation, but not other operations.

The derivation consists of applying an algorithm with some additional inputs (such as a random seed value) to a password in order to obtain another password that is then used in one secured operation. Of course, the algorithm and its additional inputs must also be shared to enable the computation of the same derived key by someone that is intended to decrypt the message.

Note that passwords are often only composed of alphanumeric characters that eases a bit more the job of a hacker, whereas a derived key is composed of any binary data produced by the algorithm used for the derivation.

Table 521: Derived keys methods

Method	Description
http://schemas.xmlsoap.org/ws/2005/02/sc/dk/p_sha1	Only algorithm supported. See specification for details.

CryptoKey Features
Features of the `xml.CryptoKey` class.

Table 522: CryptoKey Features

Name	Description
`KeyName` See specification for details.	Defines or returns whether a user-defined key name is added during a XML**signature**

Name	Description
	or **encryption** in order to identify it to other applications, or by the key store. The default value is NULL, meaning that no key name is used.
`KeyValue` See specification for details.	Defines or returns whether the public part of the asymmetric key is added during a XML **signature** or **encryption**. Only for RSA and DSA keys. The default value is FALSE, meaning that no key value is used.
`RetrievalMethod` See specification for details.	Defines or returns the URL where the XML form of: • a DSA or RSA public key will be set during a XML **signature**, and loaded during a XML **verification** process. • a RSA public key will be set and used to encrypt a XML node during XML encryption • a symmetric key with encryption usage will be used to encrypt a XML node or decrypt it back The default value is NULL, meaning that no retrieval method is used. The XML form of a DSA or RSA public key can be obtain by the `savePublic` method. The XML form of a symmetric key can be obtain by the `encryptKey` method.

Examples
Examples using the `xml.CryptoKey` class.

Examples:

- Loading an asymmetric RSA key on page 2225
- Generating a symmetric AES256 key on page 2226
- Setting a HMAC key on page 2226
- Deriving a HMAC key on page 2226
- Computing the shared secret with Diffie-Hellman on page 2227

Loading an asymmetric RSA key

```
IMPORT xml

MAIN
  DEFINE key xml.CryptoKey
  LET key = xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#rsa-1_5")
  TRY
    CALL key.loadPEM("RSA1024Key.pem")
    CALL key.setFeature("KeyName","MyRsaKey")
    DISPLAY "Key size (in bits) : ",key.getSize() # displays 1024 (bits)
    DISPLAY "Key type : ",key.getType() # displays PRIVATE or PUBLIC
    DISPLAY "Key usage : ",key.getUsage() # displays KEYENCRYPTION
  CATCH
    DISPLAY "Unable to load key :",STATUS
```

```
  END TRY
END MAIN
```

Note: All keys in PEM or DER format were created with the OpenSSL tool.

Generating a symmetric AES256 key

```
IMPORT xml

MAIN
  DEFINE key xml.CryptoKey
  LET key = xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#aes256-
cbc")
  TRY
    CALL key.generateKey(NULL)
    DISPLAY "Key size (in bits) : ",key.getSize() # displays 256 (bits)
    DISPLAY "Key type : ",key.getType() # displays SYMMETRIC
    DISPLAY "Key usage : ",key.getUsage() # displays ENCRYPTION
  CATCH
    DISPLAY "Unable to generate key :",STATUS
  END TRY
END MAIN
```

Note: All keys in PEM or DER format were created with the OpenSSL tool.

Setting a HMAC key

```
IMPORT xml

MAIN
  DEFINE key xml.CryptoKey
  LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#hmac-
sha1")
  TRY
    CALL key.setKey("secretpassword")
    # displays 112 (size of secretpassword in bits)
    DISPLAY "Key size (in bits) : ",key.getSize()
    DISPLAY "Key type : ",key.getType() # displays HMAC
    DISPLAY "Key usage : ",key.getUsage() # displays SIGNATURE
  CATCH
    DISPLAY "Unable to set key :",STATUS
  END TRY
END MAIN
```

Note: All keys in PEM or DER format were created with the OpenSSL tool.

Deriving a HMAC key

```
IMPORT xml
IMPORT com

MAIN
  DEFINE key xml.CryptoKey
  # will contain a random binary data encoded in Base64
  DEFINE seedBase64 STRING
  LET key = xml.CryptoKey.CreateDerivedKey(
    "http://www.w3.org/2000/09/xmldsig#hmac-sha1")
  TRY
    # Creates a random 24 bytes long binary data encoded into a Base64 form
 string
    CALL key.setKey("secretpassword")
```

```
    # Derives the 14 bytes long "secretpassword" into a 64 bytes long key
    # from a random 24 bytes long seed value and shifting the resulting key
    # from 255 bytes
    LET seedBase64 = com.Util.CreateRandomString(24)
    CALL key.deriveKey(
      "http://schemas.xmlsoap.org/ws/2005/02/sc/dk/p_sha1",
      NULL,seedBase64,NULL,255,64)
    # Displays 512 (size of 'secretpassword' derivation in bits)
    DISPLAY "Key size (in bits) : ",key.getSize()
    # Note: Key is derived and can be used in
    # any encryption or signature function
  CATCH
    DISPLAY "Unable to derive key :",STATUS
  END TRY
END MAIN
```

Note: All keys in PEM or DER format were created with the OpenSSL tool.

Computing the shared secret with Diffie-Hellman

Load the Diffie-Hellman parameters from a PEM file, the other peer's public key from an XML file and compute the shared secret.

Function `generateKey` is called with a 0, parameters are already filled.

```
FUNCTION BuildSharedSecret(DHdoc)
  DEFINE myKey, othersPubKey, sharedSecret xml.CryptoKey
  DEFINE DHdoc xml.DomDocument
  LET myKey =
    xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#DHKeyValue")
  LET othersPubKey =
    xml.CryptoKey.Create("http://www.w3.org/2001/04/xmlenc#DHKeyValue " )
  TRY
    CALL othersPubKey.loadPublic(DHdoc)

    # populate myKey with the parameters previously generated by the
    # other peer.
    CALL myKey.loadPEM("DHParam.pem")

    # Randomly generate a private key and compute the public key. Key
    # length is the parameters length.
    CALL myKey.generateKey(0)
    LET sharedSecret = myKey.computeKey(othersPubKey,
      "http://www.w3.org/2000/09/xmldsig#hmac-sha1")

  CATCH
    DISPLAY "ERROR : should not raise exception”
    EXIT PROGRAM (-1)
  END TRY
END FUNCTION
```

The CryptoX509 class

The `xml.CryptoX509` class provides methods to manipulate X509 certificates needed for identification of individual persons, groups or any entities during XML encryption or signature process.

It also provides additional load and save functions to interact with other applications in XML or in BASE64, such as in WS-Security compliant applications. It follows the XML-Signature and XML-Encryption specifications.

The `STATUS` variable is set to zero after a successful method call.

Important: This class is not supported on GMI mobile devices.

xml.CryptoX509 methods
Methods for the `xml.CryptoX509` class.

Table 523: Class methods: Creation

Name	Description
`xml.CryptoX509.Create()` `  RETURNING object xml.CryptoX509`	Constructor of an empty CryptoX509 object.

`xml.CryptoX509.CreateFromNode(`
`  node xml.DomNode )`
`  RETURNING object xml.CryptoX509`
	Constructor of a new CryptoX509 object from a XML X509 certificate node, according to the XML-Signature specification

Table 524: Object methods: Access

Name	Description
`getIdentifier()` `  RETURNING idpart STRING`	Gets the identification part of an X509 certificate
`getThumbprintSHA1()` `  RETURNING setp STRING`	Gets the SHA1 encoded thumbprint identifying this X509 certificate.

Table 525: Object methods: Modify

Name	Description

`createPublicKey(`
`  url STRING )`
`  RETURNING object xml.CryptoX509`
	Creates a new public CryptoKey object for the given url, from the public key embedded in a certificate.

Table 526: Object methods: Load and save

Name	Description
`load(` `  xml xml.DomDocument )`	Loads the given XML document with ds:X509Data as root node according to the XML-Signature specification, into the CryptoX509 object.
`loadDER(` `  file STRING )`	Loads a X509 certificate from a file in DER format.
`loadFromString(` `  str STRING )`	Loads the given X509 certificate in BASE64 string format into this CryptoX509 object.
`loadPEM(`	Loads a X509 certificate from a file in PEM format.

Name	Description
file STRING)	
save() RETURNING *object* xml.DomDocument	Saves the CryptoX509 certificate into a XML document with ds:X509Data as root node according to the XML-Signature specification.
saveToString() RETURNING *cert* STRING	Saves the CryptoX509 certificate into a BASE64 string format.

Table 527: Object methods: Feature

Name	Description
getFeature(*feature* STRING) RETURNING *value* STRING	Get the value of a given feature of a CryptoX509 object.
setFeature(*feature* STRING, *value* STRING)	Sets or resets the given feature for this CryptoX509 object.

xml.CryptoX509.Create

Constructor of an empty CryptoX509 object.

Syntax

```
xml.CryptoX509.Create()
  RETURNING object xml.CryptoX509
```

Usage

Returns a CryptoX509 object or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.CreateFromNode

Constructor of a new CryptoX509 object from a XML X509 certificate node, according to the XML-Signature specification

Syntax

```
xml.CryptoX509.CreateFromNode(
   node xml.DomNode )
  RETURNING object xml.CryptoX509
```

1. *node* is an ELEMENT DomNode node with **X509Data** as local name, and belonging to the XML-Signature namespace **http://www.w3.org/2000/09/xmldsig#**.

Usage

Returns a CryptoX509 object or NULL.

If the X509 certificate is incomplete, the certificate will be created from the application global certificate list if one of **SubjectName** or **Issuer** matches. (See addCertificate for more details.)

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.createPublicKey

Creates a new public CryptoKey object for the given url, from the public key embedded in a certificate.

Syntax

```
createPublicKey(
   url STRING )
  RETURNING object xml.CryptoX509
```

1. *url* is the given url.

Usage

Creates a new public CryptoKey object for the given url, from the public key embedded in this certificate if any; NULL otherwise.

Returns a CryptoX509 object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.getFeature

Get the value of a given feature of a CryptoX509 object.

Syntax

```
getFeature(
   feature STRING )
  RETURNING value STRING
```

1. *feature* is a feature of the CryptoX509 object.

Usage

Returns the value of the given feature for this CryptoX509 object, or NULL if feature is not set.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.getIdentifier

Gets the identification part of an X509 certificate

Syntax

```
getIdentifier()
  RETURNING idpart STRING
```

Usage

Returns the identification part of this X509 certificate in a STRING.

Example: `/C=FR/ST=France/L=Schiltigheim/O=MC/OU=My Company Name/CN=cert`

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.getThumbprintSHA1

Gets the SHA1 encoded thumbprint identifying this X509 certificate.

Syntax

```
getThumbprintSHA1()
  RETURNING setp STRING
```

Usage

Returns the SHA1 encoded thumbprint identifying this X509 certificate in a BASE64 encoded STRING.

Example: `CM4y6z7zzLnTGMe1lE46RKIKAPI=`

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.load

Loads the given XML document with ds:X509Data as root node according to the XML-Signature specification, into the CryptoX509 object.

Syntax

```
load(
   xml xml.DomDocument )
```

1. xml is a DomDocument object.

Usage

If the X509 certificate in the XML document is incomplete, the certificate will be loaded from the global certificate list if one of **SubjectName** or **Issuer** matches.

See the w3.org site for more information on ds:X509Data as root node according to the XML-Signature specification.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.loadDER

Loads a X509 certificate from a file in DER format.

Syntax

```
loadDER(
   file STRING )
```

1. *file* is the filename or an entry in the FGLPROFILE file.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.loadFromString

Loads the given X509 certificate in BASE64 string format into this CryptoX509 object.

Syntax

```
loadFromString(
   str STRING )
```

1. *str* is the X509 certificate in BASE64 string format to load.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.loadPEM

Loads a X509 certificate from a file in PEM format.

Syntax

```
loadPEM(
   file STRING )
```

1. *file* is the filename or an entry in the FGLPROFILE file.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.save

Saves the CryptoX509 certificate into a XML document with ds:X509Data as root node according to the XML-Signature specification.

Syntax

```
save()
  RETURNING object xml.DomDocument
```

Usage

See the w3.org site for more information on ds:X509Data as root node according to the XML-Signature specification.

(See also the RetrievalMethod feature)

Returns an xml.DomDocument object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.saveToString

Saves the CryptoX509 certificate into a BASE64 string format.

Syntax

```
saveToString()
  RETURNING cert STRING
```

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.CryptoX509.setFeature

Sets or resets the given feature for this CryptoX509 object.

Syntax

```
setFeature(
   feature STRING,
   value STRING )
```

1. *feature* is the feature to be set.
2. *value* is the value to set.

Usage

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

CryptoX509 Features
Features of the `xml.CryptoX509` class.

Table 528: CryptoX509 Features

Feature	Description
`X509Certificate` See specification for details.	Defines or returns whether the complete X509 certificate is added during XML **signature** or **encryption**. Default value is FALSE.
`X509SubjectName` See specification for details.	Defines or returns whether the subject name of the X509 certificate is added during XML **signature** or **encryption**. Default value is FALSE.
`X509IssuerSerial` See specification for details.	Defines or returns whether the issuer name and serial number of the X509 certificate is added during XML **signature** or **encryption**. Default value is FALSE.

Feature	Description
`RetrievalMethod` See specification for details.	Defines or returns the URL where the XML form of the X509 certificate will be set during a XML **signature**, and loaded during a XML **verification** process, and based on that CryptoX509 object. Default value is NULL, meaning that no retrieval method is used. **Note:** The XML form of a X509 certificate can be obtain by the `save()` method.

Examples
Examples using the `xml.CryptoX509` class.

Topics:

- Loading a certificate from a PEM file on page 2234
- Creating a public key for signature verification from a certificate on page 2234
- Saving the subjectName of a certificate in XML on page 2235

Loading a certificate from a PEM file

```
IMPORT xml

MAIN
  DEFINE x509 xml.CryptoX509
  LET x509 = xml.CryptoX509.Create()
  TRY
    CALL x509.loadPEM("Certificate.crt");
    DISPLAY "Id : ",x509.getIdentifier()
  CATCH
    DISPLAY "Unable to load certificate :",STATUS
  END TRY
END MAIN
```

Note: All certificates in PEM format were created with the OpenSSL tool.

Creating a public key for signature verification from a certificate

```
IMPORT xml

MAIN
  DEFINE x509 xml.CryptoX509
  DEFINE key xml.CryptoKey
  LET x509 = xml.CryptoX509.Create()
  TRY
    CALL x509.loadPEM("RSA1024Certificate.crt");
  CATCH
    DISPLAY "Unable to load certificate :",STATUS
    EXIT PROGRAM
  END TRY
  TRY
    LET key = x509.createPublicKey("http://www.w3.org/2000/09/xmldsig#rsa-
sha1")
    DISPLAY "Key size (in bytes) : ",key.getSize() # displays 1024 (bits)
    DISPLAY "Key type : ",key.getType() # displays PUBLIC
    DISPLAY "Key usage : ",key.getUsage() # displays SIGNATURE
  CATCH
    DISPLAY "Unable to create public key :",STATUS
  END TRY
```

```
END MAIN
```

Note: All certificates in PEM format were created with the OpenSSL tool.

Saving the subjectName of a certificate in XML

```
IMPORT xml

MAIN
  DEFINE x509 xml.CryptoX509
  DEFINE key xml.CryptoKey
  DEFINE doc xml.DomDocument
  LET x509 = xml.CryptoX509.Create()
  TRY
    CALL x509.loadPEM("RSA1024Certificate.crt");
  CATCH
    DISPLAY "Unable to load certificate :",STATUS
    EXIT PROGRAM
  END TRY
  TRY
    CALL x509.setFeature("X509SubjectName",TRUE)
    LET doc = x509.save()
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("RSAX509SubjectName.xml")
  CATCH
    DISPLAY "Unable to save certificate :",STATUS
  END TRY
END MAIN
```

Note: All certificates in PEM format were created with the OpenSSL tool.

The Signature class
The `xml.Signature` class provides methods to create detached, enveloped or enveloping XML signatures of one or more references of XML documents or document fragments, and to determine whether a signed referenced document has been modified afterwards.

It follows the XML-Signature specifications.

The `STATUS` variable is set to zero after a successful method call.

Important: This class is not supported on GMI mobile devices.

xml.Signature methods
Methods for the `xml.Signature` class.

Table 529: Class methods: Creation

Name	Description
`xml.Signature.Create()` `  RETURNING sign xml.Signature`	Constructor of a blank Signature object.

`xml.Signature.CreateFromNode(`
`  signode xml.DomNode )`
`  RETURNING sign xml.Signature`
	Constructor of a new Signature object from a XML Signature node, according to the XML-Signature specification.

Table 530: Class methods: Object access

Name	Description

`xml.Signature.RetrieveObjectDataListFromSignatureNode(`
`  sign xml.DomNode,`
`  ind INTEGER )`
`  RETURNING nodelist xml.DomNodeList`
	Returns a DomNodeList containing all embedded XML nodes related to the signature object of index *ind* in the XML Signature node *sign*.

Note: In addition to this class method categorized under Object Access, there are also object methods. These are listed in Table 537: Object methods: Object access on page 2238.

Table 531: Object methods: Key and certificate

Name	Description
`setCertificate(` `  cert xml.CryptoX509 )`	Defines the X509 certificate to be added to the Signature object when signing a document.
`setKey(` `  key xml.CryptoKey )`	Defines the key used for signing or validation.

Table 532: Object methods: Modifier

Name	Description
`setCanonicalization(` `  url STRING )`	Sets the canonicalization method to use for the signature.
`setID(` `  id STRING )`	Sets an ID value for the signature.

Table 533: Object methods: Access

Name	Description
`getCanonicalization()` `  RETURNING ident STRING`	Returns one of the four canonicalization identifier of the signature.
`getDocument()` `  RETURNING doc xml.DomDocument`	Returns a new DomDocument object representing the signature in XML.
`getID()` `  RETURNING id STRING`	Returns the ID value of the signature.
`getSignatureMethod()`	Returns the algorithm method of the signature.

Name	Description
`RETURNING algo STRING`	
`getType()` `RETURNING str STRING`	Returns a string with the type of the Signature object.

Table 534: Object methods: Reference modifier

Name	Description

`appendReferenceTransformation(`
`ind INTEGER,`
`trans STRING,`
`... )`
	Appends a transformation related to the reference of index *ind*, and is executed before any computation

`createReference(`
`uri STRING,`
`digest STRING )`
`RETURNING ind INTEGER`
	Creates a new reference that will be signed with the `compute()` method

`setReferenceID(`
`ind INTEGER,`
`value STRING )`
	Sets an ID *value* for the signature reference of index *ind*.

Table 535: Object methods: Reference access

Name	Description
`getReferenceCount()` `RETURNING num INTEGER`	Returns the number of references in this Signature object.

`getReferenceDigest(`
`ind INTEGER )`
`RETURNING algo STRING`
	Returns the digest algorithm identifier of the reference of index *ind* in this Signature object.

`getReferenceURI(`
`ind INTEGER )`
`RETURNING uri STRING`
	Returns the URI of the reference of index *ind* in this Signature object.

`getReferenceID(`
`ind INTEGER )`
`RETURNING value STRING`
	Returns the ID value of the reference of index *ind* in this Signature object, or NULL if there is none.

`getReferenceTransformation(`
`ind INTEGER,`
`pos INTEGER )`
	Gets the transformation identifier related to the reference of index *ind* at position *pos* in the list of transformation.

Name	Description
`RETURNING ident STRING`	
`getReferenceTransformationCount( ind INTEGER ) RETURNING num INTEGER`	Returns the number of transformation related to the reference of index *ind*.

Table 536: Object methods: Object modifier

Name	Description
`appendObjectData( ind INTEGER, node xml.DomNode )`	Appends a copy of a XML node *node* to the signature object of index *ind*.
`createObject() RETURNING ind INTEGER`	Creates a new object that will embed additional XML nodes.
`setObjectID( ind INTEGER, value STRING )`	Sets an ID *value* for the signature object of index *ind*.

Table 537: Object methods: Object access

Name	Description
`getObjectCount() RETURNING num INTEGER`	Returns the number of objects in this Signature object.
`getObjectId( ind INTEGER ) RETURNING id STRING`	Returns the ID value of the signature object of index *ind* in this Signature object.

Note: In addition to these object methods categorized under Object Access, there is also a class method. It is listed in Table 530: Class methods: Object access on page 2236.

Table 538: Object methods: Signature computation and verification

Name	Description
`compute( doc xml.DomDocument )`	Computes the signature of all references set in this Signature object.
`signString( key xml.CryptoKey, strToSign STRING )`	Sign the passed string according to the specified key.

Name	Description
`RETURNING sig STRING`	
`verify( doc xml.DomDocument ) RETURNING flag INTEGER`	Verifies whether all references in this Signature object haven't changed.
`verifyString( key xml.CryptoKey, signedStr STRING, signature STRING ) RETURNING flag INTEGER`	Verify the signature is consistent with the given key and the original message.

xml.Signature.appendObjectData

Appends a copy of a XML node *node* to the signature object of index *ind*.

Syntax

```
appendObjectData(
   ind INTEGER,
   node xml.DomNode )
```

1. *ind* is the index in this Signature object.
2. node is the XML node.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.appendReferenceTransformation

Appends a transformation related to the reference of index *ind*, and is executed before any computation

Syntax

```
appendReferenceTransformation(
   ind INTEGER,
   trans STRING,
   ... )
```

1. *ind* is the index in this Signature object.
2. *trans* represents an URL as identifier of the transformation algorithm.

Usage

A transformation modifies the reference URI before signing or validating it. Several transformations are executed one after another, and only once the last transformation was applied, is the reference really signed or verified.

Depending on the transformation identifier, additional parameters are necessary.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.compute

Computes the signature of all references set in this Signature object.

Syntax

```
compute(
   doc xml.DomDocument )
```

1. *doc* is the XML document.

Usage

If the signature type is:

- Enveloping: then *doc* must be NULL because all document fragment references are inside the Signature itself
- Enveloped: then *doc* must be the XML document where the signature must be added afterwards to get a valid enveloped signature
- Detached: then *doc* can be NULL if all references are absolute, otherwise it can be the XML document the fragment references are referencing

See XML Signature concepts for more details.

Also, see Windows™ .NET special recommendation.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.Create

Constructor of a blank Signature object.

Syntax

```
xml.Signature.Create()
  RETURNING sign xml.Signature
```

Usage

Returns a Signature object or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.CreateFromNode

Constructor of a new Signature object from a XML Signature node, according to the XML-Signature specification.

Syntax

```
xml.Signature.CreateFromNode(
   signode xml.DomNode )
  RETURNING sign xml.Signature
```

1. *sign* is the XML Signature node.

Usage

Returns a Signature object or NULL.

The *node* must be an ELEMENT node with **Signature** as the local name, and belonging to the XML-Signature namespace **http://www.w3.org/2000/09/xmldsig#**, as defined here.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.createObject

Creates a new object that will embed additional XML nodes.

Syntax

```
createObject()
  RETURNING ind INTEGER
```

Usage

The returned value represents the index for any further manipulation of this signature object.

> **Note:** An object is enveloping additional XML nodes, but is not necessarily signed unless there is a reference on it.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.createReference

Creates a new reference that will be signed with the compute() method

Syntax

```
createReference(
   uri STRING,
   digest STRING )
  RETURNING ind INTEGER
```

1. *uri* represents the data to be signed.
2. *digest* is a URL as identifier for the hash algorithm.

Usage

The returned value represents the index for any further manipulation of this reference.

The *uri* can be:

- An absolute url such as **http://, https://, tcp://, tcps://, file:///** and **alias://** (see FGLPROFILE Configuration for more details about URL mapping with aliases), and where the data can be a XML document or any kind of data such as images or html pages.
- NULL to sign the whole document, but only one NULL is allowed in the entire signature.
- A fragment like **#tobesigned**. Note that a DOM node fragment is identified via the value of an attribute of type ID such as **xml:id** or any attribute whose type was changed to ID with setIdAttribute() or setIdAttributeNS().

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getCanonicalization

Returns one of the four canonicalization identifier of the signature.

Syntax

```
getCanonicalization()
  RETURNING ident STRING
```

Usage

Returns the canonicalization identifier of the signature.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getDocument

Returns a new DomDocument object representing the signature in XML.

Syntax

```
getDocument()
  RETURNING doc xml.DomDocument
```

Usage

Returns a xml.DomDocument object.

If the type of the signature is **enveloped**, it's up to the user to add it at the right place in the XML document it is intended to sign.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getID

Returns the ID value of the signature.

Syntax

```
getID()
  RETURNING id STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getObjectCount

Returns the number of objects in this Signature object.

Syntax

```
getObjectCount()
  RETURNING num INTEGER
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getObjectId

Returns the ID value of the signature object of index *ind* in this Signature object.

Syntax

```
getObjectId(
   ind INTEGER )
  RETURNING id STRING
```

1. *ind* is the index in this Signature object.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getReferenceCount

Returns the number of references in this Signature object.

Syntax

```
getReferenceCount()
  RETURNING num INTEGER
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getReferenceDigest

Returns the digest algorithm identifier of the reference of index *ind* in this Signature object.

Syntax

```
getReferenceDigest(
   ind INTEGER )
  RETURNING algo STRING
```

1. *ind* is the index in this Signature object.

Usage

Returns the digest algorithm identifier of the reference of index *ind* in this Signature object.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getReferenceID

Returns the ID value of the reference of index *ind* in this Signature object, or NULL if there is none.

Syntax

```
getReferenceID(
   ind INTEGER )
  RETURNING value STRING
```

1. *ind* is the index in this Signature object.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getReferenceTransformation

Gets the transformation identifier related to the reference of index *ind* at position *pos* in the list of transformation.

Syntax

```
getReferenceTransformation(
   ind INTEGER,
   pos INTEGER )
  RETURNING ident STRING
```

1. *ind* is the index in this Signature object.
2. *pos* is the position in the list of transformation.

Usage

Returns the transformation identifier related to the reference of index *ind*, and at position *pos* in the list of transformation.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getReferenceTransformationCount

Returns the number of transformation related to the reference of index *ind*.

Syntax

```
getReferenceTransformationCount(
   ind INTEGER )
  RETURNING num INTEGER
```

1. *ind* is the index in this Signature object.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getReferenceURI

Returns the URI of the reference of index *ind* in this Signature object.

Syntax

```
getReferenceURI(
   ind INTEGER )
  RETURNING uri STRING
```

1. *ind* is the index in this Signature object.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getSignatureMethod

Returns the algorithm method of the signature.

Syntax

```
getSignatureMethod()
  RETURNING algo STRING
```

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.getType

Returns a string with the type of the Signature object.

Syntax

```
getType()
  RETURNING str STRING
```

Usage

The string can be Detached, Enveloped, Enveloping or Invalid according to the XML-Signature specification.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.RetrieveObjectDataListFromSignatureNode

Returns a DomNodeList containing all embedded XML nodes related to the signature object of index *ind* in the XML Signature node *sign*.

Syntax

```
xml.Signature.RetrieveObjectDataListFromSignatureNode(
   sign xml.DomNode,
   ind INTEGER )
  RETURNING nodelist xml.DomNodeList
```

1. *sign* is the XML Signature node.
2. *ind* is the index of the signature object.

Usage

Returns a DomNodeList

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.setCanonicalization

Sets the canonicalization method to use for the signature.

Syntax

```
setCanonicalization(
   url STRING )
```

1. *url* is one of the four canonicalization identifier.

Usage

The default value is the c14nmethod.

> **Note:** Windows™ .NET default c14n canonicalization method is not compatible with the W3C standard, therefore it is recommended to use the exc-c14n method when inter-operating with a Windows™ system.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.setCertificate

Defines the X509 certificate to be added to the Signature object when signing a document.

Syntax

```
setCertificate(
   cert xml.CryptoX509 )
```

1. *cert* is the X509 certificate to be added.

Usage

If NULL, no certificate is added.

During the computation of the signature, some certificate information can be added according to the feature set on that CryptoX509 object. If no features are set, the complete X509 certificate is automatically added.

During the verification of a signature the certificate set with the setCertificate method isn't used. See XML Signature concepts for more details.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.setID

Sets an ID value for the signature.

Syntax

```
setID(
   id STRING )
```

1. *id* is the ID value to be set.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.setKey

Defines the key used for signing or validation.

Syntax

```
setKey(
   key xml.CryptoKey )
```

1. key is the key to be used for signing or validation.

Usage

Only RSA, DSA or HMAC keys intended for SIGNATURE are allowed.

During the computation of the signature, some key information can be added according to the feature set on that CryptoKey object. If no features are set, nothing is added. See XML Signature concepts for more details.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.setObjectID

Sets an ID *value* for the signature object of index *ind*.

Syntax

```
setObjectID(
   ind INTEGER,
   value STRING )
```

1. *ind* is the index value.
2. *value* is the value to be set.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.setReferenceID

Sets an ID *value* for the signature reference of index *ind*.

Syntax

```
setReferenceID(
   ind INTEGER,
   value STRING )
```

1. *ind* is the index value.
2. *value* is the value to be set.

Usage

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.signString

Sign the passed string according to the specified key.

Syntax

```
signString(
  key xml.CryptoKey,
  strToSign STRING
 )
 RETURNING sig STRING
```

1. *key* is the key to be used for the signature.
2. *strToSign* is the string to be sign.

Usage

The key can be a HMAC key, a RSA private key or a DSA private key. The signing process is performed with SHA-1 digest, as recommended by the XmlSec specification.

Returns *sig*, or the signature in base64 format.

This method does not belong to the XML encryption specification.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.verify

Verifies whether all references in this Signature object haven't changed.

Syntax

```
verify(
   doc xml.DomDocument )
  RETURNING flag INTEGER
```

1. *doc* is the XML document.

Usage

Returns TRUE if valid, FALSE otherwise.

If the signature type is:

- Enveloping: then *doc* must be NULL because all document fragment references are inside the Signature itself,
- Enveloped: then *doc* must be the XML document where the signature was enveloped,
- Detached: then *doc* can be null if all references are absolute, otherwise it can be the XML document the fragment references are referencing.

See XML Signature concepts for more details.

By default, the validation process uses the CryptoKey set with `setKey()` to verify the signature. However, if the signature contains a X509 certificate or a X509 retrieval method, it uses the list of trusted certificate, or if the signature contains a RSA or DSA retrieval method, it uses the RSA or DSA public key automatically loaded.

> **Note:** See Windows™ .NET special recommendation.

Before loading the XML document to verifiy the signature, you might need to set some options to retrieve the "id" nodes with the `xml.DomDocument.setFeature()` method:

```
DEFINE doc xml.DomDocument
...
CALL doc.setFeature(feature, TRUE)
...
```

Here *feature* must be `"auto-id-attribute"` if the `"id"` attribute has no namespace, or `"auto-id-qualified-attribute"`, when `"id"` has a namespace.

This is especially needed when you encounter error messages such as:

```
Xml security operation failed : libxml2 library function failed :
 expr=xpointer(id('id-1436767651')).
```

Meaning that the parser could not find the `"id"` attribute in the XML document.

Note that the `"auto-id-*"` features will declare all XML attributes where the name is `"id"`, `"ID"`, `"Id"` or `"iD"` to be of type `ID`, and thus be usable via `xml.DomDocument.getElementById()` method used during signature validation.

If needed, you can also set features for a specific attribute with the `xml.DomNode.setIdAttribute()` method, or with the `xml.DomNode.setIdAttributeNS()` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Signature.verifyString

Verify the signature is consistent with the given key and the original message.

Syntax

```
verifyString(
   key xml.CryptoKey,
   signedStr STRING,
   signature STRING )
  RETURNING flag INTEGER
```

1. *key* is the key to use for verification.
2. *signedStr* is the signed string in its clear form.
3. *signature* is the signature to be verified.

Usage

The key can be a HMAC key, a RSA private key or a DSA private key. The HMAC key must be the same as the one used for signing. The public RSA and DSA key must be the public key corresponding to the private key used for signing.

Returns 1 when verification is successful; 0 (zero) is returned if verification fails.

This method does not belong to the XML encryption specification.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

XML Signature concepts

The purpose of a signature is to guarantee the integrity of a XML document, that it was not altered, and that it still contains the same data as when it was created. An additional purpose of a signature is to authenticate the author of the document. There are different ways to achieve this guarantee.

Sign and verify with a common shared HMAC key

Use if the sender of the XML document and the receiver share a common secret key.

How to sign

1. Create a HMAC key with the constructor of the CryptoKey class.
2. Set or load the common shared key value in the CryptoKey object.
3. Create a blank signature with the constructor of the Signature class.
4. Assign the CryptoKey object to the Signature object.
5. Create one or more references to be signed.
6. Compute the signature.
7. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a HMAC key with the constructor of the CryptoKey.
2. Set or load the common shared key value in the CryptoKey object.
3. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
4. Assign the CryptoKey object to the Signature object.
5. Verify the signature validity.

Sign with the originator private RSA or DSA key, and verify with the originator public RSA or DSA key

Use if the receiver of the XML document has the RSA or DSA public key of the sender.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a blank signature with the constructor of the Signature class.
4. Assign the CryptoKey object to the Signature object.
5. Create one or more references to be signed.
6. Compute the signature.
7. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a RSA or DSA key with the constructor of the CryptoKey class.

2. Load the RSA or DSA public key into the CryptoKey object.
3. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
4. Assign the CryptoKey object to the Signature object.
5. Verify the signature validity.

Sign with the originator private RSA or DSA key, and verify with a RSA or DSA retrieval method

Use if the sender of the XML document provides the public RSA or DSA key in XML form (and via http, tcp or a file protocol).

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Set the RetrievalMethod feature on the CryptoKey object with the URL where the XML form of the public RSA or DSA key is available.
4. Create a blank signature with the constructor of the Signature class.
5. Assign the CryptoKey object to the Signature object.
6. Create one or more references to be signed.
7. Compute the signature.
8. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
2. Verify the signature validity.

Note: There is no key nor certificate to set in the Signature object during validation.

Sign with the originator private RSA or DSA key, and verify with the originator X509 certificate associated to the private RSA or DSA key

Use if the receiver of the XML document has the X509 certificate associated to the RSA or DSA private key.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a blank signature with the constructor of the Signature class.
4. Assign the CryptoKey object to the Signature object.
5. Create one or more references to be signed.
6. Compute the signature.
7. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a X509 certificate with the constructor of the CryptoX509 class.
2. Load the X509 certificate into the CryptoKey object.
3. Create the RSA or DSA public key from the X509 certificate of the CryptoX509 object.

4. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
5. Assign the CryptoKey object containing the public key to the Signature object.
6. Verify the signature validity.

Sign with the originator private RSA or DSA key, and verify with trusted X509 certificates

Use if the sender of the XML document adds a X509 certificate that was signed by another trusted X509 certificate.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a X509 certificate with the constructor of the CryptoX509 class.
4. Load the X509 certificate associated to the RSA or DSA private key into the CryptoKey object.
5. Create a blank signature with the constructor of the Signature class.
6. Assign the CryptoKey object to the Signature object.
7. Assign the CryptoX509 object to the Signature object.
8. Create one or more references to be signed.
9. Compute the signature.
10. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a X509 certificate with the constructor of the CryptoX509 class.
2. Load the X509 certificate that was used to sign the originator X509 certificate into the CryptoX509 object.
3. Add the X509 certificate as trusted certificate to the application.
4. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
5. Verify the signature validity.

Note: Point 1 to 3 can be omitted if entry `xml.application.calist` has been set in FGLPROFILE file with the trusted certificate.

Note: There is no key nor certificate to set in the Signature object during validation.

Sign with the originator private RSA or DSA key, and verify with a X509 certificate retrieval method and trusted X509 certificates

Use if the sender of the XML document adds a X509 retrieval method that was signed by another trusted X509 certificate.

Only the originator can sign a message with this specific pair of keys. Any other peer needs the corresponding public key and does not have access to the private key.

How to sign

1. Create a RSA or DSA key with the constructor of the CryptoKey class.
2. Load the RSA or DSA private key into the CryptoKey object.
3. Create a X509 certificate with the constructor of the CryptoX509 class.
4. Set the RetrievalMethod feature on the CryptoX509 object with the URL where the XML form of the originator X509 certificate is available.
5. Create a blank signature with the constructor of the Signature class.
6. Assign the CryptoKey object to the Signature object.

7. Assign the CryptoX509 object to the Signature object.
8. Create one or more references to be signed.
9. Compute the signature.
10. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a X509 certificate with the constructor of the CryptoX509 class.
2. Load the X509 certificate that was used to sign the originator X509 certificate into the CryptoX509 object.
3. Add the X509 certificate as trusted certificate to the application.
4. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
5. Verify the signature validity.

   Note: Steps 1 - 3 can be omitted if entry `xml.application.calist` has been set in FGLPROFILE file with the trusted certificate.

   Note: There is no key or certificate to set in the Signature object during validation.

Sign with a named key and verify using the keystore

Use if the sender and the receiver exchange multiple XML documents signed with different keys.

How to sign

1. Create a HMAC, RSA or DSA key with the constructor of the CryptoKey class.
2. Set the HMAC key or load the RSA or DSA key in the CryptoKey object.
3. Set the KeyName feature with the name identifying the key.
4. Create a blank signature with the constructor of the Signature class.
5. Assign the CryptoKey object to the Signature object.
6. Create one or more references to be signed.
7. Compute the signature.
8. Retrieve the XML signature document from the Signature object.

How to verify

1. Create a HMAC, RSA or DSA key with the constructor of the CryptoKey.
2. Set the HMAC key or load the RSA or DSA key in the CryptoKey object.
3. Set the KeyName feature with the name identifying the key.
4. Register the key to be used by key name for any signature verification.
5. Create a signature with the constructor of the Signature class and from a XML signature node obtain after the above compute operation.
6. Verify the signature validity.

   Note: Steps 1 - 4 should be done once at application startup for each key used in the application. Steps 5 - 6 can then quickly be executed for any XML signature to be checked.

Digest identifier

Table 539: Digest identifiers

Identifier	Description
http://www.w3.org/2000/09/xmldsig#sha1 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 160 bits.

Identifier	Description
	Note: It is the only digest algorithm recommended by the W3C.
http://www.w3.org/2001/04/xmlenc#sha512 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 512 bits.
http://www.w3.org/2001/04/xmldsig-more#sha384 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 384 bits.
http://www.w3.org/2001/04/xmlenc#sha256 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 256 bits.
http://www.w3.org/2001/04/xmldsig-more#sha224 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a SHA algorithm of 224 bits.
http://www.w3.org/2001/04/xmldsig-more#md5 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a MD5 algorithm.
http://www.w3.org/2001/04/xmlenc#ripemd160 See specification for details.	Computes the digest of the reference set with createReference(), by applying a hash operation using a RIPEMD algorithm.

Transformation identifier

Table 540: Transformation identifiers

Identifier	Description	Additional Parameters
http://www.w3.org/2000/09/xmldsig#base64 See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into the raw data associated to a BASE64 encoded form. This is intended to sign the raw data associated with the BASE64 encoded content of an element. See specification for details.	No
http://www.w3.org/TR/2001/REC-xml-c14n-20010315 See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document without any XML comments. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: The following two XML nodes are equivalent but would produce different hash values if not canonicalized. • `<tag Attr1="hello" Attr2="world"/>` • `<tag Attr2="world" Attr1="hello" />`	No

Identifier	Description	Additional Parameters
	See specification for details.	
http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document keeping all XML comments. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: The following two XML nodes are equivalent but would produce different hash values if not canonicalized. • `<tag Attr1="hello" Attr2="world"/>` • `<tag Attr2="world" Attr1="hello" />` See specification for details.	No
http://www.w3.org/2001/10/xml-exc-c14n# See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document without any XML comments, and removing all unused namespaces declaration. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: The following two XML nodes are equivalent but would produce different hash values if not canonicalized. • `<tag Attr1="hello" Attr2="world"/>` • `<tag Attr2="world" Attr1="hello" />` See specification for details.	No
http://www.w3.org/2001/10/xml-exc-c14n#WithComments See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a canonicalized XML document keeping all XML comments, and removing all unused namespaces declaration. This is intended to transform two equivalent XML documents into a standardized XML representation in order to obtain the same hash value. For instance: Following two XML nodes are equivalent but would produce different hash values if not canonicalized. • `<tag Attr1="hello" Attr2="world"/>` • `<tag Attr2="world" Attr1="hello" />` See specification for details.	No

Identifier	Description	Additional Parameters
http://www.w3.org/2000/09/xmldsig #enveloped-signature See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into the same XML document or fragment, but without the Signature node. This is intended to create **enveloped** signatures where the**<dsig:Signature>** node is inside the document, but without taking it into account during signature computation or verification. See specification for details.	No

http://www.w3.org/TR/1999/REC-xpath-19991116
 	
See specification for details.
	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a XML document according to a XPath filtering expression applied to each node of the input document, where the expression represents a predicate to the XPath expression`(//. | //@* | //namespace::*)`.
	
	In other words: **`(//. | //@* | //namespace::*)[expr]`**
	
	This is intended to identify the nodes to be signed using a XPath expression instead of an attribute of type ID.
	
	For instance: The following samples output only the **MyCode** node of the input document:
	
	`CALL s.appendReferenceTransformation( i, "http://www.w3.org/TR/1999/REC-xpath-19991116", "ancestor-or-self::MyCode", NULL`
	
	`CALL s.appendReferenceTransformation( i, "http://www.w3.org/TR/1999/REC-xpath-19991116", "ancestor-or-self::p:MyCode", "p", "http://www.tempuri.org")`
	
	See specification for details.
		XPath expression, followed by NULL or a list of prefix, namespace matching the XPath expression.

http://www.w3.org/2002/06/xmldsig-filter2 See specification for details.	Transforms the output from the previous transformation (or the reference if there is no previous transformation), into a XML document according to a XPath filtering 2.0 expression applied to the entire document at once.	XPathFilter2.0 type (**intersect**, **subtract** or **union**), followed by the XPath expression, followed by NULL or a list of prefix, namespace

Identifier	Description	Additional Parameters

	This is intended to identify the nodes to be signed using a XPath expression instead of an attribute of type ID, and to perform fast and more complex operations such as **intersect**, **subtract** or **union**.
	
	For instance: The following samples output the entire document without the **MyCode** node child of the **MyElement** root node:
	
	`CALL s.appendReferenceTransformation( i, "http://www.w3.org/2002/06/xmldsig-filter2", "subtract", "/MyElement/MyCode")`
	
	`CALL s.appendReferenceTransformation( i, "http://www.w3.org/2002/06/xmldsig-filter2", "subtract", "/p1:MyElement/p2:MyCode", "p2", "http://www.tempuri.org/ns2", "p1", "http://www.tempuri.org/ns1")`
	
	See specification for details.
		matching the XPath expression.

Examples
Examples using the XML Signature class.

- Create a detached signature using a HMAC key on page 2257
- Verify a detached signature using a HMAC key on page 2258
- Create an enveloping signature using a DSA key on page 2259
- Verify an enveloping signature using a X509 certificate on page 2260
- Create an enveloped signature using a RSA key on page 2260
- Verify an enveloped signature using a RSA key on page 2261

Create a detached signature using a HMAC key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE root xml.DomNode
  DEFINE sig xml.Signature
  DEFINE key xml.CryptoKey
  DEFINE index INTEGER
  # Create DomDocument object
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significant in crytography,
  # therefore it is recommended that you remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
```

```
    # Load document to be signed
    CALL doc.load("MyDocument.xml")
    # Create HMAC key
    LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#hmac-
sha1")
    CALL key.setKey("secretpassword")
     # Create signature object with the key to use
    LET sig = xml.Signature.Create()
    CALL sig.setKey(key)
    # Set XML node to be signed. In our case, the node with attribute
    # 'xml:id="code"'
    LET index = sig.createReference("#code",
      "http://www.w3.org/2000/09/xmldsig#sha1")
    # Set canonicalization method on the XML fragment to be signed.
    CALL sig.appendReferenceTransformation(index,
      "http://www.w3.org/2001/10/xml-exc-c14n#")
    # Compute detached signature
    CALL sig.compute(doc)
    # Retrieve signature document
    LET doc=sig.getDocument()
    # Save signature on disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("MyDocumentDetachedSignature.xml")
  CATCH
    DISPLAY "Unable to create a detached signature :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Verify a detached signature using a HMAC key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE sig xml.Signature
  DEFINE key xml.CryptoKey
  DEFINE isVerified INTEGER
  # Create DomDocument object
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load Signature into a DomDocument object
    CALL doc.load("MyDocumentDetachedSignature.xml")
    # Create signature object from DomDocument root node
    LET sig = xml.Signature.CreateFromNode(doc.getDocumentElement())
    # Create HMAC key and assign it to the signature object
    LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#hmac-
sha1")
    CALL key.setKey("secretpassword")
    CALL sig.setKey(key)
    # Load original XML document into a DomDocument object
    CALL doc.load("MyDocument.xml")
    # Verify detached signature validity of original document
    LET isVerified = sig.verify(doc)
    # Notice that if something has been modified in the node
    # with attribute 'xml:id="code"' of the original XML document,
    # the program will display "FAILED".
    IF isVerified THEN
```

```
      DISPLAY "Signature OK"
    ELSE
      DISPLAY "Signature FAILED"
    END IF
  CATCH
    DISPLAY "Unable to verify the detached signature :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Create an enveloping signature using a DSA key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE root xml.DomNode
  DEFINE sig xml.Signature
  DEFINE key xml.CryptoKey
  DEFINE index INTEGER
  DEFINE objInd INTEGER
  # Create DomDocument object
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load document to be signed
    CALL doc.load("MyDocument.xml")
    # Create DSA key and load it from file
    LET key = xml.CryptoKey.Create(
      "http://www.w3.org/2000/09/xmldsig#dsa-sha1")
    CALL key.loadPEM("DSAKey.pem")
    # Create signature object with the key to use
    LET sig = xml.Signature.Create()
    CALL sig.setKey(key)
    # Create an object inside the signature to envelop the root node
    LET objInd = sig.createObject()
    # Set the object id to get a reference
    CALL sig.setObjectId(objInd,"data")
    # Copy the enveloping node from the document
    CALL sig.appendObjectData(objInd,doc.getDocumentElement())
    # Set the reference to be signed on the object node.
    # In our case, the object node with attribute 'data'
    LET index = sig.createReference("#data",
      "http://www.w3.org/2000/09/xmldsig#sha1")
    # Set canonicalization method on the enveloping object to be signed.
    CALL sig.appendReferenceTransformation(index,
      "http://www.w3.org/2001/10/xml-exc-c14n#")
    # Compute enveloping signature
    CALL sig.compute(NULL)
    # Retrieve signature document
    LET doc=sig.getDocument()
    # Save signature on disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("MyDocumentEnvelopingSignature.xml")
  CATCH
    DISPLAY "Unable to create an enveloping signature :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Verify an enveloping signature using a X509 certificate

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE sig xml.Signature
  DEFINE cert xml.CryptoX509
  DEFINE pub xml.CryptoKey
  DEFINE isVerified INTEGER
  # Create DomDocument object
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load Signature into a DomDocument object
    CALL doc.load("MyDocumentEnvelopingSignature.xml")
    # Create signature object from DomDocument root node
    LET sig = xml.Signature.CreateFromNode(doc.getDocumentElement())
    # Create X509 certificate
    LET cert = xml.CryptoX509.Create()
    CALL cert.loadPEM("DSACertificate.crt")
    # Create public key from that X509 certificate
    LET pub = cert.createPublicKey(
      "http://www.w3.org/2000/09/xmldsig#dsa-sha1")
    # Assign it to the signature
    CALL sig.setKey(pub)
    # Verify enveloping signature validity
    LET isVerified = sig.verify(NULL)
    # Notice that if something has been modified in the signature
    # or if the certificate isn't associated to the
    # private DSA key of exemple 3,
    # the program will display "FAILED".
    IF isVerified THEN
      DISPLAY "Signature OK"
    ELSE
      DISPLAY "Signature FAILED"
    END IF
  CATCH
    DISPLAY "Unable to verify the enveloping signature :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Create an enveloped signature using a RSA key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE doc2 xml.DomDocument
  DEFINE root xml.DomNode
  DEFINE node xml.DomNode
  DEFINE signNode xml.DomNode
  DEFINE sig xml.Signature
  DEFINE key xml.CryptoKey
  DEFINE index INTEGER
```

```
  # Create DomDocument object
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load document to be signed
    CALL doc.load("MyDocument.xml")
    # Create rsa key
    LET key = xml.CryptoKey.Create("http://www.w3.org/2000/09/xmldsig#rsa-
sha1")
    CALL key.loadPEM("RSAKey.pem")
    # Create signature object with the key to use
    LET sig = xml.Signature.Create()
    CALL sig.setKey(key)
    # Set XML node to be signed. In our case, the node with
    # attribute 'xml:id="code"'
    LET index = sig.createReference("#code",
      "http://www.w3.org/2000/09/xmldsig#sha1")
    # Add enveloped method to not take the XML signature node into account
    # when computing the entire document.
    CALL sig.appendReferenceTransformation(index,
            "http://www.w3.org/2000/09/xmldsig#enveloped-signature")
    # Set canonicalization method on the XML fragment to be signed.
    CALL sig.appendReferenceTransformation(index,
            "http://www.w3.org/2001/10/xml-exc-c14n#")
    # Compute enveloped signature
    CALL sig.compute(doc)
    # Retrieve signature document
    LET doc2=sig.getDocument()
    # Append the signature node to the original document to get
    # a valid enveloped signature
    # Notice that the enveloped signature can be added anywhere in the
    # original document
    LET signNode = doc2.getDocumentElement() # Get Signature node
    # Import it into the original document
    LET node = doc.importNode(signNode,true)
    # Retrieve the original document root node
    LET root = doc.getDocumentElement()
    # Append the signature node as last child of the original document
    CALL root.appendChild(node)
    # Save document with enveloped signature back to disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("MyDocumentEnvelopedSignature.xml")
  CATCH
    DISPLAY "Unable to create an enveloped signature :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Verify an enveloped signature using a RSA key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE sig xml.Signature
  DEFINE key xml.CryptoKey
  DEFINE list xml.DomNodeList
  DEFINE isVerified INTEGER
  # Create DomDocument object
```

```
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load original document with enveloped signature into a DomDocument
 object
    CALL doc.load("MyDocumentEnvelopedSignature.xml")
    # Because the signature can be anywhere in the original document,
    # we must first retrieve it
    LET list = doc.getElementsByTagNameNS("Signature",
      "http://www.w3.org/2000/09/xmldsig#")
    IF list.getCount() != 1 THEN
      DISPLAY "Unable to find one Signature node"
      EXIT PROGRAM (-1)
    ELSE
      LET node = list.getItem(1)
    END IF
    # Create RSA key
    LET key = xml.CryptoKey.Create(
      "http://www.w3.org/2000/09/xmldsig#rsa-sha1")
    CALL key.loadPEM("RSAKey.pem")
    # Create signature object from DomNode object and set RSA key to use
    LET sig = xml.Signature.CreateFromNode(node)
    CALL sig.setKey(key)
    # Verify enveloped signature validity of original document
    LET isVerified = sig.verify(doc)
    # Notice that if something has been modified in the node with
    # attribute 'xml:id="code"' of the original XML document with the
    # enveloped signature, the program will display "FAILED".
    IF isVerified THEN
      DISPLAY "Signature OK"
    ELSE
      DISPLAY "Signature FAILED"
    END IF
  CATCH
    DISPLAY "Unable to verify the enveloped signature :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

The Encryption class

The `xml.Encryption` class provides methods to encrypt and decrypt XML documents, nodes or symmetric keys.

It follows the XML-Encryption specifications.

The STATUS variable is set to zero after a successful method call.

Important: This class is not supported on GMI mobile devices.

xml.Encryption methods

Methods for the `xml.Encryption` class.

Table 541: Class methods: Creation

Name	Description
`xml.Encryption.Create()` `  RETURNING` *enc* `xml.Encryption`	Constructor of an Encryption object.

Table 542: Class methods: String encryption and decryption

Name	Description
`xml.Encryption.DecryptString( key xml.CryptoKey , str STRING ) RETURNING rstr STRING`	Decrypts the encrypted string *str* encoded in BASE64, using the symmetric key *key*, and returns the string in clear text.
`xml.Encryption.EncryptString( key xml.CryptoKey, str STRING ) RETURNING rstr STRING`	Encrypts the string *str* using the symmetric key *key*, and returns the encrypted string encoded in BASE64.
`xml.Encryption.RSADecrypt( key STRING, enc STRING ) RETURNING rstr STRING`	Decrypts the BASE64 encrypted string *enc* using the RSA key *key* and returns it in clear text
`xml.Encryption.RSAEncrypt( key STRING, str STRING ) RETURNING rstr STRING`	Encrypts the string *str* using the RSA key *key* and returns it encoded in BASE64.

The methods listed in Table 542: Class methods: String encryption and decryption on page 2263 do not belong to the XML encryption specification, but are helper functions to allow BDL application to encrypt and decrypt short passwords with RSA keys, or big strings by using symmetric keys. Notice that a common way to encrypt data is to use symmetric keys, and to use RSA keys to encrypt the symmetric key value.

Table 543: Object methods: Key and certificate setting

Name	Description
`getEmbeddedKey() RETURNING key xml.CryptoKey`	Get a copy of the embedded symmetric key that was used in the last decryption operation.
`setCertificate( cert xml.CryptoX509 )`	Assigns a copy of the X509 certificate to this Encryption object.
`setKey( key xml.CryptoKey )`	Assigns a copy of the symmetric key to this Encryption object.
`setKeyEncryptionKey( key xml.CryptoKey )`	Assigns a copy of the **key-encryption** key to this Encryption object.

Table 544: Object methods: XML elements encryption and decryption

Name	Description
`decryptElement( enc xml.DomNode )`	Decrypts the EncryptedData DomNode *enc* using the symmetric key.
`decryptElementContent( enc xml.DomNode )`	Decrypts the EncryptedData DomNode *enc* using the symmetric key.
`encryptElement( node xml.DomNode )`	Encrypts the ELEMENT DomNode *node* and all its children using the symmetric key.
`encryptElementContent( node xml.DomNode )`	Encrypts all child nodes of the ELEMENT DomNode *node* using the symmetric key.

Table 545: Object methods: Detached XML elements encryption and decryption

Name	Description
`decryptElementDetached( enc xml.DomNode ) RETURNING node xml.DomNode`	Decrypts the EncryptedData DomNode *enc* using the symmetric key, and returns it in a new ELEMENT node
`decryptElementContentDetached( enc xml.DomNode ) RETURNING node xml.DomNode`	Decrypts the EncryptedData DomNode *enc* using the symmetric key, and returns all its children in one new DOCUMENT_FRAGMENT_NODE node.
`encryptElementDetached( node xml.DomNode ) RETURNING rnode xml.DomNode`	Encrypts the ELEMENT DomNode *node* and all its children using the symmetric key, and returns them as one new EncryptedData node.
`encryptElementContentDetached( node xml.DomNode ) RETURNING rnode xml.DomNode`	Encrypts all child nodes of the ELEMENT DomNode *node* using the symmetric key, and returns them as one new EncryptedData node.

Table 546: Object methods: Key encryption and decryption

Name	Description
`decryptKey( xml xml.DomDocument, url STRING ) RETURNING key xml.CryptoKey`	Decrypts the EncryptedKey as root in the given XML document, and returns a new CryptoKey of the given kind.
`encryptKey( key xml.CryptoKey )`	Encrypts the given symmetric or HMAC key as an EncryptedKey node and returns it as root node of a new XML document .

Name	Description
RETURNING *doc* xml.DomDocument	

xml.Encryption.Create

Constructor of an Encryption object.

Syntax

```
xml.Encryption.Create()
  RETURNING enc xml.Encryption
```

Usage

Returns a Encryption object or NULL.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.decryptElement

Decrypts the EncryptedData DomNode *enc* using the symmetric key.

Syntax

```
decryptElement(
   enc xml.DomNode )
```

1. *enc* is the encrypted DomNode.

Usage

The encrypted DomNode *enc* is replaced at the same place in the XML document with the resulting ELEMENT DomNode and its children.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.decryptElementContent

Decrypts the EncryptedData DomNode *enc* using the symmetric key.

Syntax

```
decryptElementContent(
   enc xml.DomNode )
```

1. *enc* is the encrypted DomNode.

Usage

The encrypted DomNode *enc* is replaced at the same place in the XML document with the resulting child nodes.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.decryptElementContentDetached

Decrypts the EncryptedData DomNode *enc* using the symmetric key, and returns all its children in one new DOCUMENT_FRAGMENT_NODE node.

Syntax

```
decryptElementContentDetached(
   enc xml.DomNode )
  RETURNING node xml.DomNode
```

1. *enc* is the encrypted DomNode.

Usage

Returns all its children in one new DOCUMENT_FRAGMENT_NODE node.

The resulting child nodes aren't added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the encrypted node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.decryptElementDetached

Decrypts the EncryptedData DomNode *enc* using the symmetric key, and returns it in a new ELEMENT node

Syntax

```
decryptElementDetached(
   enc xml.DomNode )
  RETURNING node xml.DomNode
```

1. *enc* is the encrypted DomNode.

Usage

The resulting DomNode and its children aren't added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the encrypted node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.decryptKey

Decrypts the EncryptedKey as root in the given XML document, and returns a new CryptoKey of the given kind.

Syntax

```
decryptKey(
   xml xml.DomDocument,
   url STRING )
  RETURNING key xml.CryptoKey
```

1. xml is the DomDocument object.
2. url is the string.

Usage

Returns a new CryptoKey of the given kind.

Only symmetric or HMAC keys are allowed.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.DecryptString

Decrypts the encrypted string *str* encoded in BASE64, using the symmetric key *key*, and returns the string in clear text.

Syntax

```
xml.Encryption.DecryptString(
   key xml.CryptoKey ,
   str STRING )
  RETURNING rstr STRING
```

1. *key* is the symmetric key to use for decryption.
2. *str* is the encrypted string for decryption.

Usage

The key must be of usage: **encryption**.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.encryptElement

Encrypts the ELEMENT DomNode *node* and all its children using the symmetric key.

Syntax

```
encryptElement(
   node xml.DomNode )
```

Usage

The ELEMENT DomNode *node* and all its children are replaced at the same place in the XML document with the resulting EncryptedData node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.encryptElementContent

Encrypts all child nodes of the ELEMENT DomNode *node* using the symmetric key.

Syntax

```
encryptElementContent(
   node xml.DomNode )
```

Usage

The child nodes of the ELEMENT DomNode *node* are replaced at the same place in the XML document with the resulting EncryptedData node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.encryptElementContentDetached

Encrypts all child nodes of the ELEMENT DomNode *node* using the symmetric key, and returns them as one new EncryptedData node.

Syntax

```
encryptElementContentDetached(
   node xml.DomNode )
  RETURNING rnode xml.DomNode
```

1. node is the ELEMENT DomNode to encrypt.

Usage

Encrypts all child nodes of the ELEMENT DomNode *node* using the symmetric key, and returns them as one new EncryptedData node.

The resulting DomNode isn't added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the nodes in clear form.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.encryptElementDetached

Encrypts the ELEMENT DomNode *node* and all its children using the symmetric key, and returns them as one new EncryptedData node.

Syntax

```
encryptElementDetached(
   node xml.DomNode )
  RETURNING rnode xml.DomNode
```

1. *node* is the ELEMENT DomNode to encrypt.

Usage

Encrypts the ELEMENT DomNode *node* and all its children using the symmetric key, and returns them as one new EncryptedData node.

The resulting DomNode isn't added at any place in the XML document. It's up to the user to insert it at the right place, and to remove the nodes in clear form.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.encryptKey

Encrypts the given symmetric or HMAC key as an EncryptedKey node and returns it as root node of a new XML document .

Syntax

```
encryptKey(
   key xml.CryptoKey )
```

```
   RETURNING doc xml.DomDocument
```

1. *key* is the given symmetric or HMAC key as an EncryptedKey node.

Usage

Returns it as root node of a new XML document. The key-encryption key must been set otherwise it will fail.

Depending on the feature set on the key-encryption key, the returned XML document will contain an additional KeyInfo node.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.EncryptString

Encrypts the string *str* using the symmetric key *key*, and returns the encrypted string encoded in BASE64.

Syntax

```
xml.Encryption.EncryptString(
   key xml.CryptoKey,
   str STRING )
  RETURNING rstr STRING
```

1. *key* is the key.
2. *str* is the string to be encrypted.

Usage

The key must be of usage: **encryption**.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.getEmbeddedKey

Get a copy of the embedded symmetric key that was used in the last decryption operation.

Syntax

```
getEmbeddedKey()
  RETURNING key xml.CryptoKey
```

Usage

Returns a copy of the embedded symmetric key that was used in the last decryption operation, or NULL if there is none.

An embedded symmetric key is always encrypted, and needs therefore a **key-encryption** key to be set in order to decrypt it.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.RSADecrypt

Decrypts the BASE64 encrypted string *enc* using the RSA key *key* and returns it in clear text

Syntax

```
xml.Encryption.RSADecrypt(
   key STRING,
   enc STRING )
  RETURNING rstr STRING
```

1. *key* is the file name of a RSA private key in PEM format or an entry in the FGLPROFILE file.
2. *enc* is a string that was encrypted with the fglpass tool or with the `xml.Encryption.RSAEncrypt` method.

Usage

RSA decryption is only intended to short strings that cannot exceed the size of the RSA key minus 12 bytes. For instance, if you have a RSA key of 512 bits, you password cannot exceed 512/8-12 = 52 bytes. If you need to handle big strings, you must use symmetric keys and the `DecryptString` method. However, you can use RSA keys to decrypt symmetric key values.

> **Important:** YOU MUST PAY ATTENTION TO RESTRICT ACCESS TO THAT RSA PRIVATE KEY FILE ONLY TO THE PERSON OR GROUP OF PERSON AUTHORIZED.

If the RSA private key is protected with a password the recommended way is to unprotect it with the openssl tool and to put the key file on a restricted file system. But you can also use a script or the fglpass agent to provide the password to the application.

For example, you can encrypt a database password with the fglpass tool and store it in the FGLPROFILE file, then you can decrypt it with the `base.Application.getResourceEntry` and the `xml.Encryption.RSADecrypt` method to connect to the database.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.RSAEncrypt

Encrypts the string *str* using the RSA key *key* and returns it encoded in BASE64.

Syntax

```
xml.Encryption.RSAEncrypt(
   key STRING,
   str STRING )
  RETURNING rstr STRING
```

1. *key* is the file name of a RSA public or private key in PEM format or an entry in the FGLPROFILE file.
2. *str* is the string to be encrypted.

Usage

RSA encryption is only intended to short strings that cannot exceed the size of the RSA key minus 12 bytes. For instance, if you have a RSA key of 512 bits, you password cannot exceed 512/8-12 = 52 bytes. If you need to handle big strings, you must use symmetric keys and the `EncryptString` method. However, you can use RSA keys to encrypt symmetric key values.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.setCertificate

Assigns a copy of the X509 certificate to this Encryption object.

Syntax

```
setCertificate(
   cert xml.CryptoX509 )
```

1. *cert* is the copy of the X509 certificate.

Usage

The certificate will then be added to any further XML document or node encryption.

- NULL is allowed to avoid the certificate being added.
- To encrypt using a certificate, you must use the `createPublicKey` method of the X509 class to obtain the public key embedded in the certificate, and then provide it to the encryption object with above `setKeyEncryptionKey` method.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.setKey

Assigns a copy of the symmetric key to this Encryption object.

Syntax

```
setKey(
   key xml.CryptoKey )
```

1. key is the symmetric key.

Usage

Any further XML document or node encryption or decryption will use that symmetric key.

When decrypting a XML document that has an embedded symmetric key, the embedded key will be used instead.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.Encryption.setKeyEncryptionKey

Assigns a copy of the **key-encryption** key to this Encryption object.

Syntax

```
setKeyEncryptionKey(
   key xml.CryptoKey )
```

1. *key* is the **key-encryption** key.

Usage

Any further XML encryption will use that **key-encryption** key to encrypt the symmetric key set with `setKey()` within the resulting XML, and any further XML decryption will use that **key-encryption** key to decrypt the embedded symmetric key.

- NULL is allowed, meaning that embedded symmetric keys will not be encrypted nor decrypted anymore, assuming that they have been exchanged in another way.
- Only public or private RSA keys, or key-wrap keys are allowed.
- Public RSA keys can encrypt but not decrypt.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Examples

Examples using the xml.Encryption class.

Encrypt a XML node with a symmetric AES128 key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE root xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
  LET doc = xml.DomDocument.Create()
  # Notice that white spaces are significant in crytography,
  # therefore it is recommended that you remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load XML file to be encrypted
    CALL doc.load("XMLFileToBeEncrypted.xml")
    LET root = doc.getDocumentElement()
    # Create symmetric AES128 key for XML encryption purposes
    LET symkey = xml.CryptoKey.Create(
                    "http://www.w3.org/2001/04/xmlenc#aes128-cbc")
    CALL symkey.setKey(">secretpassword<") # password of 128 bits
    CALL symKey.setFeature("KeyName","MySecretKey") # Name the password
                      # in order to identify the key (Not mandatory)
    # Encrypt the entire document
    LET enc = xml.Encryption.Create()
    CALL enc.setKey(symkey) # Set the symmetric key to be used
    CALL enc.encryptElement(root) # Encrypt
    # Save encrypted document back to disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("EncryptedXMLFile.xml")
  CATCH
    DISPLAY "Unable to encrypt XML file :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Decrypt a XML node with a symmetric AES128 key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
  DEFINE list xml.DomNodeList
  DEFINE str String
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
```

```
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load encrypted XML file
    CALL doc.load("EncryptedXMLFile.xml")
    # Retrieve encrypted node (if any) from the document
    LET list = doc.getElementsByTagNameNS(
               "EncryptedData","http://www.w3.org/2001/04/xmlenc#")
    IF list.getCount()==1 THEN
      LET node = list.getItem(1)
    ELSE
      DISPLAY "No encrypted node found"
      EXIT PROGRAM
    END IF
    # Check if symmetric key name matches the expected "MySecretKey"
        (Not mandatory)
    LET list = node.selectByXPath(
                  "dsig:KeyInfo/dsig:KeyName[position()=1 and
                  text()=\"MySecretKey\"]","dsig",
                  "http://www.w3.org/2000/09/xmldsig#")
    IF list.getCount()!=1 THEN
      DISPLAY "Key name doesn't match"
      EXIT PROGRAM
    END IF
    # Create symmetric AES128 key for XML decryption purpose
    LET symkey = xml.CryptoKey.Create(
                    "http://www.w3.org/2001/04/xmlenc#aes128-cbc")
    CALL symkey.setKey(">secretpassword<") # password of 128 bits
    # Decrypt the entire document
    LET enc = xml.Encryption.Create()
    CALL enc.setKey(symkey) # Set the symmetric key to be used
    CALL enc.decryptElement(node) # Decrypt
    # Save encrypted document back to disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("DecryptedXMLFile.xml")
  CATCH
    DISPLAY "Unable to decrypt XML file :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Encrypt a XML node with a generated symmetric key protected with the public RSA key within a X509 certificate

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE root xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
  DEFINE kek xml.CryptoKey
  DEFINE cert xml.CryptoX509
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load XML file to be encrypted
    CALL doc.load("XMLFileToBeEncrypted.xml")
    LET root = doc.getDocumentElement()
    # Load the X509 certificate and retrieve the public RSA key
    # for key-encryption purpose
    LET cert = xml.CryptoX509.Create()
```

```
    CALL cert.loadPEM("RSA1024Certificate.crt")
    LET kek = cert.createPublicKey(
      "http://www.w3.org/2001/04/xmlenc#rsa-1_5")
    # Generate symmetric key for XML encryption purpose
    LET symkey = xml.CryptoKey.Create(
      "http://www.w3.org/2001/04/xmlenc#aes256-cbc")
    CALL symkey.generateKey(NULL)
    # Encrypt the entire document
    LET enc = xml.Encryption.Create()
    CALL enc.setKey(symkey) # Set the symmetric key to be used
    CALL enc.setKeyEncryptionKey(kek) # Set the key-encryption key to
                           # be used for protecting the symmetric key
    CALL enc.setCertificate(cert) # Set the certificate to be added
                                  # (not mandatory)
    CALL enc.encryptElement(root) # Encrypt
    # Save encrypted document back to disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("EncryptedXMLFile.xml")
  CATCH
    DISPLAY "Unable to encrypt XML file :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

Decrypt a XML node encrypted with a symmetric key protected with a private RSA key

```
IMPORT xml

MAIN
  DEFINE doc xml.DomDocument
  DEFINE node xml.DomNode
  DEFINE enc xml.Encryption
  DEFINE symkey xml.CryptoKey
  DEFINE kek xml.CryptoKey
  DEFINE list xml.DomNodeList
  LET doc = xml.DomDocument.Create()
  # Notice that whitespaces are significants in crytography,
  # therefore it is recommended to remove unnecessary ones
  CALL doc.setFeature("whitespace-in-element-content",FALSE)
  TRY
    # Load encrypted XML file
    CALL doc.load("EncryptedXMLFile.xml")
    # Retrieve encrypted node (if any) from the document
    LET list = doc.getElementsByTagNameNS("EncryptedData",
      "http://www.w3.org/2001/04/xmlenc#")
    IF list.getCount()==1 THEN
      LET node = list.getItem(1)
    ELSE
      DISPLAY "No encrypted node found"
      EXIT PROGRAM
    END IF
    # Load the private RSA key
    LET kek = xml.CryptoKey.create(
      "http://www.w3.org/2001/04/xmlenc#rsa-1_5")
    CALL kek.loadPEM("RSA1024Key.pem")
    # Decrypt the entire document
    LET enc = xml.Encryption.Create()
    CALL enc.setKeyEncryptionKey(kek) # Set the key-encryption key to
                                # decrypted the protected symmetric key
    CALL enc.decryptElement(node) # Decrypt
    # Retrieve the embedded symmetric key for futher usage and display
    # info about it
```

```
    LET symkey = enc.getEmbeddedKey()
    DISPLAY "Key size (in bytes) : ",symkey.getSize() # displays 1024
    DISPLAY "Key type : ",symkey.getType() # displays SYMMETRIC
    DISPLAY "Key usage : ",symkey.getUsage() # displays ENCRYPTION
    # Encrypted document back to disk
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("DecryptedXMLFile.xml")
  CATCH
    DISPLAY "Unable to decrypt XML file :",STATUS
  END TRY
END MAIN
```

Note: All keys or certificates in PEM or DER format were created with the OpenSSL tool.

The KeyStore class

The `xml.KeyStore` class provides static methods to handle a key store global to the entire application. It enables to register X509 and trusted certificates, and any kind of key by name for automatic XML signature validation or XML decryption.

The `STATUS` variable is set to zero after a successful method call.

Important: This class is not supported on GMI mobile devices.

xml.KeyStore methods

Methods for the `xml.KeyStore` class.

Table 547: Class methods

Name	Description
`xml.KeyStore.AddCertificate( cert xml.CryptoX509 )`	Registers the given X509 certificate as a certificate for the application. It will be used when an incomplete X509 certificate is detected during signature or encryption to complete the process by checking the certificate issuer name and serial number.
`xml.KeyStore.AddKey( key xml.CryptoX509 )`	Registers the given key by name to the application. It is used for XML signature verification or XML decryption when a key name was specified in the XML KeyInfo node and no other key was set in the Signature or Encryption object.
`xml.KeyStore.AddTrustedCertificate( cert xml.CryptoX509 )`	Registers the given X509 certificate as a trusted certificate for the application. It will be used for signature verification if no other certificate was set for that purpose.

xml.KeyStore.AddCertificate

Registers the given X509 certificate as a certificate for the application. It will be used when an incomplete X509 certificate is detected during signature or encryption to complete the process by checking the certificate issuer name and serial number.

Syntax

```
xml.KeyStore.AddCertificate(
   cert xml.CryptoX509 )
```

1. *cert* is the X509 certificate to register.

Usage

The method has the same effect as the `FGLPROFILE` entry `xml.keystore.x509list`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.KeyStore.AddKey

Registers the given key by name to the application. It is used for XML signature verification or XML decryption when a key name was specified in the XML KeyInfo node and no other key was set in the Signature or Encryption object.

Syntax

```
xml.KeyStore.AddKey(
   key xml.CryptoX509 )
```

1. *key* is the key object xml.CryptoX509 to add to the keystore.

Usage

Registers the given key by name to the application. It is used for XML signature verification or XML decryption when a key name was specified in the XML KeyInfo node and no other key was set in the Signature or Encryption object.

The CryptoKey must have the KeyName feature set, and the name must be unique in the application.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

xml.KeyStore.**AddTrustedCertificate**

Registers the given X509 certificate as a trusted certificate for the application. It will be used for signature verification if no other certificate was set for that purpose.

Syntax

```
xml.KeyStore.AddTrustedCertificate(
   cert xml.CryptoX509 )
```

1. *cert* is the X509 certificate to register.

Usage

Registers the given X509 certificate as a trusted certificate for the application. It will be used for signature verification if no other certificate was set for that purpose.

Has the same effect as the `FGLPROFILE` entry `xml.keystore.calist`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

OM to XML Migration

Information to migrate code that uses the language build-in `om` package, to the Web Services extensions `xml` package.

Before you migrate, make sure that you are using the classes from the appropriate package:

- Classes from the `om` package exist to manipulate the AUI tree.

- Classes from the `xml` package provide full support for XML document processing, and should be used to manipulate XML documents.

Why would you migrate from `om` to `xml` classes and methods?

- You need to be able to utilize a feature (such as a StyleSheet) that requires use of methods from the `xml` library classes.

OM - XML Mapping

Table 548: OM - XML Mapping

OM class method	XML class method(s)
om.DomDocument.create	xml.DomDocument.createDocument
om.DomDocument.createFromXmlFile	xml.DomDocument.load
om.DomDocument.createFromString	xml.DomDocument.loadFromString
om.DomDocument.copy	xml.DomNode.clone
om.DomDocument.createChars	xml.DomDocument.createTextNode
om.DomDocument.createEntity	xml.DomDocument.createEntityReference
om.DomDocument.createElement	xml.DomDocument.createElement
om.DomDocument.getDocumentElement	xml.DomDocument.getFirstDocumentNode
om.DomDocument.getElementById	xml.DomDocument.getElementById + xml.DomNode.setIdAttribute or xml.DomNode.setIdAttributeNS
om.DomDocument.removeElement	xml.DomDocument.removeDocumentNode
om.DomNode.appendChild	xml.DomDocument.createNode + xml.DomNode.appendChild
om.DomNode.createChild	xml.DomDocument.createNode + xml.DomNode.appendChild
om.DomNode.insertBefore	xml.DomNode.insertBeforeChild
om.DomNode.removeChild	xml.DomNode.removeChild
om.DomNode.replaceChild	xml.DomNode.replaceChild
om.DomNode.loadXml	xml.DomDocument.loadFromString
om.DomNode.parse	xml.DomDocument.createNode + add it to the DomDocument
om.DomNode.toString	xml.DomNode.toString
om.DomNode.writeXml	xml.DomDocument.save
om.DomNode.write	xml.DomNode.toString
om.DomNode.getId	N/A
om.DomNode.getTagName	xml.DomNode.getLocalName
om.DomNode.setAttribute	xml.DomNode.setAttribute
om.DomNode.getAttribute	xml.DomNode.getAttribute

OM class method	XML class method(s)
om.DomNode.getAttributeInteger	xml.DomNode.getAttribute + condition for the default value and the cast
om.DomNode.getAttributeString	xml.DomNode.getAttribute + condition for the default value and the cast
om.DomNode.getAttributeName	xml.DomNode.getAttributeNodeItem + xml.DomNode.getLocalName
om.DomNode.getAttributesCount	xml.DomNode.getAttributesCount
om.DomNode.getAttributeValue	xml.DomNode.getAttributeNodeItem + xml.DomNode.getNodeValue
om.DomNode.removeAttribute	xml.DomNode.removeAttribute
om.DomNode.getChildCount	xml.DomNode.getChildrenCount
om.DomNode.getChildByIndex	xml.DomNode.getChildNodeItem
om.DomNode.getFirstChild	xml.DomNode.getFirstChild
om.DomNode.getLastChild	xml.DomNode.getLastChild
om.DomNode.getNext	xml.DomNode.getNextSibling
om.DomNode.getParent	xml.DomNode.getParentNode
om.DomNode.getPrevious	xml.DomNode.getPreviousSibling
om.DomNode.selectByTagName	xml.DomNode.getElementsByTagName
om.DomNode.selectByPath	xml.DomNode.selectByXPath
om.NodeList.item	xml.DomNodeList.getItem
om.NodeList.getLength	xml.DomNodeList.getCount

For more information on Genero built-in classes (such as the OM class), refer to Built-in packages on page 1687.

The security package

The Genero Web Services `security` package provides classes and methods to support basic cryptographic features.

Use the `IMPORT` statement at the top of the module using this library:

```
IMPORT security
```


- The RandomGenerator class on page 2279
- The Base64 class on page 2280
- The HexBinary class on page 2286
- The Digest class on page 2291

The RandomGenerator class
The `security.RandomGenerator` class includes methods for creating random strings or numbers.
security.RandomGenerator methods
Methods of the `security.RandomGenerator` class.

Table 549: Class methods

Name	Description
`security.RandomGenerator.CreateRandomNumber()` `  RETURNING result BIGINT`	Generates a 8-byte strong random number.

`security.RandomGenerator.CreateRandomString(`
`  size INTEGER )`
`  RETURNING result STRING`
	Creates a random base64 string.

`security.RandomGenerator.CreateUUIDString()` `  RETURNING result STRING`	Creates a new universal unique identifier (UUID).

security.RandomGenerator.CreateRandomNumber
Generates a 8-byte strong random number.

Syntax

```
security.RandomGenerator.CreateRandomNumber()
  RETURNING result BIGINT
```

1. *result* is a random big integer.

Usage

Generates a 8-byte strong random number and returns it as a `BIGINT`.

The generated number can then be used for advanced cryptographic features.

This method is based on openssl, using `/dev/random` on Unix and `CryptGenRandom()` on Microsoft Windows, which are following CSPRNG specifications.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.RandomGenerator.CreateRandomString
Creates a random base64 string.

Syntax

```
security.RandomGenerator.CreateRandomString(
   size INTEGER )
  RETURNING result STRING
```

1. *size* is the size of the random string.
2. *result* is the generated random string in Base64.

Usage

Generates a random binary data of *size* bytes long and returns it in a `STRING` encoded in a Base64 form.

The size must be greater than 0.

Use this function when randomness is required, such as in `xml.CryptoKey.deriveKey()` or `security.Digest.CreateDigestString()`.

This method is based on openssl, using `/dev/random` on Unix and `CryptGenRandom()` on Microsoft Windows, which are following CSPRNG specifications.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.RandomGenerator.CreateUUIDString
Creates a new universal unique identifier (UUID).

Syntax

```
security.RandomGenerator.CreateUUIDString()
  RETURNING result STRING
```

1. *result* is the new generated UUID string.

Usage

Generates an universal unique identifier and returns the value as `STRING`.

The generated strings follows the UUID version 4 specification. Version 4 UUIDs have the form `xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx` where `x` is any hexadecimal digit and `y` is one of 8, 9, A, or B.

> **Note:** This method replaces `com.Util.CreateUUIDString()`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The Base64 class
The `security.Base64` class includes methods for encoding to base64 or decoding from base64.
security.Base64 methods
Methods of the `security.Base64` class.

Table 550: Class methods

Name	Description
`security.Base64.FromByte( source BYTE ) RETURNING result STRING`	Encodes the given BYTE data in base64.
`security.Base64.FromHexBinary( source STRING ) RETURNING result STRING`	Decodes the given hexadecimal string to base64.
`security.Base64.FromString( source STRING )`	Encodes the given string in base64.

Name	Description
`RETURNING result STRING`	
`security.Base64.FromStringWithCharset( source STRING, charset STRING ) RETURNING result STRING`	Encodes the given string in base64, according to a given charset.
`security.Base64.LoadBinary( path STRING ) RETURNING result STRING`	Reads data from a file and encodes to base64.
`security.Base64.SaveBinary( path STRING, data STRING )`	Decodes the given base64 string and writes the data to a file.
`security.Base64.ToByte( source STRING, destination BYTE )`	Decodes the given base64 string into a BYTE.
`security.Base64.ToHexBinary( source STRING ) RETURNING result STRING`	Decodes the given base64 string to hexadecimal.
`security.Base64.ToString( source STRING ) RETURNING result STRING`	Decodes the given base64 string.
`security.Base64.ToStringWithCharset( source STRING, charset STRING ) RETURNING result STRING`	Decodes the given base64 string, according to a given charset.
`security.Base64.Xor( b64str1 STRING, b64str2 STRING ) RETURNING result STRING`	Computes the exclusive disjunction between two base64 encoded strings.

security.Base64.FromByte
Encodes the given BYTE data in base64.

Syntax

```
security.Base64.FromByte(
   source BYTE )
  RETURNING result STRING
```

1. *source* is the BYTE to be encoded.
2. *result* is the base64 encoded string.

Usage

Encodes the given BYTE data in base64 and returns the string.

> **Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.FromHexBinary
Decodes the given hexadecimal string to base64.

Syntax

```
security.Base64.FromHexBinary(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string in its hexadecimal form
2. *result* is a string encoded in base64

Usage

Decodes the given hexadecimal string and returns it in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.FromString
Encodes the given string in base64.

Syntax

```
security.Base64.FromString(
   source STRING )
  RETURNING result STRING
```

1. *source* is the string to be encoded.
2. *result* is the base64 encoded string.

Usage

Encodes the given string and returns it in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.FromStringWithCharset
Encodes the given string in base64, according to a given charset.

Syntax

```
security.Base64.FromStringWithCharset(
   source STRING,
   charset STRING )
  RETURNING result STRING
```

1. *source* is the string to be encoded.
2. *charset* is the character set to be used.
3. *result* is the base64 encoded string.

Usage

Encodes the given string and returns it in base64.

Before conversion, the string is converted from the local DVM charset to the specified encoding.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.LoadBinary
Reads data from a file and encodes to base64.

Syntax

```
security.Base64.LoadBinary(
   path STRING )
  RETURNING result STRING
```

1. *path* is the path to the binary file.
2. *result* is a string encoded in base64.

Usage

Reads the file located at `path` and encodes these binary data in Base64 format.

For example, this method can be used to send images through a network.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.SaveBinary
Decodes the given base64 string and writes the data to a file.

Syntax

```
security.Base64.SaveBinary(
   path STRING,
   data STRING )
```

1. *path* is the path to the binary file
2. *data* is a base64 string to be written.

Usage

Decodes the given Base64 string and writes the binary data to the file defined by *path*.

This method can be used to save data from a network on the disk.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.ToHexBinary
Decodes the given base64 string to hexadecimal.

Syntax

```
security.Base64.ToHexBinary(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string encoded in base64.
2. *result* is a string decoded in hexadecimal.

Usage

Decodes the given base64 string and returns it in its hexadecimal form.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.ToByte
Decodes the given base64 string into a BYTE.

Syntax

```
security.Base64.ToByte(
   source STRING,
   destination BYTE )
```

1. *source* is a string in base64.
2. *destination* is the BYTE to fill with data.

Usage

Decodes the given base64 string and fills the BYTE variable with binary data.

> **Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.ToString
Decodes the given base64 string.

Syntax

```
security.Base64.ToString(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string encoded in base64.
2. *result* is the decoded string.

Usage

Decodes the given base64 string and returns it in its clear (human readable) form.

If the base64 string does not contain a human readable data, the method will raise an exception.

If the base64 string contains bytes sequences that do not match a valid character in the current encoding, the method raises a conversion error.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.ToStringWithCharset
Decodes the given base64 string, according to a given charset.

Syntax

```
security.Base64.ToStringWithCharset(
   source STRING,
   charset STRING )
  RETURNING result STRING
```

1. *source* is a string encoded in base64.
2. *charset* is the character set to be used.
3. *result* is the decoded string.

Usage

Decodes the given base64 string and returns it in its clear human readable form, according to a given charset.

The original base64 encoded string is first decoded to a string that will be converted from the specified charset to the local DVM charset. In case of charset conversion error, the error -15700 is raised.

If the base64 string does not contain a human readable data, the method will raise an exception.

If the base64 string contains bytes sequences that do not match a valid character in the current encoding, the method raises a conversion error.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Base64.Xor
Computes the exclusive disjunction between two base64 encoded strings.

Syntax

```
security.Base64.Xor(
   b64str1 STRING,
   b64str2 STRING )
  RETURNING result STRING
```

1. *b64str1* is a first string encoded in base64.
2. *b64str2* is a second string encoded in base64.
3. *result* is the xor result encoded in base64.

Usage

Decodes the two given strings and does an exclusive disjunction between the two binary and returns the result encoded in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

The HexBinary class

The `security.HexBinary` class includes methods for encoding to hexadecimal or decoding from hexadecimal.

security.HexBinary methods

Methods of the `security.HexBinary` class.

Table 551: Class methods

Name	Description
`security.HexBinary.FromBase64( source STRING ) RETURNING result STRING`	Converts a base64 string to the hexadecimal equivalent.
`security.HexBinary.FromByte( source BYTE ) RETURNING result STRING`	Encodes BYTE data in hexadecimal.
`security.HexBinary.FromString( source STRING ) RETURNING result STRING`	Encodes a given string in hexadecimal.
`security.HexBinary.FromStringWithCharset( source STRING, charset STRING ) RETURNING result STRING`	Encodes a given string in hexadecimal, according to a given charset.
`security.HexBinary.LoadBinary( path STRING ) RETURNING result STRING`	Reads binary data from a file and converts it to hexadecimal.
`security.HexBinary.SaveBinary( path STRING, data STRING )`	Decodes an hexadecimal strings and writes the binary data to a file.
`security.HexBinary.ToBase64( source STRING ) RETURNING result STRING`	Converts an hexadecimal string to the base64 equivalent
`security.HexBinary.ToByte( source STRING, destination BYTE )`	Decodes an hexadecimal string into a BYTE variable.
`security.HexBinary.ToString( source STRING ) RETURNING result STRING`	Decodes an hexadecimal string to a clear, human-readable string.
`security.HexBinary.ToStringWithCharset( source STRING, charset STRING )`	Decodes an hexadecimal string to a clear, human-readable string, according to a given charset.

Name	Description
`RETURNING result STRING`	
`security.HexBinary.Xor( hexstr1 STRING, hexstr2 STRING ) RETURNING result STRING`	Computes the exclusive disjunction between two hexadecimal encoded strings.

security.HexBinary.FromBase64
Converts a base64 string to the hexadecimal equivalent.

Syntax

```
security.HexBinary.FromBase64(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string encoded in base64.
2. *result* is a string decoded in hexadecimal.

Usage

Decodes the given base64 string and returns it in its hexadecimal form.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.FromByte
Encodes BYTE data in hexadecimal.

Syntax

```
security.HexBinary.FromByte(
   source BYTE )
  RETURNING result STRING
```

1. *source* is the BYTE to be encoded in hexadecimal.
2. *result* is the encoded hexadecimal string.

Usage

Encodes the given BYTE data in hexadecimal and returns the string.

> **Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.FromString
Encodes a given string in hexadecimal.

Syntax

```
security.HexBinary.FromString(
```

```
    source STRING )
  RETURNING result STRING
```

1. *source* is the source string to be encoded in hexadecimal.
2. *result* is the encoded hexadecimal string.

Usage

Encodes the given string and returns it in hexadecimal.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.FromStringWithCharset
Encodes a given string in hexadecimal, according to a given charset.

Syntax

```
security.HexBinary.FromStringWithCharset(
  source STRING,
  charset STRING )
  RETURNING result STRING
```

1. *source* is the source string to be encoded in hexadecimal.
2. *charset* is the character set to be used.
3. *result* is the encoded hexadecimal string.

Usage

Encodes the given string and returns it in hexadecimal.

Before conversion, the string is converted from the local DVM charset to the specified encoding.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.LoadBinary
Reads binary data from a file and converts it to hexadecimal.

Syntax

```
security.HexBinary.LoadBinary(
   path STRING )
  RETURNING result STRING
```

1. *path* is the path to the binary file.
2. *result* is the string in hexadecimal format.

Usage

Reads the file located at *path* and returns these binary data in hexadecimal format.

For example, this method can be used to send images through a network.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.SaveBinary
Decodes an hexadecimal strings and writes the binary data to a file.

Syntax

```
security.HexBinary.SaveBinary(
   path STRING,
   data STRING )
```

1. *path* is the path to the binary file.
2. *data* is the hexadecimal string to be written.

Usage

Decodes the given hexadecimal string and writes the binary data to the file defined by *path*.

This method can be used to save data from a network on the disk.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.ToBase64
Converts an hexadecimal string to the base64 equivalent

Syntax

```
security.HexBinary.ToBase64(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string in its hexadecimal form.
2. *result* is a string encoded in base64.

Usage

Decodes the given hexadecimal string and returns it in base64.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.ToByte
Decodes an hexadecimal string into a BYTE variable.

Syntax

```
security.HexBinary.ToByte(
   source STRING,
   destination BYTE )
```

1. *source* is a string in hexadecimal.
2. *destination* is the BYTE to fill with data.

Usage

Decodes the given hexadecimal string and fills the BYTE variable with binary data.

Important: The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.ToString
Decodes an hexadecimal string to a clear, human-readable string.

Syntax

```
security.HexBinary.ToString(
   source STRING )
  RETURNING result STRING
```

1. *source* is a string in hexadecimal.
2. *result* is a human readable string.

Usage

Decodes the given hexadecimal string and returns it in its clear, human readable, form. If the hexadecimal string does not contain a human readable string, the method will raise an exception.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.ToStringWithCharset
Decodes an hexadecimal string to a clear, human-readable string, according to a given charset.

Syntax

```
security.HexBinary.ToStringWithCharset(
   source STRING,
   charset STRING )
  RETURNING result STRING
```

1. *source* is a string in hexadecimal.
2. *charset* is the character set to be used.
3. *result* is a human readable string.

Usage

Decodes the given hexadecimal string and returns it in its clear human readable form, according to a given charset.

The original hexadecimal encoded string is first decoded to a string that will then be converted from the specified charset to the local DVM charset. In case of charset conversion error, the error -15700 is raised.

If the hexadecimal string does not contain a human readable string, the method will raise an exception.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.HexBinary.Xor
Computes the exclusive disjunction between two hexadecimal encoded strings.

Syntax

```
security.HexBinary.Xor(
   hexstr1 STRING,
```

```
   hexstr2 STRING )
  RETURNING result STRING
```

1. *hexstr1* is a first string in hexadecimal.
2. *hexstr1* is a second string in hexadecimal.
3. *result* is the xor result encoded in hexadecimal.

Usage

Decodes the two given string and does an exclusive disjunction between the two binary and returns the result in hexadecimal.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

The Digest class

The `security.Digest` class implements digest algorithms to process data.

The class implements several methods that allow you to add data by pieces and process these data with a specified digest algorithm.

Steps to process data with a digest algorithm:

1. Define the digest algorithm with the `security.Digest.CreateDigest` method.
2. Add data to the digest buffer with methods such as `security.Digest.AddData`, `security.Digest.AddBase64Data`, `security.Digest.AddHexBinaryData`, `security.Digest.AddStringData`.
3. When all data pieces are added, the buffer can be processed by calling methods like `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

Alternatively, a simple data string can be processed with the `security.Digest.CreateDigestString` method.

security.Digest methods

Methods of the `security.Digest` class.

Table 552: Class methods

Name	Description
`security.Digest.AddData( data BYTE )`	Adds a data from a BYTE variable to the digest buffer.
`security.Digest.AddBase64Data ( data STRING )`	Adds a data in base64 format to the digest buffer.
`security.Digest.AddHexBinaryData( data STRING )`	Adds a data in hexadecimal format to the digest buffer.
`security.Digest.AddStringData( data STRING )`	Adds a data string to the digest buffer.
`security.Digest.AddStringDataWithCharset( data STRING,`	Adds a data string to the digest buffer, after converting to the specified character set.

Name	Description
charset STRING)	
security.Digest.CreateDigest(*algo* STRING)	Defines a new digest context by specifying the algorithm to be used.
security.Digest.CreateDigestString(*password* STRING, *randBase64* STRING) RETURNING *result* STRING	Creates a SHA1 digest from the given string.
security.Digest.DoBase64Digest() RETURNING *b64Digest* STRING	Creates a digest of the buffered data and returns the result in base64 format.
security.Digest.DoHexBinaryDigest() RETURNING *hexBinDigest* STRING	Creates a digest of the buffered data and returns the result in hexadecimal format.

security.Digest.AddData
Adds a data from a BYTE variable to the digest buffer.

Syntax

```
security.Digest.AddData(
   data BYTE )
```

1. *data* is binary data to be added to the digest buffer.

Usage

Adds the binary data contained in the given BYTE to the digest context.

After adding all data pieces, the buffer can be processed by calling security.Digest.DoBase64Digest or security.Digest.DoHexBinaryDigest.

> **Important:** The BYTE must be located in memory.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.AddBase64Data
Adds a data in base64 format to the digest buffer.

Syntax

```
security.Digest.AddBase64Data (
   data STRING )
```

1. *data* is the base64 data string to be added to the digest buffer.

Usage

Decodes the given base64 string and adds the binary data to the digest buffer.

After adding all data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.AddHexBinaryData
Adds a data in hexadecimal format to the digest buffer.

Syntax

```
security.Digest.AddHexBinaryData(
   data STRING )
```

1. *data* is the hexadecimal data string to be added to the digest buffer.

Usage

Decodes the given hexadecimal string and adds the binary data to the digest buffer.

After adding all data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.AddStringData
Adds a data string to the digest buffer.

Syntax

```
security.Digest.AddStringData(
   data STRING )
```

1. *data* is a human-readable character string to be added to the digest buffer.

Usage

Adds the specified string data to the digest buffer.

After adding all data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.AddStringDataWithCharset
Adds a data string to the digest buffer, after converting to the specified character set.

Syntax

```
security.Digest.AddStringDataWithCharset(
   data STRING,
   charset STRING )
```

1. *data* is a human-readable character string to be added to the digest buffer.
2. *charset* is the charset to be used.

Usage

Adds the specified string data to the digest buffer.

Before adding the string, it is converted from the local DVM charset to the specified encoding.

After adding all data pieces, the buffer can be processed by calling `security.Digest.DoBase64Digest` or `security.Digest.DoHexBinaryDigest`.

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.CreateDigest
Defines a new digest context by specifying the algorithm to be used.

Syntax

```
security.Digest.CreateDigest(
   algo STRING )
```

1. *algo* is the digest algorithm to be used.

Usage

Creates and initializes a digest context to compute data digest according to the given algorithm.

Available digest algorithms are:

- `"SHA1"`
- `"SHA224"`
- `"SHA256"`
- `"SHA384"`
- `"SHA512"`
- `"MD5"`

In case of error, the method throws an exception and sets the `STATUS` variable. Depending on the error, a human-readable description of the problem is available in the `SQLCA.SQLERRM` register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.CreateDigestString
Creates a SHA1 digest from the given string.

Syntax

```
security.Digest.CreateDigestString(
   password STRING,
   randBase64 STRING )
  RETURNING result STRING
```

1. *password* is the password to be digested.
2. *randBase64* is a random string in Base64.
3. *result* is a base64 encoded string.

Usage

Computes the SHA1 digest from a *password* value and an optional *randBase64* random Base64 form string, and returns it into a string encoded in Base64 form.

The random value must be a valid Base64 String. You typically generate this value with the `security.RandomGenerator.CreateRandomString()` method.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Example

```
DEFINE password, digest STRING
...
LET digest =
    security.Digest.CreateDigestString(
       password,
       security.RandomGenerator.CreateRandomString(16) )
```

security.Digest.DoBase64Digest
Creates a digest of the buffered data and returns the result in base64 format.

Syntax

```
security.Digest.DoBase64Digest()
  RETURNING b64Digest STRING
```

1. *b64Digest* is the digest in base64

Usage

Processes the digest on all data previously added to the context and encodes it in base64.

After that call, the internal buffer is cleaned and ready to be populated again with new data to be digested.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

security.Digest.DoHexBinaryDigest
Creates a digest of the buffered data and returns the result in hexadecimal format.

Syntax

```
security.Digest.DoHexBinaryDigest()
  RETURNING hexBinDigest STRING
```

1. *hexBinDigest* is the hexadecimal digest.

Usage

Processes the digest on all data previously added to the context and returns it in hexadecimal.

After that call, the internal buffer is cleaned and ready to be populated again with new data to be digested.

In case of error, the method throws an exception and sets the STATUS variable. Depending on the error, a human-readable description of the problem is available in the SQLCA.SQLERRM register. See Error handling in GWS calls (STATUS) on page 2546.

Example
Computing a hash value of a string.

Program example ComputeHash.4gl :

```
IMPORT SECURITY

MAIN
```

```
  DEFINE result STRING

  IF num_args() != 2 THEN
    DISPLAY "Usage: ComputeHash <string> <hashcode>"
    DISPLAY "  string: the string to digest"
    DISPLAY "  hashcode: SHA1, SHA512, SHA384, SHA256, SHA224, MD5"
  ELSE
    LET result = ComputeHash(arg_val(1), arg_val(2))
    IF result IS NOT NULL THEN
      DISPLAY "Hash value is :",result
    ELSE
      DISPLAY "Error"
    END IF
  END IF

END MAIN

FUNCTION ComputeHash(toDigest, algo)

  DEFINE toDigest, algo, result STRING
  DEFINE dgst security.Digest

  TRY
    LET dgst = security.Digest.CreateDigest(algo)
    CALL dgst.AddStringData(toDigest)
    LET result = dgst.DoBase64Digest()
  CATCH
    DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
    EXIT PROGRAM(-1)
  END TRY

  RETURN result
END FUNCTION
```

Example execution:

```
$ fglrun ComputeHash "Hello World" SHA1
```

Hash value is :Ck1VqNd45QIvq3AZd8XYQLvEhtA=

File extensions

This page describes the file extensions used by the language.

Table 553: File extensions

Extension	Type	Description
`.4gl`	Text	Source module
`.42m`	Binary	Compiler p-code module
`.per`	Text	Form specification file
`.42f`	XML	Compiled form specification file
`.42s`	Binary	Localized strings compiled file
`.4st`	XML	Presentation styles resource file
`.4sm`	XML	Startmenu resource file

Extension	Type	Description
`.4tm`	XML	Topmenu resource file
`.4tb`	XML	Toolbar resource file
`.4ad`	XML	Action defaults resource file
`.sch`	Text	Database schema file - column types
`.42d`	Binary (development only)	Database schema file index (for .sch)
`.str`	Text	Localized strings source file
`.val`	Text	Database schema file - form field attributes
`.att`	Text	Database schema file - video attributes
`.42r`	Binary	Compiled program
`.42x`	Binary	Compiled p-code library
`.msg`	Text	Message definition source file
`.iem`	Binary	Compiled message definition file

Genero BDL errors

System error messages sorted by error number.

Table 554: Genero system error messages

Number	Description
-201	`A syntax error has occurred.` This general SQL error message indicates mistakes in the syntax of an SQL statement. Look for missing or extra punctuation; keywords misspelled, misused, or out of sequence, or a reserved word used as an identifier.
-204	`An illegal floating point number has been found in the statement.` A numeric constant that is punctuated like a floating-point number (with a decimal point and/or an exponent starting with e) is unacceptable. Possibly the exponent is larger than can be processed.
-206	`The specified table table-name is not in the database.` The database server cannot find a table or view specified in the statement. The table or view might have been renamed or dropped from the database.
-213	`Statement interrupted by user.` The database server received an interrupt signal from the user. The statement ended early. A program should roll back the current transaction and terminate gracefully.
-217	`Column column-name not found in any table in the query.` The column specified does not exist in the database tables used in this SQL statement.

Number	Description
-235	`Character column size is too big.` The SQL statement specifies a width for a character data type that is greater than 65,534 bytes. If you need a column of this size, use the TEXT data type, which allows unlimited lengths. Otherwise, inspect the statement for typographical errors.
-236	`Number of columns in INSERT does not match number of VALUES.` Each column that is named or implied in an INSERT statement must have a separate value expression. If the statement does not list specific columns, review the definition of the table for the number of columns and their data types. Also check that the list of expressions in the VALUES clause has no extra or missing comma that might result in an incorrect number of values. Be especially careful of long character strings and expressions with parentheses.
-239	`Could not insert new row - duplicate value in a UNIQUE INDEX column.` The row that is being inserted (or being updated to have a new primary key) contains a duplicate value of some row that already exists, in a column or columns that are constrained to have unique values.
-244	`Could not do a physical-order read to fetch next row.` The database server cannot read the data block for this SQL client program. The database server returns this error when a record is locked by another process, and the lock timeout defined by the current program has expired. Consider using the `SET LOCK MODE TO WAIT` instruction to define a lock timeout. By default, with most databases, this timeout is zero and error -244 is returned immediately when a lock conflict occurs. If all programs do short transactions (holding locks for a short period of time), it is usually safe to define a lock timeout of 5 to 10 seconds to avoid this SQL error.
-250	`Cannot read record from file for update.` The database server cannot get a row of a table prior to update.
-251	`ORDER BY or GROUP BY column number is too big.` The ORDER BY or GROUP BY clause uses column-sequence numbers, and at least one of them is larger than the count of columns in the select list.
-253	`Cannot read record from file for update.` The database server cannot get a row of a table prior to update.
-254	`Too many or too few host variables given.` The number of host variables that you named in the INTO clause of this statement does not match the number of columns that you referenced in the SQL statement.
-255	`Not in transaction.`

Number	Description
	The database server cannot execute this COMMIT WORK or ROLLBACK WORK statement because no BEGIN WORK was executed to start a transaction. Because no transaction was started, you cannot end one.
-256	`Transaction not available.` The database server does not support transactions.
-257	`System limit on maximum number of statements exceeded, maximum is count.` The database server can handle only a fixed number of prepared SQL statements for each user. This limit includes statements that were prepared with the PREPARE statement and cursors that were declared with the DECLARE statement.
-259	`Cursor not open.` The current statement refers to a cursor that has not been opened. Review the logic of the program to see how it failed to execute the OPEN statement before it reached this point.
-263	`Could not lock row for UPDATE.` This statement, probably a FETCH statement that names a cursor declared FOR UPDATE, failed because the row it should have fetched could not be locked.
-266	`There is no current row for UPDATE/DELETE cursor.` The current statement uses the WHERE CURRENT OF cursor-name clause, but that cursor has not yet been associated with a current row. Either no FETCH statement has been executed since it was opened, or the most recent fetch resulted in an error so that no row was returned. Revise the logic of the program so that it always successfully fetches a row before it executes this statement.
-268	`Unique constraint constraint-name violated.` The current statement uses the WHERE CURRENT OF cursor-name clause, but that cursor has not yet been associated with a current row. Either no FETCH statement has been executed since it was opened, or the most recent fetch resulted in an error so that no row was returned. Revise the logic of the program so that it always successfully fetches a row before it executes this statement.
-272	`No SELECT permission for table/column.` The person who created this table has not granted SELECT privilege to your account name or to the public for the table or the column. The owner of the table or the DBA must grant this privilege before you can select data from the table or column.
-273	`No UPDATE permission for table/column.` The person who created this table has not granted UPDATE privilege to your account name or to the public for the table or the column. The owner of the table or the DBA must grant this privilege before you can update a row in this table or update the column.
-274	`No DELETE permission for table.`

Number	Description
	The person who created this table has not granted DELETE privilege to your account name or to the public. The owner of the table or the DBA must grant this privilege before you can delete a row in this table.
-275	`The Insert privilege is required for this operation.` The Insert access privilege on this table or column is not currently held by your account name, nor by the PUBLIC group, nor by your current role. The owner of the table or the DBA must grant the Insert privilege before you can insert a row into this table.
-280	`A quoted string exceeds 256 bytes.` A character literal in this statement exceeds the maximum length. Check the punctuation and length of all quoted strings in the statement. Possibly two missing quotes make a long string out of two short ones. You must revise the statement to use a shorter character string.
-282	`Found a quote for which there is no matching quote.` Inspect the current statement, examining the punctuation of all quoted strings.
-284	`A subquery has returned not exactly one row.` A subquery that is used in an expression in the place of a literal value must return only a single row and a single column. In this statement, a subquery has returned more than one row, and the database server cannot choose which returned value to use in the expression. You can ensure that a subquery will always return a single row. Use a WHERE clause that tests for equality on a column that has a unique index. Or select only an aggregate function. Review the subqueries, and check that they can return only a single row. This error can also occur when you use a singleton SELECT statement to retrieve multiple rows. You must use the DECLARE/OPEN/FETCH series of statements or the EXECUTE INTO statement to retrieve multiple rows.
-285	`Invalid cursor received by sqlexec.` The cursor that this statement uses has not been properly declared or prepared, or the FREE statement has released it, or an automatic re-prepare has been attempted while opening the cursor but that operation failed, leaving the cursor unavailable. Review the program logic to ensure that the cursor has been declared. If it has, and if the DECLARE statement refers to a statement identifier, check that the referenced statement has been prepared.
-290	`Cursor not declared with FOR UPDATE clause.` This statement attempts to update with a cursor that was not declared for update. To use a cursor with the UPDATE or DELETE statements, you must declare it FOR UPDATE. Review the program logic to make sure that this statement uses the intended cursor.
-294	`The column column-name must be in the GROUP BY list.` In a grouping SELECT, you must list every nonaggregate column in the GROUP BY clause to ensure that a well-defined value exists for each selected column in each grouped row. A column contains either a single aggregate value or a value unique to that group. If a selected column were neither an aggregate nor in the list, two or more values for that column might possibly exist in some group, and the database server

Number	Description
	could not choose which value to display. Revise the query to include either the column name or its positional number in the clause.
-307	`Illegal subscript.` The substring values (two numbers in square brackets) of a character variable are incorrect. The first is less than zero or greater than the length of the column, or the second is less than the first. Review all uses of square brackets in the statement to find the error. Possibly the size of a column has been altered and makes a substring fail that used to work.
-309	`ORDER BY column or expression must be in SELECT list.` An expression or column name is in the ORDER BY clause of this SELECT statement, but the expression or column name is not in the select list (the list of values that follows the word SELECT). This action is not supported when a UNIQUE or DISTINCT operator is being used in a query.
-316	`Index index-name already exists in database.` This statement tries to create an index with the name shown, but an index of that name already exists. Only one index of a given name can exist in a single database.
-324	`Ambiguous column column-name.` The column name appears in more than one of the tables that are listed in the FROM clause of this query. The database server needs to know which columns to use. Revise the statement so that this name is prefixed by the name of its table (table-name.column) wherever it appears in the query.
-329	`Database not found or no system permission.` The database you tried to connect to is not known by the db server. Check database client configuration settings and make sure that there is no spelling error in the name of the database
-330	`Cannot create or rename the database.` Possibly you tried to create a database with the same name as one that already exists or rename a database to a name that already exists; if so, choose a different name.
-349	`Database not selected yet.` The SQL statement cannot be executed because no current database exists. You must issue a DATABASE or CONNECT TO instruction before executing other SQL statements.
-350	`Index already exists on the column (or on the set of columns).` This CREATE INDEX statement cannot be executed because an index on the same column or combination of columns already exists. For a given collation order, at most two indexes can exist on any combination of columns, one ascending and one descending.
-354	`Incorrect database or cursor name format.`

Number	Description
	This statement contains the name of a database or a cursor in some invalid format. If the statement is part of a program, the name might have been passed in a host variable.
-360	`Cannot modify a table or view that is also used in subquery.` The UPDATE, INSERT, or DELETE statement uses data taken from the same table in a subquery. Because of the danger of entering an endless loop, this action is not allowed, except in the case of an uncorrelated subquery in the WHERE clause of the DELETE or UPDATE statement.
-363	`CURSOR not on SELECT statement.` The cursor named in this statement (probably an OPEN) has been associated with a prepared statement that is not a SELECT statement. Review the program logic, especially the DECLARE for the cursor, the statement id specified in it, and the PREPARE that set up that statement. If you intended to use a cursor with an INSERT statement, you can only do that when the INSERT statement is written as part of the DECLARE statement. If you intended to execute an SQL statement, do that directly with the EXECUTE statement, not indirectly through a cursor.
-366	`The scale exceeds the maximum precision specified.` A problem exists with the precision or scale of a DECIMAL or a MONEY data type usage, for example in a DEFINE statement. The first should be declared as DECIMAL(p) or DECIMAL(p,s) where p, the precision (total number of digits) is between 1 and 32, and s, the scale (number of digits to the right of the decimal point) is greater or equal to zero and not greater than p. The MONEY type follows the same rules. Review the DECIMAL or MONEY type definition, and make sure that the precision is in the range [1,32] and that the scale is in the range [0,precision].
-371	`Cannot create unique index on column with duplicate data.` This CREATE UNIQUE INDEX statement cannot be completed because the column (or columns) contains one or more duplicate rows.
-387	`No connect permission.` You cannot access the database that this statement requests because you have not been granted CONNECT privilege to it. Contact a person who has Database Administrator privilege to that database and ask to be granted CONNECT privileges to it.
-388	`No resource permission.` If you issued a CREATE TABLE, CREATE INDEX, or CREATE PROCEDURE statement, you cannot execute this statement because your account has not been granted the RESOURCE privilege for this database. You need the RESOURCE privilege to create permanent tables, indexes on permanent tables, and procedures.
-389	`No DBA permission.`

Number	Description
	This statement cannot be executed because you have not been granted DBA privilege for this database. Contact a person who has DBA privilege for the database and ask to be granted DBA privilege (or simply ask to have this statement executed for you).
-391	`Cannot insert a null into column column-name.` This statement tries to put a null value in the noted column. However, that column has been defined as NOT NULL. Roll back the current transaction. If this is a program, review the definition of the table, and change the program logic to not use null values for columns that cannot accept them.
-400	`Fetch attempted on unopen cursor.` This FETCH statement names a cursor that has never been opened or has been closed. Review the program logic, and check that it will open the cursor before this point and not accidentally close it. Unless a cursor is declared WITH HOLD, it is automatically closed by a COMMIT WORK or ROLLBACK WORK statement.
-404	`The cursor or statement is not available.` You used a statement that names a cursor that is was destroyed. Review the program logic and check that the cursor specified is declared and opened, but not freed, prior to reaching this statement.
-410	`Prepare statement failed or was not executed.` This EXECUTE statement refers to a statement id that has not been prepared. Either no PREPARE statement was done, or one was done but returned an error code. Review the program logic to ensure that a statement is prepared and the PREPARE return code is checked. A negative error code from PREPARE usually reflects an error in the statement being prepared.
-412	`Command pointer is NULL.` This statement (probably an EXECUTE or DECLARE) refers to a dynamic SQL statement that has never been prepared or that has been freed. Review the program logic to ensure that the statement has been prepared, the PREPARE did not return an error code, and the FREE statement has not been used to release the statement before this point.
-413	`Insert attempted on unopen cursor.` This INSERT statement names a cursor that has never been opened or that has been closed. Review the program logic, and check that it will open the cursor before this point and not accidentally close it. An insert cursor is automatically closed by a COMMIT WORK or ROLLBACK WORK statement.
-422	`Flush attempted on unopen cursor.` This FLUSH statement names a cursor that has never been opened or has been closed. Review the program logic to ensure that it will open the cursor before this point and not accidentally close it. An insert cursor is automatically closed by a COMMIT WORK or ROLLBACK WORK.
-450	`Illegal ESQL locator, or uninitialized blob variable in BDL.` An SQL statement is using a TEXT or BYTE variable that was not initialized with LOCATE IN FILE or MEMORY.

Number	Description
	LOCATE the TEXT or BYTE variable before using it in SQL statements.
-458	`Long transaction aborted.` The database server ran out of log space in which to record this transaction. A transaction that is not fully recorded cannot be rolled back. To preserve database integrity, the operating system ended the transaction and rolled it back automatically. All changes made since the start of the transaction have been removed. Terminate the application, and replan it so that it modifies fewer rows per transaction. Alternatively, contact the database server administrator to discuss increasing the number or the size of the logical logs. Be prepared to talk about the number of rows being updated or inserted and the size of each row.
-481	`Invalid statement name or statement was not prepared.` The statement has not been prepared, or the format of the statement name is not valid. A valid statement name does not exceed the maximum length, begins with a letter or underscore, does not contain any blanks or nonalphanumeric characters except underscores.
-482	`Invalid operation on a non-SCROLL cursor.` You cannot issue a FETCH PRIOR, FETCH FIRST, FETCH LAST, FETCH CURRENT, FETCH RELATIVE n, or FETCH ABSOLUTE n statement with a non-scroll cursor. To do so, you must first declare the cursor as a scroll cursor.
-507	`Cursor cursor-name not found.` The cursor that is named in the WHERE CURRENT OF clause in this UPDATE or DELETE statement does not exist. Review the spelling of the name. If it is as you intended, check the DECLARE statement to ensure that it has been executed. Also make sure that the cursor has not been freed with the FREE statement or during a failed automatic re-prepare attempt.
-513	`Statement not available with this database server.` The SQL statement used by the program is not valid for the target database server. Review the code, the SQL instruction cannot be used.
-517	`The total size of the index is too large or too many parts in index.` All database servers have limits on the number of columns that can be included in an index and on the total number of bytes in a key (the sum of the widths of the columns). This CREATE INDEX statement would exceed that limit for this database server.
-522	`Table table-name not selected in query.` The table name used in an expression (for example, in the WHERE clause) has not been listed in the clause defininig the tables to be used in the query (typically the FROM clause in SELECT statements).
-526	`Updates are not allowed on a scroll cursor.` For a DECLARE statement, the clause FOR UPDATE is not allowed in conjunction with the SCROLL keyword.

Number	Description
-530	`Check constraint constraint-name failed.` The check constraint placed on the table column was violated.
-535	`Already in transaction.` This BEGIN WORK statement is redundant; a transaction is already in progress. If this is a program, review its logic to make sure it has not accidentally failed to end the previous transaction.
-551	`The constraint contains too many columns.` The total number of columns listed in a UNIQUE, PRIMARY KEY, or FOREIGN KEY clause is limited. The limit depends on the database server in use.
-674	`Routine routine-name cannot be resolved.` You called a routine that does not exist in the database, you do not have permission to execute the routine, or you called the routine with too few or too many arguments.
-681	`Column specified more than once in the INSERT list.` The error occurs if the user specifies a column name more than once in the INSERT column list.
-691	`Missing key in referenced table for referential constraint constraint-name.` A referential constraint has been violated. This condition usually occurs when you are trying to insert a value into or update the value of a column that is part of a referential constraint. The value you are trying to enter does not exist in the referenced (parent-key) column.
-743	`Object object_name already exists in database.` You are trying to define an object that already exists in the database.
-768	`Internal error in routine routine-name.` If this internal error recurs, note all circumstances and contact your technical support.
-805	`Cannot open file for load.` The input file that is specified in this LOAD statement could not be opened. Check the statement. Possibly a more complete pathname is needed, the file does not exist, or your account does not have read permission for the file or a directory in which it resides.
-806	`Cannot open file for unload.` The output file that is specified in this UNLOAD statement could not be opened. Check the statement. Possibly a more complete pathname is needed; the file exists, but your account does not have write permission for it; or the disk is full.
-809	`SQL Syntax error has occurred.` The INSERT statement in this LOAD/UNLOAD statement has invalid syntax.

Number	Description
	Review it for punctuation and use of keywords.
-846	`Number of values in load file is not equal to number of columns.` The LOAD processor counts the delimiters in the first line of the file to determine the number of values in the load file. One delimiter must exist for each column in the table, or for each column in the list of columns if one is specified. Check that you specified the file that you intended and that it uses the correct delimiter character. An empty line in the text can also cause this error. If the LOAD statement does not specify a delimiter, verify that the default delimiter matches the delimiter that is used in the file. If you are in doubt about the default delimiter, specify the delimiter in the LOAD statement.
-930	`Cannot connect to database server servername.` The application is trying to access the database server but failed.
-942	`Transaction commit failed - transaction will be rolled back.` This error can occur at transaction-commit time if the database server could not commit the transaction.
-1102	`Field name not found in form.` A field name listed in an INPUT, INPUT ARRAY, CONSTRUCT, SCROLL or DISPLAY statement does not appear in the form specification of the screen form that is currently displayed. Review the program logic to ensure that the intended window is current, the intended form is displayed in it, and all the field names in the statement are spelled correctly.
-1107	`Field subscript out of bounds.` The subscript of a screen array in an INPUT, DISPLAY, or CONSTRUCT statement is either less than 1 or greater than the number of fields in the array. Review the program source in conjunction with the form specification to see where the error lies.
-1108	`Record name not in form.` The screen record that is named in an INPUT ARRAY or DISPLAY ARRAY statement does not appear in the screen form that is now displayed. Review the program source in conjunction with the form specification to see if the screen record names match.
-1109	`List and record field counts differ.` The number of program variables does not agree with the number of screen fields in a CONSTRUCT, INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY statement. Review the statement in conjunction with the form specification to see where the error lies. Common problems include a change in the definition of a screen record that is not reflected in every statement that uses the record, and a change in a program record that is not reflected in the form design.
-1110	`Form file (file-name) not found.`

Number	Description
	The form file that is specified in an OPEN FORM or OPEN WINDOW WITH FORM statement was not found. Inspect the form name used in the statement. It should not include the form file suffix. If the form is not in the current directory, verify that FGLRESOURCEPATH / DBPATH environment variables contain the path to the form file.
-1112	`A form is incompatible with the current BDL version. Rebuild your form.` The form file that is specified in an OPEN FORM statement is not acceptable. Possibly it was corrupted in some way, or it was compiled with a version of the Form Compiler that is not compatible with the version of the BDL compiler that compiled this program. Use a current version of the Form Compiler to recompile the form specification.
-1114	`No form has been displayed.` The current statement requires the use of a screen form. For example, DISPLAY...TO or an INPUT statement must use the fields of a form. However, the DISPLAY FORM statement has not been executed since the current window was opened. Review the program logic to ensure that it opens and displays a form before it tries to use a form.
-1119	`NEXT FIELD name not found in form.` This statement (INPUT or INPUT ARRAY) contains a NEXT FIELD clause that names a field that is not defined in the form. Review the form and program logic. Perhaps the form has been changed, but the program has not.
-1129	`Field ( field-name ) in BEFORE/AFTER clause not found in form.` This statement includes a BEFORE FIELD clause or an AFTER FIELD clause that names a field that is not defined in the form that is currently displayed. Review the program to ensure that the intended form was displayed, and review this statement against the form specification to ensure that existing fields are named.
-1133	`The NEXT OPTION name is not in the menu.` This MENU statement contains a NEXT OPTION clause that names a menu-option that is not defined in the statement. The string that follows NEXT OPTION must be identical to one that follows a COMMAND clause in the same MENU statement. Review the statement to ensure that these clauses agree with each other.
-1140	`NEXT OPTION is a hidden option.` The option that is named in this NEXT OPTION statement has previously been hidden with the HIDE OPTION statement. Because it is not visible to the user, it cannot be highlighted as the next choice. Use the SHOW OPTION statement to unhide the menu option.
-1141	`Cannot close window with active INPUT, DISPLAY ARRAY, or MENU statement.`

Number	Description
	This CLOSE WINDOW statement cannot be executed because an input operation is still active in that window. The CLOSE WINDOW statement must have been contained in, or called from within, the input statement itself. Review the program logic, and revise it so that the statement completes before the window is closed.
-1143	`Window is already open.` This OPEN WINDOW statement names a window that is already open. Review the program logic, and see whether it should contain a CLOSE WINDOW statement, or whether it should simply use a CURRENT WINDOW statement to bring the open window to the top.
-1146	`PROMPT message is too long to fit in the window.` Although BDL truncates the output of MESSAGE and COMMENT to fit the window dimensions, it does not do so for PROMPT and the user's response. Reduce the length of the prompt string, or make the window larger. You could display most of the prompting text with DISPLAY and then prompt with a single space or colon.
-1150	`Window is too small to display this menu.` The window must be at least two rows tall, and it must be wide enough to display the menu title, the longest option name, two sets of three-dot ellipses, and six spaces. Revise the program to make the window larger or to give the menu a shorter name and shorter options. Review the OPEN WINDOW statement for the current window in conjunction with this MENU statement.
-1168	`Command does not appear in the menu.` The SHOW OPTION, HIDE OPTION, or NEXT OPTION statement cannot refer to an option (command) that does not exist. Check the spelling of the name of the option.
-1170	`The type of your terminal is unknown to the system.` Check the setting of your TERM environment variable and the setting of your TERMCAP or TERMINFO environment variable. Check with your system administrator if you need help with this action.
-1202	`An attempt was made to divide by zero.` Zero cannot be a divisor. Check that the divisor is not zero. In some cases, this error arises because the divisor is a character value that does not convert properly to numeric.
-1204	`Invalid year in date.` The year in a DATE value or literal is invalid. For example, the number 0000 is not acceptable as the year. Check the value of year.

Number	Description
-1205	`Invalid month in date.` The month in a DATE value or literal must be a one- or two-digit number from 1 to 12. Check the value of month.
-1206	`Invalid day in date.` The day number in a DATE value or literal must a one- or two-digit number from 1 to 28 (or 29 in a leap year), 30, or 31, depending on the month that accompanies it. Check the value of day.
-1210	`Date could not be converted to month/day/year format.` The DATE type is compatible with the INTEGER type, but not all integer values are valid dates. The range of valid integer values for dates is from -693,594 to +2,958,464. Numbers that are outside this range have no representation as dates. Check the value of the number used to assign the date variable.
-1212	`Date conversion format must contain a month, day, and year component.` When a date value is converted between internal binary format and display or entry format, a pattern directs the conversion. When conversion is done automatically, the pattern comes from the environment variable DBDATE. When it is done with an explicit call to the rfmtdate(), rdefmtdate(), or USING functions, a pattern string is passed as a parameter. In any case, the pattern string (the format of the message) must include letters that show the location of the three parts of the date: 2 or 3 letters d; 2 or 3 letters m; and either 2 or 4 letters y. Check the pattern string and the value of DBDATE.
-1213	`A character to numeric conversion process failed.` A character value is being converted to numeric form for storage in a numeric column or variable. However, the character string cannot be interpreted as a number. Check the character string. It must not contain characters other than white space, digits, a sign, a decimal, or the letter e. Verify the parts are in the right order. If you are using NLS, the decimal character or thousands separator might be wrong for your locale.
-1214	`Value too large to fit in a SMALLINT.` The SMALLINT data type can accept numbers with a value range from -32,767 to +32,767. To store numbers that are outside this range, redefine the column or variable to use INTEGER or DECIMAL type.
-1215	`Value too large to fit in an INTEGER.` The INTEGER data type can accept numbers with a value range from -2,147,483,647 to +2,147,483,647. Check the other data types available, such as DECIMAL.
-1218	`String to date conversion error.`

Number	Description
	The data value does not properly represent a date: either it has non-digits where digits are expected, an unexpected delimiter, or numbers that are too large or are inconsistent. Check the value being converted.
-1222	`Value will not fit in a SMALLFLOAT.` A statement tries to assign a value that exceeds the limits of the SMALLFLOAT data type. Review the code and consider using a FLOAT or DECIMAL type.
-1223	`Value will not fit in a FLOAT.` A statement tries to assign a value that exceeds the limits of the FLOAT data type. Review the code and consider using a DECIMAL type.
-1226	`Decimal or money value exceeds maximum precision.` The data value has more digits to the left of the decimal point than the declaration of the variable allows. Revise the program to define the variable with an appropriate precision.
-1260	`It is not possible to convert between the specified types.` Data conversion does not make sense, or is not supported. Possibly you referenced the wrong variable or column. Check that you have specified the data types that you intended and that literal representations of data values are correctly formatted.
-1261	`Too many digits in the first field of datetime or interval.` The first field of a DATETIME literal must contain 1 or 2 digits (if it is not a YEAR) or else 2 or 4 digits (if it is a YEAR). The first field of an INTERVAL literal represents a count of units and can have up to 9 digits, depending on the precision that is specified in its qualifier. Review the DATETIME and INTERVAL literals in this statement, and correct them.
-1262	`Non-numeric character in datetime or interval.` A DATETIME or INTERVAL literal can contain only decimal digits and the allowed delimiters: the hyphen between year, month, and day numbers; the space between day and hour; the colon between hour, minute, and second; and the decimal point between second and fraction. Any other characters, or these characters in the wrong order, produce an error. Check the value of the literal.
-1263	`A field in a datetime or interval is out of range.` At least one of the fields in a datetime or interval is incorrect. Inspect the DATE, DATETIME, and INTERVAL literals in this statement. In a DATE or DATETIME literal, the year might be zero, the month might be other than 1 to 12, or the day might be other than 1 to 31 or inappropriate for the month. Also in a DATETIME

Number	Description
	literal, the hour might be other than 0 to 23, the minute or second might be other than 0 to 59, or the fraction might have too many digits for the specified precision.
-1264	`Extra characters at the end of a datetime or interval.` Only spaces can follow a DATETIME or INTERVAL literal. Inspect this statement for missing or incorrect punctuation.
-1265	`Overflow occurred on a datetime or interval operation.` An arithmetic operation involving a DATETIME and/or INTERVAL produced a result that cannot fit in the target variable. Check if the data type can hold the result of the operation. For example, extend the INTERVAL precision by using YEAR(9) or DAY(9).
-1266	`Intervals or datetimes are incompatible for the operation.` An arithmetic operation mixes DATETIME and/or INTERVAL values that do not match. Check the data types of the variable used in the operation.
-1267	`The result of a datetime computation is out of range.` In this statement, a DATETIME computation produced a value that cannot be stored. This situation can occur, for example, if a large interval is added to a DATETIME value. This error can also occur if the resultant date does not exist, such as Feb 29, 1999. Review the expressions in the statement and see if you can change the sequence of operations to avoid the overflow.
-1268	`Invalid datetime or interval qualifier.` This statement contains a DATETIME or INTERVAL qualifier that is not acceptable. These qualifiers can contain only the words YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, FRACTION, and TO. A number from 1 to 5 in parentheses can follow FRACTION. Inspect the statement for missing punctuation and misspelled words. A common error is adding an s, as in MINUTES.
-1279	`Value exceeds string column length.` You attempted to insert into a CHAR, NCHAR, VARCHAR, NVARCHAR or LVARCHAR column using a string host variable, but the string is too long.
-1284	`Value will not fit in a BIGINT or INT8.` The BIGINT data type can accept numbers with a value range from -9223372036854775807 to +9223372036854775807. To store numbers that are outside this range, redefine the column or variable to use the DECIMAL type.
-1301	`This value is not among the valid possibilities.` A list or range of acceptable values has been established for this column in the form-specification file. You must enter a value within the acceptable range.

Number	Description
-1302	`The two entries were not the same -- please try again.` To guard against typographical errors, this field has been designated VERIFY in the form-specification file. You must enter the value in this field twice, identically. Carefully reenter the data. Alternatively, you can cancel the form entry with the Interrupt key.
-1303	`You cannot use this editing feature because a picture exists.` This field is defined in the form-specification file with a PICTURE attribute to specify its format. You cannot use certain editing keys (for example, CTRL-A, CTRL-D, and CTRL-X) while you are editing such a field. Use only printable characters and backspace to enter the value.
-1304	`Error in field.` You entered a value in this field that cannot be stored in the program variable that is meant to receive it. Possibly you entered a decimal number when the application provided only an integer variable, or you entered a character string that is longer than the application expected.
-1305	`This field requires an entered value.` The cursor is in a form field that has been designated REQUIRED. You must enter some value before the cursor can move to another field. To enter a null value, type any printable character and then backspace. Alternatively, you can cancel the form entry with the Interrupt key.
-1306	`Please type again for verification.` The cursor is in a form field that has been designated VERIFY. This procedure helps to ensure that no typographical errors occur during data entry. You must enter the value twice, identically, before the cursor can move to another field. Alternatively, you can cancel the form entry with the Interrupt key.
-1307	`Cannot insert another row - the input array is full.` You are entering data into an array of records that is represented in the program by a static array of program variables. That array is now full; no place is available to store another record. Press the ACCEPT key to process the records that you have entered.
-1308	`Cannot delete row - it has no data.` You try to delete a row in an empty row. Nothing was deleted.
-1309	`There are no more rows in the direction you are going.` You are attempting to scroll an array of records farther than it can go, either scrolling up at the top or scrolling down at the bottom of the array. Further attempts will have the same result.
-1312	`FORMS statement error number error-num.`

Number	Description
	An error occurred in the form at runtime. Edit your source file: go to the specified line, correct the error, and recompile the file.
-1313	`SQL statement error number error-num.` The current SQL statement returned this error code number.
-1314	`Program stopped at 'filename', line number line-number.` At runtime an error occurred in the specified file at the specified line. No .err file is generated. Edit your source file, go to the specified line, correct the error, and recompile the file.
-1318	`A parameter count mismatch has occurred between the calling function and the called function.` Either too many or too few parameters were given in the call to the function. The call is probably in a different source module from the called functions. Inspect the definition of the function, and check all places where it is called to ensure that they use the number of parameters that it declares.
-1320	`A function has not returned the correct number of values expected.` A function that returns several variables has not returned the correct number of parameters. Check your source code and recompile.
-1321	`A validation error has occurred as a result of the VALIDATE command.` The VALIDATE LIKE statement tests the current value of variables against rules that are stored in the syscolval table. It has detected a mismatch. Ordinarily, the program would use the WHENEVER statement to trap this error and display or correct the erroneous values. Inspect the VALIDATE statement to see which variables were being tested and find out why they were wrong.
-1322	`A report output file cannot be opened: description` The file that the REPORT TO statement specifies cannot be opened. See the description for more details. Check that your account has permission to write such a file, that the disk is not full, and that you have not exceeded some limit on the number of open files.
-1323	`A report output pipe cannot be opened.` The pipe that the REPORT TO PIPE statement specifies could not be started. Check that all programs that are named in it exist and are accessible from your execution path. Also look for operating-system messages that might give more specific errors.
-1324	`A report output file cannot be written to.`

Number	Description
	The file that the REPORT TO statement specifies was opened, but an error occurred while writing to it. Possibly the disk is full. Look for operating- system messages that might give more information.
-1326	`An array variable has been referenced outside of its specified dimensions.` The subscript expression for an array has produced a number that is either less than one or greater than the number of elements in the array. Review the program logic that leads up to this statement to determine how the error was made.
-1327	`An insert statement could not be prepared for inserting rows into a temporary table used for a report.` Within the report function, BDL generated an SQL statement to save rows into a temporary table. The dynamic preparation of the statement (see the reference material on the PREPARE statement) produced an error. Probably the database tables are not defined now, at execution time, as they were when the program was compiled. Either the database has been changed, or the program has selected a different database than the one that was current during compilation. Possibly the database administrator has revoked SELECT privilege from you for one or more of the tables that the report uses. Look for other error messages that might give more details.
-1328	`A temporary table needed for a report could not be created in the selected database.` Within the report definition, BDL generated an SQL statement to save rows into a temporary table, but the temporary table could not be created. You must have permission to create tables in the selected database, and there must be sufficient disk space left in the database. You may already have a table in your current database with the same name as the temporary table that the report definition is attempting to create as a sorting table; the sorting table is named "t_ reportname ". Another possible cause with some database servers is that you have exceeded an operating-system limit on open files.
-1329	`A database index could not be created for a temporary database table needed for a report.` Within the report definition, BDL generated SQL statements to save rows into a temporary table. However, an index could not be created on the temporary table. Probably an index with the same name already exists in the database. (The sorting index is named "i_reportname"; for example, "i_order_rpt".) Possibly no disk space is available in the file system or dbspace. Another possibility with some database servers is that you have exceeded an operating-system limit on open files.
-1330	`A row could not be inserted into a temporary report table.` Within the report definition, BDL generated SQL statements that would save rows into a temporary table. However, an error occurred while rows were being inserted.

Number	Description
	Probably no disk space is left in the database. Look for other error messages that might give more details.
-1331	`A row could not be fetched from a temporary report table.` Within the report definition, BDL generated SQL statements to select rows from a temporary table. The table was built successfully but now an error occurred while rows were being retrieved from it. Almost the only possible cause is a hardware failure or an error in the database server. Check for operating-system messages that might give more details.
-1332	`A character variable has referenced subscripts that are out of range.` In the current statement, a variable that is used in taking a substring of a character value contains a number less than one or a number greater than the size of the variable, or the first substring expression is larger than the second. Review the program logic that leads up to this statement to find the cause of the error.
-1335	`A report is accepting output or being finished before it has been started.` The program executed an OUTPUT TO REPORT or FINISH REPORT statement before it executed a START REPORT. Review the program logic that leads up to this statement to find the cause of the error.
-1337	`The variable variable-name has been redefined with a different type or length, definition in module-name-1.4gl, redefinition in module-name-2.4gl.` The variable that is shown is defined in the GLOBALS section of two or more modules, but it is defined differently in some modules than in others. Possibly modules were compiled at different times, with some change to the common GLOBALS file between. Possibly the variable is declared as a module variable in some module that does not include the GLOBALS file.
-1338	`The function 'function-name' has not been defined in any module in the program.` The named function is called from at least one module of the program, but it is defined in none. Verify that the module containing the function is a part of the program, and that the function name is correctly spelled.
-1340	`The error log has not been started.` The program called the errorlog() function without first calling the startlog() function. Review the program logic to find out the cause of this error.
-1349	`Character to numeric conversion error.` A character value is being converted to numeric form for storage in a numeric column or variable. However, the character string cannot be interpreted as a number. It

Number	Description
	contains some characters other than white space, digits, a sign, a decimal, or the letter e, or else the parts are in the wrong order so that the number cannot be deciphered.
-1353	`Use '!' to edit TEXT and BYTE fields.` This is a normal message text used outside an error context.
-1355	`Cannot build temporary file.` A TEXT or BYTE variable has been located in a temporary file using the LOCATE statement. The current statement assigns a value into that variable, so BDL attempted to create the temporary file, but an error occurred. Possibly no disk space is available, or your account does not have permission to create a temporary file. Look for operating-system error messages that might give more information.
-1359	`Read error on blob file 'file-name'.` The operating system signaled an error during output to a temporary file in which a TEXT or BYTE variable was being saved. Possibly the disk is full, or a hardware failure occurred. For more information, look for operating-system messages.
-1360	`No PROGRAM= clause for this field.` No external program has been designated for this field using the PROGRAM attribute in the form-specification file (For Text User Interface mode on ASCII terminals only)
-1373	`The field 'field-name' is not in the list of fields in the CONSTRUCT/INPUT statement.` The built-in function get_fldbuf() or field_touched() has been called with the field name shown. However, input from that field was not requested in this CONSTRUCT or INPUT statement. As a result, the function cannot return any useful value. Review all uses of these functions, and compare them to the list of fields at the beginning of the statement.
-1374	`SQL character truncation or transaction warning.` The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement involved is a DATABASE statement, the condition is that the database that was just opened uses a transaction log. On any other statement, the condition is that a character value from the database had to be truncated to fit in its destination.
-1375	`SQL NULL value in aggregate or mode ANSI database warning.` The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement that is involved is a DATABASE statement, the condition is that the database that was just opened is ANSI compliant. On any other statement, the condition is that a null value has been used in the computation of an aggregate value.
-1376	`SQL, database server, or program variable mismatch warning.` The program set WHENEVER WARNING STOP, and a warning condition arose. If the statement that is involved is a DATABASE or CREATE DATABASE statement, the condition is that the database server opened the database. On any other statement,

Number	Description
	the condition is that a SELECT statement returned more values than there were program variables to contain them.
-1377	`SQL float-to-decimal conversion warning.` The program set WHENEVER WARNING STOP, and a warning condition arose. The condition is that in the database that was just opened, the database server will use the DECIMAL data type for FLOAT values.
-1378	`SQL non-ANSI extension warning.` A database operation was performed that is not part of ANSI SQL, although the current database is ANSI compliant. This message is informational only.
-1396	`A report PRINT FILE source file cannot be opened for reading.` The file that is named in a PRINT FILE statement cannot be opened. Review the file name. If it is not in the current directory, you must specify the full path. If the file exists, make sure your account has permissions to read it.
-2017	`The character data value does not convert correctly to the field type.` You have entered a character value (a quoted string) into a field that has a different data type (for example INTEGER). However, the characters that you entered cannot be converted to the type of the field. Re-enter the data.
-2024	`There is already a record 'record-name' specified.` A screen record is automatically defined for each table that is used in the ATTRIBUTES section to define a field. If you define a record with the name of a table, it is seen as a duplicate. Check that the record-name of every screen record and screen array is unique in the form specification.
-2028	`The symbol 'symbol-name' does not represent a table prefix used in this form.` In a SCREEN RECORD statement, each component must be introduced by the name of the table as defined in the TABLES section or by the word FORMONLY. Review the spelling of the indicated name against the TABLES section, and check the punctuation of the rest of the statement.
-2029	`Screen record array 'record-name' has different component sizes.` The screen record array name has component sizes which either differ from the specified dimension of the array or differ among themselves. This error message appears when one or more of the columns appear a different number of times. The dimension of the screen array is written in square brackets that follow its name. Verify that the dimensions of the screen array match the screen fields.
-2039	`The attributes AUTONEXT, DEFAULT, INCLUDE, VERIFY, RIGHT and ZEROFILL are not supported for BLOB fields.`

Number	Description
	Columns of the data type specified cannot be used in the ways that these attributes imply. Check that the table and column names are as you intended, and verify the current definition of the table in the database that the DATABASE statement names.
-2041	`The form 'form-name' cannot be opened.` The form filename cannot be opened. This is probably because it does not exist, or the user does not have read permission. Check the spelling of filename. Check that the form file exists in your current directory. If it is in another directory, check that the correct pathname has been provided. On a UNIX™ system, if these things are correct, verify that your account has read permission on the file.
-2045	`The conditional attributes of a field cannot depend on the values of other fields.` The boolean expression in a WHERE clause of a COLOR attribute can use only the name of that field and constants. Revise this attribute, and recompile the form.
-2100	`Field 'field-name' has validation string error, String = string.` One of the formatting or validation strings that is stored in the syscolval or syscolatt tables is improperly coded. The string is shown as is the field to which it applies. Update the string in the tables.
-2810	`The name 'database-name' is not an existing database name.` This name, which was found in the DATABASE statement at the start of the form specification, is not a database that can be found. Check the spelling of the database name and the database entries in the fglprofile file.
-2820	`The label name between brackets is incorrectly given or the label is missing.` In the layout section of a form specification, the brackets should contain a simple name. Instead, they contain spaces or an invalid name. Check the layout section of the form for invalid form item labels.
-2830	`A left square bracket has been found on this line, with no right square bracket to match it.` Every left square bracket field delimiter must have a right square bracket delimiter on the same line. Review the form definition file to make sure all fields are properly marked.
-2840	`The field label 'label-name' was not defined in the form.` The indicated name appears at the left of this ATTRIBUTES statement, but it does not appear within brackets in the SCREEN section. Review the field tags that have been defined to see why this one was omitted.

Number	Description
-2843	`The column 'column-name' does not appear in the form specification.` A name in this ATTRIBUTES statement should have been defined previously in the form specification. Check that all names in the statement are spelled correctly and defined properly.
-2846	`The field 'field-name' is not a member of the table 'table-name'.` Something in this statement suggests that the name shown is part of this table, but that is not true in the current database. Review the spelling of the two names. If they are as you intended, check that the correct database is in use and that the table has not been altered.
-2859	`The column 'column-name' is a member of more then one table -- you must specify the table name.` Two or more tables that are named in the TABLES section have columns with the name shown. You must make clear which table you mean. To do this, write the table name as a prefix of the column name, as table.column, wherever this name is used in the form specification.
-2860	`There is a column/value type mismatch for 'column-name'.` This statement assigns a value to the field with the DEFAULT clause or uses its value with the INCLUDE clause, but it does so with data that does not agree with the data type of the field. Review the data type of the field (which comes from the column with which it is associated), and make sure that only compatible values are assigned.
-2862	`The table 'table-name' cannot be found in the database.` The indicated table does not exist in the database that is named in the form. Check the spelling of the table name and database name. If they are as you intended, either you are not using the version of the database that you expected, or the database has been changed.
-2863	`The column 'column-name' does not exist among the specified tables.` The tables that are specified in the TABLES section of the form exist, but column-name, which is named in the ATTRIBUTES section, does not. Check its spelling against the actual table. Possibly the table was altered, or the column was renamed.
-2864	`The table 'table-name' is not among the specified tables.` The indicated table is used in this statement but is not defined in the TABLES section of the form specification. Check its spelling; if it is as you intended, add the table in the TABLES section.

Number	Description
-2865	`The column 'column-name' does not exist in the table 'table-name'.` Something in this statement implies that the column shown is part of the indicated table (most likely the statement refers to table-name.column-name). However, it is not defined in that table. Check the spelling of both names. If they are as you intended, then make sure that the database schema (.sch) is up to date; possibly the table has been altered or the column renamed, and thus needs a new db schema extraction with the fgldbsch tool.
-2892	`The column 'column-name' appears more then once. If you wish a column to be duplicated in a form, use the same display field label.` The same column name is listed in the ATTRIBUTES section more than once. The expected way to display the same column in two or more places is to put two or more fields in the screen layout, each with the same tag-name. Then put a single statement in the ATTRIBUTES section to associate that tag-name with the column name. The current column value will be duplicated in all fields. If you intended to display different columns that happen to have the same column-names, prefix each column with its table-name.
-2893	`The display field label 'label-name' appears more than once in this form, but the lengths are different.` You can put multiple copies of a field in the screen layout (all will display the same column), but all copies must be the same length. Review the form definition to make sure that, if you intended to have multiple copies of one field, all copies are the same.
-2975	`The display field label 'label-name' has not been used.` A field tag has been declared in the screen section of the form- specification file but is not defined in the attributes section. Check your form-specification file.
-2992	`The display label 'label-name' has already been used.` The forms compiler indicates that name has been defined twice. These names must be defined uniquely in the form specification. Review all uses of the name to see if one of them is incorrect.
-2997	`See error number error-num.` The database server returned an error that is shown. Look up the shown error in the database server documentation.
-4303	`A blob variable or cursor name expected .` The argument to the FREE statement must be the name of a cursor or prepared statement or, in BDL, the name of a variable with the BYTE or TEXT data type. Check the name used after the FREE keyword.

Number	Description
-4307	`The number of variables and/or constants in the display list does not match the number of form fields in the display destination.` There must be exactly as many items in the list of values to display as there are fields listed following the TO keyword in this statement. Review the statement.
-4308	`The number of input variables does not match the number of form fields in the screen input list.` Your INPUT statement must specify the same number of variables as it does fields. When checking this, keep in mind that when you refer to a record using an asterisk or THRU, it is the same as listing each record component individually.
-4309	`Printing cannot be done within a loop or CASE statement contained in report headers or trailers.` BDL needs to know how many lines of space will be devoted to page headers and trailers; otherwise, it does not know how many detail rows to allow on a page. Since it cannot predict how many times a loop will be executed, or which branch of a CASE will be execute, it forbids the use of PRINT in these contexts within FIRST PAGE HEADER, PAGE HEADER, and PAGE TRAILER sections. Re-arrange the code to place the PRINT statement where it will always be executed.
-4319	`The symbol 'symbol-name' has been defined more than once.` The variable that is shown has appeared in at least one other DEFINE statement before this one. Review your code. If this DEFINE is within a function or the MAIN section, the prior one is also. If this DEFINE is outside any function, the prior one is also outside any function; however, it might be within the file included by the GLOBALS statement.
-4320	`The symbol 'symbol-name' is not the name of a table in the specified database.` The named table does not appear in the database. Review the statement. The table name may be spelled wrong in the program, or the table might have been dropped or renamed since the last time the program was compiled.
-4322	`The symbol 'symbol-name' is not the name of a column in the specified database.` The preceding statement suggests that the named column is part of a certain table in the specified database. The table exists, but the column does not appear in it. Check the spelling of the column name. If it is spelled as you intended, then either the table has been altered, or the column renamed, or you are not accessing the database you expected.
-4323	`The variable 'variable-name' is too complex to be used in an assignment statement.`

Number	Description
	The named variable is a complex variable like a record or an array, which cannot be used in a LET statement. You must assign groups of components to groups of components using asterisk notation.
-4324	`The variable 'variable-name' is not a character type, and cannot be used to contain the result of concatenation.` This statement attempts to concatenate two or more character strings (using the comma as the concatenation operator) and assign the result to the named variable. Unfortunately, it is not a character variable, and automatic conversion from characters cannot be performed in this case. Assign the concatenated string to a character variable; then, if you want to treat the result as numeric, assign the string as a whole to a numeric variable.
-4325	`The source and destination records in this record assignment statement are not compatible in types and/or length.` This statement uses asterisk notation to assign all components of one record to the corresponding components of another. However, the components do not correspond. Note that BDL matches record components strictly by position, the first to the first, second to second, and so on; it does not match them by name. If the source and destination records do not have the same number and type of components, you will have to write a simple assignment statement for each component.
-4328	`The variable 'variable-name' is too complex to be used as the destination of a return from a function.` The named variable is too complex to be assigned directly in a `RETURNING` clause. Individual members of the complex variable must be returned separately.
-4333	`The function 'function-name' has already been called with a different number of parameters.` Earlier in the program, there is a call to this same function or event with a different number of parameters in the parameter list. At least one of these calls must be in error. Examine the FUNCTION statement for the named function to find out the correct number of parameters. Then examine all calls to it, and make sure that they are written correctly.
-4334	`The variable 'variable-name' in its current form is too complex to be used in this statement.` The variable has too many component parts. Only simple variables (those that have a single component) can be used in this statement. If variable-name is an array, you must provide a subscript to select just one element. If it is a record, you must choose just one of its components. (However, if this statement permits a list of variables, as in the INITIALIZE statement, you can use asterisk or THRU notation to convert a record name into a list of components)
-4335	`The symbol 'field-name' is not an element of the record 'record-name'.`

Number	Description
	The field name used in a record.field expression is not identified as a member of the record variable. Find the definition of the record (it may be in the GLOBALS file), verify the names of its fields, and correct the spelling of field-name .
-4336	`The parameter 'param-name' has not been defined within the function or report.` The name variable-name appears in the parameter list of the FUNCTION statement for this function. However, it does not appear in a DEFINE statement within the function. All parameters must be defined in their function before use. Review your code. Possibly you wrote a DEFINE statement but did not spell variable-name the same way in both places.
-4338	`The symbol 'symbol-name' has already been defined once as a parameter.` The name that is shown appears in the parameter list of the FUNCTION statement and in at least two DEFINE statements within the function body. Review your code. Only one appearance in a DEFINE statement is permitted.
-4340	`The variable 'variable-name' is too complex a type to be used in an expression.` In an expression, only simple variables (those that have a single component) can be used. If the variable indicated is an array, you must provide a subscript to select just one element. If it is a record or object, you must choose just one of its components.
-4341	`Aggregate functions are only allowed in reports and SELECT statements.` Aggregate functions such as SUM, AVG, and MAX can only appear in SQL statements and within certain statements that you use in the context of a report body. They are not supported in ordinary expressions in program statements. Review the code and check that the aggregate functions are in an SQL statement or in the correct blocks of the REPORT routine.
-4343	`Subscripting cannot be applied to the variable 'variable-name'.` You tried to use a [x,y] subscript expression with a variable that is neither a character data type or an array type. Check the variable data type and make sure it can be used with a subscript expression.
-4347	`The variable 'variable-name' is not a record. It cannot reference record elements.` In this statement variable-name appears followed by a dot, followed by another name. This is the way you would refer to a component of a record variable; however, variable-name is not defined as a record. Either you have written the name of the wrong variable, or else variable-name is not defined the way you intended.

Number	Description
-4353	`The type of this ORDER BY or GROUP item specified for the report is not valid for sorting.` A REPORT routine defines an ORDER BY or GROUP clause using a variable defined with a type such as TEXT and BYTE, that is too complex to be used in comparisons. As result, columns with such types cannot be used to sort or group rows. Review the report and sort or group rows by using items defined with simple data types.
-4356	`A PAGE HEADER has already been specified within this report.` Only one PAGE HEADER control block is allowed in a REPORT. Search for other PAGE HEADER sections and combine all statements in a unique control block.
-4357	`A PAGE TRAILER has already been specified within this report.` Only one PAGE TRAILER control block is allowed in a REPORT. Search for other PAGE TRAILER sections and combine all statements in a unique control block.
-4358	`A FIRST PAGE HEADER has already been specified within this report.` Only one FIRST PAGE TRAILER control block is allowed in a REPORT. Search for other FIRST PAGE TRAILER sections and combine all statements in a unique control block.
-4359	`An ON EVERY ROW clause has already been specified within this report.` Only one ON EVERY ROW control block is allowed in a REPORT. Search for other ON EVERY ROW sections and combine all statements in a unique control block.
-4360	`An ON LAST ROW clause has already been specified within this report.` Only one ON LAST ROW control block is allowed in a REPORT. Search for other ON LAST ROW sections and combine all statements in a unique control block.
-4361	`Group aggregates can occur only in AFTER GROUP clauses.` The aggregate functions that apply to a group of rows (GROUP COUNT/PERCENT/ SUM/AVG/MIN/MAX) can only be used at the point in the report when a complete group has been processed, namely, in the AFTER GROUP control block. Make sure that the AFTER GROUP block exists and was recognized. If you need the value of a group aggregate at another time (for instance, in a PAGE TRAILER control block), you can save it in a module variable with a LET statement in the AFTER GROUP block.

Number	Description
-4363	`The report cannot skip lines while in a loop within a header or trailer.` BDL needs to know how many lines of space will be devoted to the page header and trailer (otherwise it does not know how many detail rows to allow on the page). It cannot predict how many times a loop will be executed, so it has to forbid the use of SKIP statements in loops in the PAGE HEADER, PAGE TRAILER, and FIRST PAGE HEADER sections. Review the report header or trailer to avoid SKIP in loops.
-4369	`The symbol 'symbol-name' does not represent a defined variable.` The name shown appears where a variable would be expected, but it does not match any variable name in a DEFINE statement that applies to this context. Check the spelling of the name. If it is the name you intended, look back and find out why it has not yet been defined. Possibly the GLOBALS statement has been omitted from this source module, or it names an incorrect file. Possibly this code has been copied from another module or another function, but the DEFINE statement was not copied also.
-4371	`Cursors must be uniquely declared within one program module.` In the statement DECLARE cursor-name CURSOR, the identifier cursor-name can be used in only one DECLARE statement in the source file. This is true even when the DECLARE statement appears inside a function. Although a program variable made with the DEFINE statement is local to the function, a cursor within a function is still global to the whole module Search for duplicated cursor names and change the name to have unique identifiers.
-4372	`The cursor 'cursor-name' has not yet been declared in this program.` The name shown appears where the name of a declared cursor or a prepared statement is expected; however, no cursor (or statement) of that name has been declared (or prepared) up to this point in the program. Check the spelling of the name. If it is the name you intended, look back in the program to see why it has not been declared. Possibly the DECLARE statement appears in a GLOBALS file that was not included.
-4374	`This type of statement can only be used within a MENU statement.` This statement only makes sense within the context of a MENU statement. Review the program in this vicinity to see if an END MENU statement has been misplaced. If you intended to set up the appearance of a menu before displaying it, use a BEFORE MENU block within the scope of the MENU.
-4375	`The page length is too short to cover the specified page header and trailer lengths.` A REPORT defines page header and trailer sections with a total number of lines that is not sufficiently less than the specified page length in order to print some detail lines. Review the [FIRST] PAGE HEADER and PAGE TRAILER blocks to use less lines or increase the page length.

Number	Description
-4379	`The input file 'file-name' cannot be opened.` Either the file does not exist, or, on UNIX™, your account does not have permission to read it. Possibly the filename is misspelled, or the directory path leading to the file was specified incorrectly.
-4380	`The listing file 'file-name' cannot be created.` The file cannot be created. Check that the directory path leading to the file is specified correctly and, on UNIX™ systems, that your account has permission to create a file in that directory. Look for other, more explicit, error messages from the operating system. Possibly the disk is full, or you have reached a limit on the number of open files.
-4382	`Record variables that contain array type elements may not be referenced by the ".*" or THROUGH shorthand, or used as a function parameter.` The .* and THROUGH/THRU notation is used to expand a record with an array member. It is allowed to define a record with an array member, but this element must always be used with its full designation of record.array[n]. The .* or THROUGH/THRU notation only expands simple members of the record.
-4383	`The elements 'name-1' and 'name-2' do not belong to the same record.` The two names shown are used where two components of one record are required; however, they are not components of the same record. Check the spelling of both names. If they are spelled as you intended, go back to the definition of the record and see why it does not include both names as component fields.
-4402	`In this type of statement, subscripting may be applied only to array.` The statement contains a name followed by square brackets, but the name is not that of an array variable. Check the punctuation of the statement and the spelling of all names. Names that are subscripted must be arrays. If you intended to use a character substring in this statement, you will have to revise the program.
-4403	`The number of dimensions for the variable 'variable-name' does not match the number of subscripts.` In this statement, the array whose name is shown is subscripted by a different number of dimensions than it was defined to have. Check the punctuation of the subscript. If it is as you intended, then review the DEFINE statement where variable-name is defined.
-4410	`There is a numeric constant in the previous line that is too large or too small.`

Number	Description
	The compiler could not process a numeric constant because it is too big or too small to represent a valid SMALLINT, INTEGER, BIGINT or DECIMAL constant. Check the number of digits and the punctuation of the numeric constant. Make sure you have not typed a letter for a digit for example.
-4414	`The label 'label-name' has been used but has never been defined within the above main program or function.` A GOTO or WHENEVER statement refers to the label shown, but there is no corresponding LABEL statement in the current function or main program. Check the spelling of the label. If it is as you intended it, find and inspect the LABEL statement that should define it. You cannot transfer out of a program block with GOTO; labels must be defined in the same function body where they are used.
-4415	`An ORDER BY or GROUP item specified within a report must be one of the report parameters.` The names used in a ORDER BY, AFTER GROUP OF, or BEFORE GROUP OF statement must also appear in the parameter list of the REPORT statement. It is not possible to order or group based on a global variable or other expression. Check the spelling of the names in the statement and compare them to the REPORT statement.
-4416	`There is an error in the validation string: 'validation-string'.` The validation string in the syscolval table is not correct. Change the appropriate DEFAULT or INCLUDE value in the syscolval table.
-4417	`This type of statement can be used only in a report.` Statements such as PRINT, SKIP, or NEED are meaningful only within the body of a report function, where there is an implicit report listing to receive output. Remove the report specific statement from the code which is not in a report body.
-4418	`The variable used in the INPUT ARRAY or DISPLAY ARRAY statement must be an array.` The name following the words DISPLAY ARRAY or INPUT ARRAY must be that of an array of records. Check the spelling of the name. If it is as you intended, find and inspect the DEFINE statement to see why it is not an array. (If you want to display or input a simple variable or a single element of an array, use the DISPLAY or INPUT statement.)
-4420	`The number of lines printed in the IF part of an IF-THEN-ELSE statement of a header or trailer clause must equal the number of lines printed in the ELSE part.` The runtime system needs to know how many lines will be filled in header and trailer sections (otherwise it could not know how many detail rows to put on the page). Because it cannot tell which part of an IF statement will be executed, it requires that both produce the same number of lines of output. Use the same number of occurrences of PRINT statements in each block of the IF statement.

Number	Description
-4425	`The variable 'variable-name' has not been defined like the table 'table-name'.` The named variable has been used in the SET clause of an UPDATE statement or in the VALUES clause of an INSERT statement, but it was not define LIKE the table being modified. As a result, then runtime system cannot associate record components with table columns. Make sure the schema file is up to date and check that the variable was defined like the table. You can also rewrite the UPDATE or INSERT statement with a different syntax to show the explicit relationship between column names and record components.
-4440	`The field 'field-name-1' precedes 'field-name-2' in the record 'record-name' and must also precede it when used with the THROUGH shorthand.` The THROUGH or THRU shorthand requires you to give the starting and ending fields as they appear in physical sequence in the record. Check the spelling of the names; if they are as you intended, then refer to the VARIABLE statement where the record was defined to see why they are not in the sequence you expected.
-4447	`'key-name' is not a recognized key value.` The key name used in an ON KEY clause is not known by the compiler. Search the documentation for possible key names (F1-F255, Control-?).
-4448	`Cannot open the file 'file-name' for reading or writing.` The file cannot be opened. Verify that the filename is correctly spelled and that your account has permission to read or write to it.
-4452	`The function (or report) 'function-name' has already been defined.` Each function (or report, which is similar to a function) must have a unique name within the program. Change the function or report name.
-4457	`You may have at most 4 keys in the list.` An interactive instruction defines a ON KEY() clause with more that 4 keys. Remove keys from the list.
-4458	`One dimension of this array has exceeded the limit of 65535.` The program is using a static array with a dimension that exceeds the limit. Use a dimension below the 65535 limit.
-4463	`The NEXT FIELD statement can only be used within an INPUT or CONSTRUCT statement.` The NEXT FIELD statement is used outside an INPUT, INPUT ARRAY or CONSTRUCT statement.

Number	Description
	Remove the NEXT FIELD statement from that part of the code.
-4464	`The number of columns must match the number of values in the SET clause of an UPDATE statement.` In an UPDATE statement, the number of values used does not match the number of columns. Check for the table definition, then either add or remove values or columns from the UPDATE statement.
-4476	`Record members may not be used with database column substring.` This statement has a reference of the form name1.name2[...]. This is the form in which you would refer to a substring of a column: table.column[...]. However, the names are not a table and column in the database, so BDL presumes they refer to a field of a record. Inspect the statement and determine what was intended: a reference to a column or to a record. If it is a column reference, verify the names of the table and column in the database. If it is a record reference, verify that the record and component are properly defined.
-4477	`The variable 'variable-name' is an array. You must specify one of its elements in this statement.` You tried to use an array without element specification in a SQL statement. Use one of the members of the array.
-4485	`Only blob variables of type BYTE or TEXT may be used in a LOCATE statement.` The LOCATE statement is using a variable defined with a data type different from BYTE or TEXT. Make sure the variables used with LOCATE are defined as BYTE or TEXT.
-4488	`The program cannot CONTINUE or EXIT statement-type at this point because it is not immediately within statement-type statement.` This CONTINUE or EXIT statement is not appropriate in its context. Review your code. Possibly the statement is misplaced, or the statement type was specified incorrectly.
-4489	`A variable used in the above statement must be a global variable.` A REPORT routine is defining an OUTPUT REPORT TO using a local function variable or report parameter. Review the report clause to use a global or module variable instead.
-4490	`You cannot have multiple BEFORE clauses for the same field.` You cannot specify more than one BEFORE FIELD clause for the same field. Review your code to eliminate multiple BEFORE FIELD clauses.
-4491	`You cannot have multiple AFTER clauses for the same field.`

Number	Description
	You cannot specify more than one AFTER FIELD clause for the same field. Review your code to eliminate multiple AFTER FIELD clauses.
-4534	`Wordwrap may not be used within report headers or trailers.` The report routine uses the WORDWRAP clause in the FIRST PAGE HEADER, PAGE HEADER or PAGE TRAILER sections. Remove the WORDWRAP clause from the expression.
-4631	`Startfield of DATETIME or INTERVAL qualifiers must come earlier in the time-list than its endfield.` The qualifier for a DATETIME or INTERVAL consists of start TO end, where the start and end are chosen from this list: YEAR MONTH DAY HOUR MINUTE SECOND FRACTION. The keyword for the start field must come earlier in the list than, or be the same as, the keyword for the end field. Check the order of the startfield and endfield qualifiers. For example, qualifiers of DAY TO FRACTION and MONTH TO MONTH are valid but one of MINUTE TO HOUR is not.
-4632	`Parenthetical precision of FRACTION must be between 1 and 5. No precision can be specified for other time units.` In a DATETIME qualifier only the FRACTION field may have a precision in parentheses, and it must be a single digit from 1 to 5. Check the DATETIME qualifiers in the current statement; one of them violates these rules. The first field of an INTERVAL qualifier may also have a parenthesized precision from 1 to 5.
-4652	`The function 'function-name' can only be used within an INPUT or CONSTRUCT statement.` The function shown is being used outside of an INPUT or CONSTRUCT statement. However, it returns a result that is only meaningful in the context of INPUT or CONSTRUCT. Review the code to make sure that an END INPUT or END CONSTRUCT statement has not been misplaced. Review the operation and use of the function to make sure you understand it.
-4653	`No more than one BEFORE or AFTER INPUT/CONSTRUCT clause can appear in an INPUT/CONSTRUCT statement.` There may be only one BEFORE block of statements to initialize each of these statement types. Make sure that the scope of all your INPUT, CONSTRUCT and MENU statements is correctly marked with END statements. Then combine all the preparation code into a single BEFORE block for each one.
-4656	`CANCEL INSERT can only be used in the BEFORE INSERT clause of an INPUT ARRAY statement.`

Number	Description
	The CANCEL INSERT statement is being used outside of the BEFORE INSERT clause of an INPUT ARRAY. Review the code to make sure that CANCEL INSERT has not been used anywhere except in the BEFORE INSERT clause.
-4657	`CANCEL DELETE can only be used in the BEFORE DELETE clause of an INPUT ARRAY statement.` The CANCEL DELETE statement is being used outside of BEFORE DELETE clause of an INPUT ARRAY. Review the code to make sure that CANCEL DELETE has not been used anywhere except in the BEFORE DELETE clause.
-4668	`The report output, specified by a START REPORT statement, is not any of file, pipe, screen, printer, pipe in line mode, or pipe in form mode.` The output of a report can be sent only to any of file, pipe (in form or line modes), screen, or printer. Check the START REPORT instruction and make sure that the OUTPUT clause specifies one of the supported values.
-4900	`This syntax is not supported here. Use [screenrecordname.]screenfieldname.` The field name specification in a BEFORE FIELD or AFTER FIELD is not valid. Check for the field name and use [screenrecordname.]screenfieldname syntax.
-4901	`Fatal internal error: description ( line-number ).` This generic error occurs when the fglcomp compiler cannot identify the problem and must stop processing the source. Check the code near the line displayed in the error message.
-6001	`The license manager daemon cannot be started.` This error occurs when a process creation fails during the start of the license manager. Increase the maximum number of processes allowed (ulimit)
-6012	`Cannot get license information. Check your environment and the license (run 'fglWrt -a info').` See error -6015.
-6013	`Time limited version: time has expired.` The license installed is a license with time limit and time has expired. The program can not start. Contact your distributor or support center.
-6014	`Your serial number is not valid for this version.` The license serial number is invalid for this version of the software.

Number	Description
	Contact your distributor or support center.

-6015	`Cannot get license information. Check your environment and the license (run 'fglWrt -a info').`
	It is not possible for the application to check the license validity.
	• License manager:
	  • The license may not have been installed
	  • The license controller can not communicate with the license manager. Check that the license manager is started and check that the fglprofile entries flm.server and flm.service contain valid information.
	  • The directory **$FLMDIR/lock** and all the files below must have read/write permission.
	• License controller:
	  • The license may not have been installed.
	  • The directory **$FGLDIR/lock** and all the files below must have read/write permission.

-6016	`Cannot get information for license (Error error-num). Check your environment and the license (run 'fglWrt -a info').` The application is unable to check the license validity. See error -6015.
-6017	`User limit exceeded. Cannot run this program.` The maximum number of users allowed by the license has been reached. The program can not start. Contact your distributor or support center.
-6018	`Cannot access internal data file. Cannot continue this program. Please, check your environment(variable-name).` When a client computer starts an application on the server, the application stores data in the **$FGLDIR/lock** directory. The client must have permission to create and delete files in this directory. • Do not remove or modify files contained in the directory $FGLDIR/lock • Change the permissions of the **$FGLDIR/lock** directory, or connect to the server with a user name having the correct permissions.
-6019	`This demonstration version allows one user only.` The demonstration version is designed to run with only one user. Another user or another graphical daemon is currently active. Wait until the user stops the current program, or use the same graphical daemon.
-6020	`Installation: Cannot open 'file-name'.` A file is missing or the permissions are not set for the current user. Check that the file permissions are correct for the user trying to execute the application. If the file is missing, re-install the compiler package.

Number	Description
-6022	`Demonstration time has expired. Please, run this program again.` The runtime demonstration version is valid only for a few minutes after you have started a program. Restart the program.
-6025	`Demonstration time has expired. Please, contact your vendor.` The demonstration version of the product has a time limit of 30 days. Either reinstall a new demonstration version, or call your software vendor to purchase a permanent license.
-6026	`Bad link for runner demonstration. Please, retry or rebuild your runner.` The runner is corrupted.
-6027	`Cannot access license server. Please check the following:` - the license server entry in your resource file. (service port) - the license server host. - the license server program. You have not specified a value for the environment variable [fgllic|fls|flm].server in the $FGLDIR/etc/fglprofile file. Check the fglprofile file for the entry point [fgllic|fls|flm].server and specify the name of the computer that runs the License Manager.
-6029	`Unknown parameter 'param-name' for checking.` The command line of the fglWrt or flmprg tool contains an unknown parameter. Check your command-line parameters and retry the command.
-6031	`Temporary license license-number has expired.` Your temporary runtime license has expired. Call your software vendor to get a new license.
-6032	`command-name: illegal option: 'option-name'.` You are not using a valid option for the specified command. Check your command line syntax and try again.
-6033	`command-name: 'option-name' option requires an argument.` You cannot use this option of the tool without a parameter. Check your command line and try again.
-6034	`Warning! This is a temporary license, installation number is 'installation-number'.` You have installed a temporary license of 30 days. You will have to enter an installation key before the end of this period if you want to keep on running the program.

Number	Description
	This is only a warning message.
-6035	`Cannot read in directory` The compiler cannot access the $FGLDIR/lock directory. The current user must have read and write permissions in this directory. Give the current user read and write permissions to the $FGLDIR/lock directory.
-6041	`Can not retrieve network interface information.` An error occurred while retrieving network interface information. Restart your program. If this does not solve your problem, contact your distributor.
-6042	`MAC Address has changed.` The MAC address of the host has changed since the license was first installed. The license must be reinstalled, or restore the old MAC address.
-6043	`The testing period is finished. You must install a new license.` The test time license of has expired. Call your software vendor to purchase a new license.
-6044	`IP Address has changed.` The IP Address of the host has changed. Restore the IP address of the host, or reinstall the license. This is no longer checked by the latest versions of the license controller.
-6045	`Host name has changed.` The host name has changed. Restore the host name or reinstall the license. This is no longer checked by the latest versions of the license controller.
-6046	`Could not get file reference number information.` The license could not get information about the license file. Reinstall the license. Contact your distributor.
-6047	`The device number of the license file has changed.` The license file has been touched. The license is no longer valid. Reinstall the license. Contact your distributor.
-6048	`The file reference number of the license file has changed.` The license file has been touched. The license is no longer valid. Reinstall the license. Contact your distributor.
-6049	`This product is licensed for runtime only. No compilation is allowed.`

Number	Description
	You have a runtime license installed with this package. You cannot compile BDL source code modules with this license. If you want to compile .4gl source code, you must purchase and install a development license. Contact your distributor.
-6050	`Temporary license license-number expired. Please contact your vendor.` A license with a time limit has been installed and the license has expired. Install a new license to activate the product. Contact your distributor.
-6051	`Temporary license license-number expired. Please contact your vendor.` A license with a time limit has been installed and the license has expired. Install a new license to activate the product. Contact your distributor.
-6052	`Temporary license license-number expired. Please contact your vendor.` A license with a time limit has been installed and the license has expired. Install a new license to activate the product. Contact your distributor.
-6053	`Installation path has changed. It must hold the original installation path.` The value of FGLDIR or the location of FGLDIR has been changed. Ask the person who installed the product for the location of the original installation directory and then set the FGLDIR environment variable.
-6054	`Cannot read a license file. Check installation path and your environment. Verify if a license is installed.` The file that contains the license is not readable by the current user. • License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is readable by the current user. • License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is readable by the current user.
-6055	`Cannot update a license file. Check installation path and your environment. Verify if a license is installed.` The file that contains the license cannot be overwritten by the current user. • License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is writable by the current user. • License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is writable by the current user.
-6056	`Cannot write into a license file. Please check your rights.` The file that contains the license cannot be overwritten by the current user. • License controller: Check that the FGLDIR environment variable is correctly set and that the file $FGLDIR/etc/f4gl.sn is writable by the current user.

Number	Description
	• License manager: Check that the file $FLMDIR/etc/license/lic?????.dat is writable by the current user.
-6057	`Cannot read a license file. Check installation path and your environment. Verify if a license is installed.` The file that contains the license cannot be read by the current user. Check that the current user can read the file $FGLDIR/etc/f4gl.sn. Also check that the FGLDIR environment variable is set correctly.
-6058	`Incorrect license file format. Verify if a license is installed.` The file that contains the license has been corrupted. Reinstall the license. If you have a backup of the current installation of Genero Business Development Language, restore the files located in the $FGLDIR/etc directory.
-6059	`Incorrect license file format. Verify if a license is installed.` The file that contains the license has been corrupted. Reinstall the license. If you have a backup of the current installation of Genero Business Development Language, restore the files located in the $FGLDIR/etc directory.
-6061	`License 'license-number' not installed.` The license shown is not installed. Reinstall the license.
-6062	`No installed license has been found for 'license-number'.` The add-user license can not be installed. No main license found to add users. Contact your distributor.
-6063	`License 'license-number' is already installed.` The license shown is already installed. No particular action to be taken.
-6064	`The resource 'flm.license.number' is required to use the license manager.` In order to use a license manager, the FGLPROFILE entry described in the error message must exist and define a license number.
-6065	`The resource 'flm.license.key' is required to use the license manager.` In order to use a license manager, the FGLPROFILE entry described in the error message must exist and define a license key.
-6066	`License 'license-number' cannot be installed over 'license-number'.`

Number	Description
	The add-user license does not match the main license. The add-user license can not be installed. Contact your distributor.
-6067	`You need a installed license if you want to add users.` The add-user license must be installed after the main license. Install the main license before the add-user license. If this does not solve your problem, contact your distributor.
-6068	`No license installed.` There is no license installed for Genero Business Development Language. Install a license. If a license is already installed, check that the $FGLDIR environment variable is set correctly.
-6069	`Cannot uninstall the license.` There was a problem during the uninstall of the Genero Business Development Language license. Check whether the FGLDIR environment variable is correctly set in your environment and the current user has permission to delete files in the $FGLDIR/etc directory.
-6070	`The license server entry must be set in your resource file in order to reach the license server.` You are using the remote license process and you have set the value of fgllic.server, in $FGLDIR/etc/fglprofile, to localhost or to the 127.0.0.1 address. You must use the real IP address of the computer even if it is the local computer.
-6071	`Cannot use directory 'directory-name'. Check installation path and verify if access rights are 'drwxrwxrwx'.` The compiler needs to operate in the specified directory. Change the permission of this directory.
-6072	`Cannot create file in 'file-name'. Check installation path and verify if access rights are 'drwxrwxrwx'.` The compiler needs to operate in the specified directory. Change the permission of this directory to 777 mode.
-6073	`Cannot change mode of a file in 'file-name'. Verify if access rights are 'drwxrwxrwx'.` The compiler needs to operate in the specified directory. Change the permission of this directory to 777 mode.
-6074	`'file-name' does not have 'rwxrwxrwx' rights or isn't a directory. Check access rights with 'ls -ld <installation-path>/lock' or execute 'rm -r <installation-path>/lock' if no users are connected.` The compiler needs to operate in the specified directory.

Number	Description
	Change the permission of this directory. The $FGLDIR/lock directory contains only data needed at runtime by BDL applications. When the application is finished, you can remove this directory. If you delete this directory while BDL applications are running, the applications will be stopped immediately.
-6075	`Cannot read from directory 'directory-name'. Check installation path and verify if access rights are 'drwxrwxrwx'.` The compiler needs to operate in the specified directory. Change the permission of this directory.
-6076	`Bad lock tree. Please check your environment.` There is a problem accessing the $FGLDIR/lock directory. Check if the current user has sufficient permission to read and write to the $FGLDIR/lock directory. Check also if the FGLDIR environment variable is correctly set.
-6077	`Bad lock tree. Please check your environment.` There is a problem accessing the $FGLDIR/lock directory. Check if the current user has sufficient permission to read and write to the $FGLDIR/lock directory. Check also if the FGLDIR environment variable is correctly set.
-6079	`Cannot get machine name or network IP address. Each graphical client must have an IP address when using a license server. FGLSERVER must hold the IP address or the host name of the client.` You are using the remote license process and you have set the value of fgllic.server, in $FGLDIR/etc/fglprofile, to localhost or to the 127.0.0.1 address. You must use the real IP address of the computer even if it is the local computer. This is also true for the value used with the FGLSERVER environment variable.
-6080	`Cannot get IP address from 'host-name' host. Check the 'flm.server' resource.` The system cannot find the IP address of the specified host. This is a configuration issue regarding your system. The command ping should not reply as well. Correct your system configuration and then try to execute your program.
-6081	`Cannot reach host 'host-name' with ping. Check license server entry in your resource file. Check your network configuration or increase 'flm.ping' value.` The license server cannot ping the client computer, or it does not get the response in the time limit specified by the fgllic.ping entry in the $FGLDIR/etc/fglprofile file. Try to manually ping the specified computer. If this works, try to increase the value of the fgllic.ping entry in fglprofile. If the ping does not respond, fix the system configuration problem and then try the program again.
-6082	`SYSERROR(error-num) description: Cannot set option TCP_NODELAY on socket. Check the system error message and retry.` There is a problem with the socket of the Windows™ computer.

Number	Description
	Check that the system is correctly configured and retry the program.
-6085	`SYSERROR(error-num) description: Cannot connect to the license server on host 'host-name'. Check following things: - license server entry. - the license server machine. - the license server TCP port.` The application cannot check the license validity. To do so, it tries to communicate with the Genero Business Development Language license service running on the computer where the product is installed. Check that the Genero Business Development Language License Server is running on the computer where the product is installed.
-6086	`SYSERROR(error-num) description: Cannot send data to the license server. Check the system error message and retry.` Theres a problem with the socket of the Windows™ computer. Check that the system is correctly configured and retry the program.
-6087	`SYSERROR(error-num) description: Cannot receive data from license server. Check the system error message and retry.` There is a problem with the socket of the Windows™ computer. Check that the system is correctly configured and retry the program.
-6088	`You are not allowed to be connected for the following reason: description` The program cannot connect to the license server because of the specified reason. Try to fix the problem described and rerun your application.
-6089	`Each graphical client must have an IP address when using a license server. FGLSERVER must hold the IP address or the host name of the client (localhost or 127.0.0.1 are not allowed).` Use the real IP address or hostname of the client.
-6090	`SYSERROR(error-num) description: Cannot create a socket to start the license server. Check the system error message and retry.` There is a problem with the socket of the Windows™ computer. Check that the system is correctly configured and rerun the program.
-6091	`SYSERROR(error-num) description: Cannot bind socket for the license server. Check the system error message and retry.` There is a problem with the socket of the Windows™ computer. Check that the system is correctly configured and rerun the program.
-6092	`SYSERROR(error-num) description: Cannot listen socket for the license server.` There is a problem with the socket of the Windows™ computer.

Number	Description
	Check that the system is correctly configured and rerun the program.
-6093	`SYSERROR(error-num) description: Cannot create a socket to search an active client.` There is a problem with the socket of the Windows™ computer. Check that the system is correctly configured and rerun the program.
-6094	`SYSERROR(error-num) description: This is a WSAStartup error. Check the system error message and retry.` There is a problem with the socket of the Windows™ computer. Check that the system is correctly configured and rerun the program.
-6095	`License problem: description` License type incompatible. You are installing an earlier version, which was not designated for use with the current license server. Reinstall and then contact your vendor.
-6096	`Connection refused by the license server.` There is problem connecting the client computer to the Windows™ license server. There is a configuration problem with the license server computer. Check the configuration of the computers and of the products.
-6100	`Bad format of line sent by the license requester.` The license request sent by the license controller is not understood by the license manager. Upgrade your license software to the latest version available. If the issue is not solved, contact your support center.
-6101	`License number 'license-number' does not correspond to license key 'license-key'.` Either the license number or the license key is invalid. Check the license number and keys entered and try again. If that does not solve the issue, upgrade your license software to the latest version available. If the issue is not solved, contact your support center.
-6102	`Verify if resource 'flm.license.number' and 'flm.license.key' correspond to a valid license.` Either the `flm.license.number` or `flm.license.key` entry in `fglprofile` is incorrectly filled. Ensure these `fglprofile` entries contain valid license numbers.
-6103	`License 'license-number' is no longer available from the license server.` The license has been uninstalled from the license server. It may still appear as some sessions are active, but the license can not be used to start a new session. Reinstall the license, or contact your support center.

Number	Description
-6107	`User limit exceeded. Please retry later.` The maximum number of clients that can be run has been reached (due to the license installed). Retry later (when the number of current users has decreased) or install a new license that allows more users.
-6108	`Environment is incorrect.` There is no local license, or the environment is not set correctly. Check your environment and your FGLDIR environment variable.
-6109	`Cannot add session #session-number.` You do not have the permissions to create the new session (the directory representing the new client). Check the permissions of the dedicated directories.
-6110	`Cannot add program 'program-name' (pid=processid).` You do not have the permissions to create the new application (the file representing the new application) for the current user . Check the permissions of the dedicated directories.
-6112	`Compilation is not allowed: This product is licensed for runtime only.` Buy and install a development license.
-6113	`Compilation is not allowed: Invalid license.` Buy and install a development license.
-6114	`Cannot start program 'program-name' or result of process number is 0.` When fglWrt -u is executed to find the number of users allowed on this installation, the command "ps" may be launched (only on UNIX™). Check the permissions for ps.
-6116	`Wrong number of characters.` The license number, license key, installation number, installation key or maintenance key provided is incomplete. Ensure that provided license numbers are correct and try again.
-6117	`The entry must be 12 characters long.` The license number, license key, installation number, installation key or maintenance key provided is incomplete. Ensure that provided license numbers are correct and try again.
-6118	`Wrong checksum result for this entry.`

Number	Description
	When entering license numbers, the checksum is verified if it is provided. This error occurs if the checksum computed does not match the provided checksum. Either the checksum or the license number is wrong. Ensure that checksum and license numbers are correct and try again.
-6122	`You must specify entry 'flm.server' in the resource file.` The `fglprofile` entry `flm.server` is missing. This entry should contain the host name or IP address of the host running the license manager. Add and configure the fglprofile entry `flm.server`.
-6123	`SYSERROR(error-num) description: Cannot open socket. Check the system error message and retry.` The license controller can not connect to the license manager. Check the error message and fix the issue. Ensure that `fglprofile` entries `flm.server` and `flm.service` are correctly filled. Check your network configuration.
-6129	`License uninstalled.` This is an information message.
-6130	`This license requires a full installation.` The installed license has not be activated, but can not be used in temporary installation mode. Contact your vendor to obtain the activation key.
-6131	`This license number is no more valid. Please, contact your vendor.` The license number is no longer accepted. Contact your vendor to obtain a new license number.
-6132	`Incompatible License Controller (fglWrt/greWrt) version. The minimum version required is min-version.` Upgrade your license controller version to the specified version or higher.
-6133	`This product requires a BDL license. The license number should start with the letter F.` A BDL license is required for this product. Call you support center to get a BDL license.
-6134	`This product requires a Genero license. The license number should start with the letter T.` A Genero license is required for this product. Call you support center to get a Genero license.
-6135	`Invalid license key.` The license key does not correspond to the license number.

Number	Description
	Call you support center to check the license key.
-6136	`The date-limited license has expired.` The time limited license has expired, the product is blocked. Call you support center to get a new license.
-6137	`This product requires a GRW license .` A GRW license is required for this product. Call you support center to get a GRW license.
-6138	`GRW licenses are not accepted by this product .` This product does not accept GRW licenses. Call you support center to check if the license corresponds to the product.
-6140	`Version version-number` This is an information message.
-6142	`Try and buy demonstration time expired. Please, restart your application.` Applications started with a Try and Buy version will stop after few minutes of execution. Restart your application.
-6143	`This license requires a valid maintenance key. Check your environment (run 'fglWrt/greWrt -a info')` Genero 2.20 and higher require a valid maintenance key. Update your maintenance key.
-6144	`The DVM build date is greater than the maintenance key expiration date. Contact your nearest FourJ's sales representative to update the maintenance key.` Update your maintenance key or downgrade your Genero installation to an older version.
-6146	`This product requires a Genero Time-Limited Evaluation license.` You have installed a trial version of the Genero product, but the installed license is not a trial license. Install a trial license for this product. Contact your support to get a trial license.
-6147	`This product requires a GRE Time-Limited Evaluation license.` You have installed a trial version of the GRE product, but the installed license is not a trial license. Install a trial license for this product. Contact your support to get a trial license.
-6148	`Installation path is not known.` You are handling licenses but the FGLDIR environment variable is not set.

Number	Description
	Set the FGLDIR environment variable and retry.
-6149	`Problem while installing license 'license-number'.` A problem occurred while licensing. Note the system-specific error number and contact your Technical Support.
-6150	`Temporary license not found for this version.` While adding a definitive license key, the temporary license has not been found. Re-install the license.
-6151	`Wrong installation key.` While adding a definitive license key, the installation key was not valid. Re-install the license.
-6152	`Problem during license installation.` A problem occurred while installing the license. Could not write information to the disk (either own files or system files). Check the FGLDIR environment variable and the rights of the license files (must be able to change them).
-6153	`License installation failed.` License information could not be written to files. Check the system error message if provided, check the file permissions for the current user.
-6154	`License installation successful.` This is an information message.
-6156	`Too many temporary licenses. You must reinstall a license.` You installed a temporary license too many times. Contact technical support to get a valid license.
-6158	`Cannot store temporary information.` A problem occurred while installing the license. Could not write information to the disk (either own files or system files). Check the FGLDIR environment variable and the rights of the license files (you must be able to change them).
-6159	`This kind of license is not permitted.` The license numbers can not be installed. Contact your support center.
-6160	`You do not have the permissions to be connected.` The host running the license controller (where the DVM is running) is not allowed to connect to this license manager. There is likely a configuration issue.

Number	Description
	Check your license manager configuration.
-6161	`You do not have the permissions to compile.` The compilation request is rejected by the license manager. Contact your support center.
-6162	`Cannot reach the license server. Please check if 'flm.server' is correctly initialized. ('flmprg -a info up' command should answer 'ok'). The license server is running but no autocheck will be done.` While this error is no longer used, it can be raised by older versions of Genero. The license controller can not connect to the license manager. Ensure that `fglprofile` entries `flm.server` and `flm.service` are filled correctly. Ensure that the license manager is running on the specified host and port.
-6168	`Cannot get information from directory 'directory-name'.` Failed to read directory information. Ensure that the user installing a license is the user that installed the product. Ensure that the user installing a license has read/write permissions on the '`etc`' directory of the product.
-6169	`SYSERROR(error-num) description: Cannot set option O_NONBLOCK on socket. Check the system error message and retry.` Failed to configure the socket in non-blocking mode. Check the system error message. Contact your support center.
-6170	`Old request format to license server detected. You must install a license program version 2.99 or higher.` The license controller version is too old for the current license manager version. Requests sent by the license controller are no longer supported by the current license manager. Upgrade the license controller to the latest version available.
-6171	`A license has been installed temporarily. Only the installation key is required. You must run 'fglWrt -k <installation-key>' to install it.` The installed license is temporarily installed, yet it is missing the installation key. Obtain your installation key and install it.
-6172	`Bad parameter: 'parameter' hasn't the right format.` Two issues can raise this error. **1.** The license manager can raise this error if it receives a request from the license controller with unknown commands. Upgrade the license software version to the latest available. Contact your support center. **2.** The license manager `etc/lmprofile` configuration is invalid.

Number	Description
	Check your `lmprofile` entries `flm.license.together` and `flm.license.allow`.
-6173	`Invalid license number or invalid license key.` During the license installation, the license number / license key couple does not match. Ensure that the license numbers are correct. Upgrade the license software to the latest version available and retry.
-6174	`This option is only available for a local license. And resource 'flm.server' was found in your configuration.` A license server is configured and the user tries to install or uninstall a license using `fglWrt`. Install the license on the license manager (`flmprg`) Or remove the license manager configuration from `fglprofile` and install the license locally.
-6175	`License number 'license-number' is invalid.` The license number is not valid. Ensure that the license number provided is correct. Upgrade the license software to the latest version available.
-6176	`In license server, following problem occurs with license number 'license-number': problem-description` This is a generic error containing the text of another error. Check the error. Contact your support center.
-6177	`Following problem occurs with license number 'license-number': description` This is a generic error containing the text of another error. Check the error. Contact your support center.
-6178	`Your machine is not allowed to be connected on any of your authorized licenses.` The CPU license rejects the connection of a new host. All CPU licenses are consumed. Contact your support center.
-6179	`License validity time is reached. The users control is reactivated.` The CPU license is time limit is reached. The CPU license is degraded and user control is reactivated. Contact your support center.
-6180	`CPU limit exceeded. The users control is reactivated.` The CPU license has less CPUs available than the number of CPUs on the connected host, resulting in one or more hosts running in degraded mode. When running in degraded mode, the number of users allowed is the number of available CPUs in the license * 100 users. User control is activated for license request coming from that host.

Number	Description
	For example, say you have a 5 CPU license and are using the license manager. • Host A with 2 CPUs connects. It consumes 2 CPUs of the 5 CPU license. 3 CPUs remain free on the license. • Host B with 2 CPUs connects. It will consume 2 CPUs of the 5 CPU license. 1 CPU remains free on the license. • Host C with 2 CPU connects. It should consume 2 CPUs, but only 1 CPU remains on the license. User control for Host C is enabled, with 1 CPU (the remaining free CPU) * 100 users allowed. • Host D connects with *N* CPU, however there are no CPUs free on the license. Host D is rejected. In summary, Host A and B can have unlimited sessions, Host C is limited to 100 sessions as the license is degraded by CPU, and any other host is rejected. In the case of a single host, user control applies to local licenses as well. If Host A has 6 CPUs, yet has a 5 CPU license, it consumes all of the CPUs for the license and run in degraded mode. That means 5 * 100 users are allowed. Contact your support center.
-6181	`Cannot get license extension information. Check your environment, the license (run 'fglWrt -a info') and the fglWrt version ('fglWrt -V' should give version-number or higher).` License information is invalid. This error is not yet used. Contact your support center.
-6182	`Your license has 'restriction-name' restriction. You are not allowed to run another mode.` The license has restrictions, and the requested use of the license is not compatible with these restrictions. For example, the license may have a text-only restriction, where GUI front-ends are not allowed. Contact your support center to obtain a license matching your needs.
-6183	`Local license controller (fglWrt) may not be compatible with this runner. Check its version ('fglWrt -V' should give version-number or higher).` The license controller is incompatible. Update the license controller to the latest version available.
-6184	`You are not authorized to run this version of runner.` Older licenses do not use the maintenance key. The DVM version that can be used is limited. The DVM version is higher that the allowed DVM version. Contact your support center. Re-licensing is required.
-6185	`Protection file is not compatible with this version of the runner. You must reinstall your license.` This error is no longer used by the current licensing software, however it may occur with older versions. Contact your support center.

Number	Description
-6186	`Demo version initialization.` This is an information message.
-6188	`Your evaluation license period has expired. Contact your support center.` The software you are using has been installed with a demo license that has expired. Contact your software vendor to extend the evaluation period or purchase a permanent license.
-6196	`You are not authorized to delete sessions from the license server 'server-name'.` The command `fglWrt -i` can only be used with local licenses. If a license server is configured, this error is raised. Use the command `flmprg` instead.
-6197	`'extension-name' extension is not allowed with this license type.` Generic error indicating that an extension check is rejected. For example, if you are using a non-Informix database, this error will raise if the ODI extension is not set in the license.
-6198	`Product identifier does not correspond to the license number.` This error indicates that a wrong license is installed in the product, such as when you attempt to use a Genero Report Writer (GRW) license when installing the Genero Business Development Language (BDL). This error should not be raised, as `fglWrt` will reject the installation of a Genero Report Engine (GRE) license when installing Genero BDL, and conversely `greWrt` will not allow the installation of a Genero BDL license. That being said, the installation checks to ensure the license is valid for the product, and raises this error it if is not. Ensure the proper license is used with the proper package.
-6199	`Cannot create directory 'directory-name'. Check installation path and verify your access rights.` The specified directory can not be created or modified.
-6200	`Module 'module-name': The function function-signature-1 will be called as function-signature-2 .` An incorrect number of parameters are used to call a BDL function. Check your source code and recompile your application.
-6201	`Module 'module-name': Bad version: Recompile your sources.` You have compiled your program with an old version. The newly compiled version of your program is not supported. Compile all source files and form files again.

Number	Description
-6202	`filename 'file-name': Bad magic: Code cannot run with this p code machine.` You have compiled your program with an old version. The new compiled version of your program is not supported. You might also have a file with the same name as the .42r. You used the fglrun 42r-Name without specifying the extension. To resolve this problem, call fglrun with the .42r extension or recompile your application.
-6203	`Module 'module-name-1': The function 'function-name' has already been defined in module 'module-name-2'.` The specified function is defined for the second time in the application. The second occurrence of the function is in the specified module. Eliminate one of the two function definitions from your source code.
-6204	`Module 'module-name': Unknown op-code.` An unknown instruction was found in the compiled BDL application. Check that the version of the Genero Business Development Language package executing the compiled application is the same as the one that compiled the application. It is also possible that the compiled module has been corrupted. If so, you will need to recompile your application.
-6205	`INTERNAL ERROR: Alignment.` This error is internal, which should not normally occur. Contact your Technical Support.
-6206	`The 42m module 'module-name' could not be loaded, check FGLLDPATH environment variable.` The 42m module is not in the current directory or in one of the directories specified by the **FGLLDPATH** environment variable. Set the environment variable **FGLLDPATH** with the path to the 42m modules to be loaded.
-6207	`The dynamic loaded module 'module-name' does not contain the function 'function-name'.` A BDL module has been changed and recompiled, but the different modules of the application have not been linked afterward. Link the new modules together before you execute your application.
-6208	`Module 'module-name' already loaded.` A module was loaded twice at runtime. This can occur because one module has been concatenated with another. Recompile and re-link your BDL modules.
-6210	`INTERNAL ERROR: exception 2 raised before invoking the exception handler for exception 1.`

Number	Description
	A module was loaded twice at runtime. This can occur because one module has been concatenated with another. Check for function names, recompile and re-link your BDL modules.
-6211	`Link has failed.` A problem occurred while linking the BDL program. Check for function names, recompile and re-link your BDL modules.
-6212	`Function function-name : local variables size is too large - Allocation failed.` A local function variable is too large and runtime could not allocate memory. Review the variable data types in the function.
-6213	`Module module-name : Module's variable size is too large - Allocation failed.` A module variable is too large and runtime could not allocate memory. Review the variable data types in the module.
-6214	`Global variable variable-name size is too large - Allocation failed.` A global variable is too large and runtime could not allocate memory. Review the variable data types in the globals.
-6215	`Memory allocation failed. Ending program.` Runtime could not allocate memory. Check for system resources and verify if the OS user is allowed to allocate as much memory as the program needs (check for **ulimits** on UNIX™ systems).
-6216	`The global 'constant-name' has been redefined with a different constant-value.` A global constant has been defined twice with a different value. A global constant may have only one value. Review your code.
-6217	`The global 'variable-name' has been defined as a constant and a variable.` The same symbol was used to define a constant and a variable. Use a different name for the constant and the variable. Review your code.
-6218	`No runtime. You must call fgl_start() before calling fgl_call().` This error occurs when a C extension has redefined the main() routine, but then does not call fgl_start() to initialize the BDL runtime environment. Check the C extension and call fgl_start() before any other operation.
-6219	`WHENEVER ERROR CALL: The error-handler recursively calls itself.`

Number	Description
	The function specified with the WHENEVER ERROR CALL instruction raises an error that would call itself recursively. Review the function called by the WHENEVER ERROR CALL and make sure it does not produce a runtime error.
-6220	`Could not load C extension library ' library-name'. Reason: description` Runtime system could not find the shared library for the reason given. Check if the C extension library exists in one of the directories defined by FGLLDPATH. If the C extension module depends from other shared libraries, make sure that these libraries can be found by the library loader of the operating system (check the LD_LIBRARY_PATH environment variable on UNIX™ or the PATH environment variable on Windows™).
-6221	`C extension initialization failed with status number.` C extension failed to initialize and returned the status shown in the error message. Check the C extension source or documentation.
-6222	`class-name class not found.` The program was compiled with the built-in class class-name but at execution time the class is not found. Check you installation, it is possible that you are executing program that was compiled with a younger version as the version used in the execution context, which certainly is missing that class in the runtime library.
-6223	`No such symbol: symbol-name.` The runtime system loads a module dynamically (on demand) and searches the symbol in this module. But the symbol could not be found, for example because of an invalid FGLLDPATH, or because the installed module does no more contain the symbol (after a recompilation).
-6300	`Can not connect to GUI: description` You have run a GUI application but the environment variable FGLSERVER is not set correctly, or the Genero client (graphical front-end) is not running. See the description for more details. The FGLSERVER environment variable should be set to the hostname and port of the graphical front end used by the runtime system to display the application windows. Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.
-6301	`Can not write to GUI: description` You are running a GUI application but for an unknown reason the front-end no longer responds and the runtime system could not write to the GUI socket. Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.
-6302	`Can not read from GUI: description`

Number	Description
	You are running a GUI application but for an unknown reason the front-end no longer responds and the runtime system could not read from the GUI socket. Check that the network connection is still available, make sure no firewall denies access to the workstation, and see whether the front-end is still running.
-6303	`Invalid user interface protocol.` You are trying to execute a program with a runtime system that uses a different AUI protocol version as the front-end. Install either a new front-end or a new runtime environment that matches (2.0x with 2.0x, 1.3x with 1.3x).
-6304	`Invalid abstract user interface definition.` You are trying to execute a program with a runtime system that uses a different AUI protocol version as the front-end. Install either a new front-end or a new runtime environment that matches (2.0x with 2.0x, 1.3x with 1.3x).
-6305	`Can not open char table file. Check your fglprofile.` This error occurs if the conversion file defined by the gui.chartable entry, in the $FGLDIR/etc/fglprofile file, is not readable by the current user. Check if the gui.chartable entry is correctly set and if the specified file is readable by the current user.
-6306	`Can not open server file. Check installation.` A file on the server side cannot be sent to the graphical interface. Check the permissions of the file located in the $FGLDIR/etc directory. These files must have at least read permission for the current user.
-6307	`GUI server autostart: can not identify workstation.` GUI Server autostart configuration is wrong. Either DISPLAY, FGLSERVER or fglprofile settings are invalid. Set the required environment variables and check for fglprofile autostart entries.
-6308	`GUI server autostart: unknown workstation: check gui.server.autostart entries.` The computer described by the X11 DISPLAY environment variable is neither the local host, nor is it listed in the fglprofile entries. Check if the X11 DISPLAY name is correctly set, or review the fglprofile entries.
-6309	`Not connected. Cannot write to GUI.` For unknown reasons there was an attempt to write on the GUI socket before the connection was initiated. Check the program for invalid GUI operations.
-6310	`Not connected. Cannot read from GUI.`

Number	Description
	For unknown reasons there was an attempt to read on the GUI socket before the connection was initiated. Check the program for invalid GUI operations.
-6311	`No current window.` The program tries to issue a MENU instruction with no current window open. Review the program logic and make sure a window is open before MENU.
-6312	`The type of the user interface (FGLGUI) is invalid.` While initiating the user interface, the runtime system did not recognize the GUI type and stopped. Make sure the FGLGUI environment variable has a correct value.
-6313	`The UserInterface has been destroyed.` The error occurs when the front-end sends a DestroyEvent event, indicating some inconsistency with the starting program. This can happen, for example, when multiple StartMenus are used, or when you try to run an MDI child without a parent container, or when two MDI containers are started with the same name, etc. Check for inconsistency and fix it.
-6314	`Wrong connection string. Check client version.` While starting the program, the runtime received a wrong or incorrectly constructed answer from the front-end. Make sure you are using a front-end that is compatible with the runtime system.
-6315	`The form is too complex for the console-ui.` The program tries to display a form with a complex layout that cannot be displayed in text mode. Review the form file and use a simple grid with a SCREEN section instead of LAYOUT.
-6316	`Error error-num returned from client: description` Front end returned the specified error during GUI connection initialization. Check the front-end documentation for more details.
-6317	`Invalid or unsupported client protocol feature.` The GUI protocol feature you are trying to use is not supported by the front-end. For example, you are trying to use protocol compression but the runtime is not able to compress data. Make sure that the front-end component is compatible with the runtime system (versions must be close). Check the runtime system version for supported protocol features. If compression is enabled, check that the zlib library is installed on your system.
-6318	`Choosing the DIALOG implementation by setting the environment variable FGL_USENDIALOG=0 has been desupported since version 2.20.03.`

Number	Description
	You try to use the old dialog implementation by setting FGL_USENDIALOG to zero. The old dialog implementation has been removed, you must unset the FGL_USENDIALOG environment variable.
-6319	`Internal error in the database library. Set FGLSQLDEBUG to get more details.` An unexpected internal error occurred in the database driver. Set the FGLSQLDEBUG environment variable to level 1, 2, 3 or 4 to get detailed debug information.
-6320	`Can't open file 'file-name'.` The runtime system tried to open a resource file in FGLDIR but access is denied or file no longer exists. Check for file permissions and existence in FGLDIR.
-6321	`No such interface capability: 'feature-name'.` The runtime system tried to use a front-end protocol capability, but is not able to use it. Check if the front-end is compatible with the runtime system.
-6322	`version-num-1 wrong version. Expecting version-num-2.` Some resource files of FGLDIR have been identified as too old for the current runtime system. Re-install the runtime system environment.
-6323	`Can't load factory profile 'file-name'.` The default fglprofile file located in FGLDIR/etc is missing or is unreadable. Check the permission of the file. If the file is missing, reinstall the software.
-6324	`Can't load customer profile 'file-name'.` The configuration file defined by the FGLPROFILE environment variable is missing or unreadable. Check if the FGLPROFILE environment variable is correctly set and if the file is readable by the current user.
-6325	`Can't load application resources 'file-name'.` The directory specified by the fglrun.default entry in FGLDIR/etc/fglprofile is missing or not readable for the current user. Check if the entry fglrun.default is correctly set in FGLDIR/etc/fglprofile and if the directory specified is readable by the current user.
-6327	`Internal error in the run time library file library-name.` Something unpredictable has occurred, generating an error. Contact your Technical Support.

Number	Description
-6328	`Bad format of resource 'entry-name' value 'entry-value': you must use the syntax: entry-name='VARNAME=envvar-value'.` The FGLPROFILE file contains an invalid environment variable definition format. Check the content of the profile file.
-6329	`All TABLE columns must be defined with the same height.` The form layout defines a TABLE with field tags using different heights. Review all cells of the table to use the same height in all columns.
-6330	`Syntax error in profile 'file-name', line number lineno , near 'token'.` The FGLPROFILE file shown in the error message contains a syntax error. Check the content of the profile file.
-6331	`Front end module could not be loaded.` A front call failed because the module does not exist. The front end is probably not supporting this module.
-6332	`Front end function could not be found.` A front call failed because the function does not exist. The front end is probably not supporting this function.
-6333	`Front end function call failed. Reason: description` A front call failed for an unknown reason. Call the support and report the problem.
-6334	`Front end function call stack problem.` A front call failed because the number of parameter or returning values does not match. Make sure the number of parameters and return values are correct.
-6340	`Can't open file: description` The channel object failed to open the file specified. Make sure the filename is correct and user has permissions to read/write to the file.
-6341	`Unsupported mode for 'open file'.` You try to open a channel with an unsupported mode. See channel documentation for supported modes.
-6342	`Can't open pipe.` The channel object failed to open a pipe to execute the command. Make sure the command you try to execute is valid.
-6343	`Unsupported mode for 'open pipe'.`

Number	Description
	You try to open a channel with an unsupported mode. See channel documentation for supported modes.
-6344	`Can't write to unopened file, pipe or socket.` You try to write to a channel object which is not open. First open the channel, then write.
-6345	`Channel write error: description` An unexpected error occurred while writing to the channel. See the description for more details.
-6346	`Cannot read from unopened file, pipe or socket.` You try to read from a channel object which is not open. First open the channel, then read.
-6360	`This runner cannot execute any SQL.` The runtime system is not ready for database connections. Check the configuration of BDL.
-6361	`Dynamic SQL: type unknown: type-name .` The database driver does not support this SQL data type. You cannot use this SQL data type, review the code.
-6364	`Cannot connect to sql back end.` The runtime system could not initialize the database driver to establish a database connection. Make sure the database driver exists.
-6365	`Database driver not connected yet.` There is an attempt to execute an SQL statement, but no database connect is established. First connect, then execute SQL statements.
-6366	`Could not load database driver driver-name.` The runtime system failed to load the specified database driver. The database driver shared object (.so or .DLL) or a dependent library could not be found. Make sure that the specified driver name does not have a spelling mistake. If the driver name is correct, there is probably an environment problem. Make sure the database client software is installed. Check the UNIX™ LD_LIBRARY_PATH environment variable or the PATH variable on Windows™. These must point to the database client libraries.
-6367	`Incompatible database driver interface.`

Number	Description
	The database driver interface does not match the interface expected by the runtime system. This can occur if you copy an old database driver into a younger FGLDIR installation. Call the support to get a valid database driver.
-6368	`SQL driver initialization function failed.` The runtime system failed to initialize the database driver, program must stop because no database connection can be established. There is probably an environment problem (for example, INFORMIXDIR or ORACLE_HOME is not set). Check your environment and try to connect with a database vendor tool (dbaccess, sqlplus) to identify the problem.
-6369	`Invalid database connection mode.` You try to mix DATABASE and CONNECT statements, but this is not allowed. Use either DATABASE or CONNECT.
-6370	`Unsupported SQL feature.` This SQL command or statement is not supported with the current database driver. Review the code and use a standard SQL feature instead.
-6371	`SQL statement error number error-num (native-error).` An SQL error has occurred having the specified error number. You can query SQLERRMESSAGE or the SQLCA record to get a description of the error.
-6372	`General SQL error, check SQLCA.SQLERRD[2].` A general SQL error has occurred. You can query SQLERRMESSAGE or the SQLCA record to get a description of the error. The native SQL error code is in SQLCA.SQLERRD[2].
-6373	`Invalid database connection string.` The database connection string that you have used is not valid. Verify the format of the connection string.
-6374	`Wrong database driver context.` You try to EXECUTE, OPEN, FETCH, PUT, FLUSH, CLOSE or FREE a cursor that was declared or prepared in a different connect and driver. Issue a SET CONNECTION before the statement to select the same connection and driver as when the cursor was created.
-6375	`LOAD cannot get describe information for table columns.` The LOAD instructions needs column description to allocate the automatic fetch buffers, but the database driver is not able to describe the table columns used in the INSERT statement.

Number	Description
	If the underlying database client API does not provide result set column description, the LOAD statement cannot be supported.
-6601	`Can not open Database dictionary 'directory-name'. Run database schema extraction tool.` The schema file does not exist or cannot be found. If the schema file exists, verify that the filename is spelled correctly, and that the file is in the current directory or the FGLDBPATH environment variable is set to the correct path. If the file does not exist, run the database schema extraction tool to create a schema file.
-6602	`Can not open globals file 'file-name'.` The globals file does not exist or cannot be found. Verify that the globals file exists. Check the spelling of the filename, and verify that the path is set correctly.
-6603	`The file 'file-name' cannot be created for writing.` The compiler failed to create the file shown in the error message for writing. Check for user permissions to make sure that the **.42m** file can be created.
-6604	`The function 'function-name' can only be used within an INPUT [ARRAY], DISPLAY ARRAY or CONSTRUCT statement.` The language provides built-in functions that can only be used within specific interactive statements. Review your code and make the necessary corrections.Check that the function is within the interactive statement and that appropriate END statements (END INPUT/ARRAY/ DISPLAY ARRAY/CONSTRUCT) have been used.
-6605	`The module 'module-name' does not contain function 'function-name'.` The module shown in the error message does not hold the function name as expected. The specified function needs to be defined in this module.
-6606	`No member function 'function-name' for class 'class-name' defined.` The function name is misspelled or is not a method of the class for which it is called. Review your code and the documentation for the method you are attempting to use. If the function is an object method, make sure the referenced object in your code is of the correct class.
-6608	`Resource error: entry-name :parameter expected` This is a generic error message for resource file problems.
-6609	`A grammatical error has been found at 'seen-token' expecting: expected-token.` A general syntax error message that indicates the location of the problem code and what code was expected.

Number	Description
	Review your code, particularly for missing END statements such as END FUNCTION or END INPUT, etc., and make the necessary corrections.
-6610	`The function 'function-name' has already been called with a different number of parameters.` Earlier in the program, there is a call to this same function or event with a different number of parameters in the parameter list. Check the correct number of parameters for the specified function. Then examine all calls to it, and make sure that they are written correctly.
-6611	`Function 'function-name': unexpected number of returned values.` The function shown returned a different number of values as expected. Check the body of the function for RETURN instructions.
-6612	`Redeclaration of function 'function-name'.` The function shown was defined multiple times. Change the name of conflicting functions.
-6613	`The library function 'function-name' is not declared.` The function shown was not declared. Change the name of the function.
-6614	`The function 'function-name' may return a different number of values.` The function shown contains multiple RETURN instructions which may return different number of values. Review the RETURN instructions to return the same number of values.
-6615	`The symbol 'symbol-name' is unused.` This is a warning indicating that the shown symbol is defined but never used. Useless definition can be removed.
-6616	`The symbol 'symbol-name' does not represent a defined CONSTANT.` The shown symbol is used as a CONSTANT, but it is not a constant. Review your code and check for this name.
-6617	`The symbol 'symbol-name' is a VARIABLE.` The symbol shown is a VARIABLE which cannot be used in the current context. Review your code and check for this name.
-6618	`The symbol 'symbol-name' is a CONSTANT.` The symbol shown is a CONSTANT which cannot be used in the current context. Review your code and check for this name.
-6619	`The symbol 'symbol-name' is not an INTEGER CONSTANT.`

Number	Description
	The symbol shown is used as if it was an INTEGER constant, but it is not. Review your code and check for this name.
-6620	`The symbol 'symbol-name' is not a REPORT.` The symbol shown is used as a REPORT, but it is not defined as a REPORT. Review your code and check for this name.
-6621	`The symbol 'symbol-name' is not a FUNCTION.` The symbol shown is used as a FUNCTION, but it is not defined as FUNCTION. Review your code and check for this name.
-6622	`The symbol 'symbol-name' does not represent a valid variable type.` The symbol shown is not known as a valid type to define a program variable. Review your code and check for the type name.
-6623	`The method 'method-name' cannot be called without an object.` The specified method is an object method of its class. Review your code. Ensure that the required object of the class has been instantiated and still exists, and that the method is called specifying the object variable as the prefix, with the period character as a separator.
-6624	`The method 'method-name' cannot be called with an object.` The specified method is a class method and cannot be called using an object reference. No object has to be created. Review your code. Ensure that the method is called using the class name as the prefix, with the period character as a separator.
-6625	`The statement is not Informix compatible.` The SQL statement is not Informix® compatible. Change the SQL statement by using Informix® SQL syntax.
-6627	`The symbol 'symbol-name' is not a VARIABLE.` The symbol shown is use as a variable, but is not defined as a variable. Review your code and check for this name.
-6628	`The GLOBALS file does not contain a GLOBALS section.` The filename specified in a GLOBALS statement references a file that does not contain a GLOBALS section. Review your code to make sure that the file specified by the filename is a valid GLOBALS file, containing the required GLOBALS section.
-6629	`The type 'type-name' is too complex to be used within a C-extension.`

Number	Description
	The type of the global variable is too complex to be used in a C extension. This error can occur when the -G option of fglcomp, to generate the C sources to share global variables with C extensions, when a global variable is defined with complex data types without a C equivalent. Review the definition of the global variables and use simple types instead, corresponding to a C data type. The BYTE, TEXT and STRING types are complex types.
-6630	`Memory overflow occurred during p-code generation. Simplify the module.` A memory overflow occurred during compilation to p-code because the .4gl source module is too large. This problem can occur with very large source files. You must split the module into multiple sources.
-6631	`Incompatible types, found: source-type, required: target-type.` A LET or RETURNING tries to assign a value or an object reference to a variable defined with a data type or class that is not compatible to the value type. This occurs usually when using Java™ classes because Java™ is a strongly type language. For example, assigning a Java™ string to a Java™ StringBuffer raises this error. Define the target variable with a type corresponding to the assigned value.
-6632	`Cannot find symbol symbol-name, location: category type-name.` The symbol used does not exist. This occurs typically when referencing a Java™ class with an invalid name in the class path, or when referencing a class member that does not exist. Check the symbol names used in the instruction.
-6633	`primitive-type cannot be dereferenced.` An expression references a method or a field with a primitive Java™ type, but primitive types are not classes and therefore do not have methods or fields. For example, you try to call a method with a symbol defined as integer or short in Java™: DISPLAY java.lang.Short.MAX_VALUE.foo Review the code using the symbol defined with a primitive type.
-6634	`Incompatible or corrupted database dictionary 'database-name'.` The .sch database schema 'database-name' contains incompatible type definitions or is corrupted. Re-generate the .sch file with the fgldbsch tool by using the correct command line options to generate compatible types.
-6636	`Invalid usage of NULL in an expression` The compiler detected an `NULL` constant in an expression that will always evaluate to `NULL` or FALSE. For example, when writing `IF var == NULL THEN`, the program flow will never enter in the `IF` block.
-6774	`The license 'license-num' is no more valid. Please contact your vendor.`

Number	Description
	The license number is no longer valid. Contact your vendor to obtain a new license number.
-6780	`Invalid license request format.` The request sent to the license manager was not recognized. Check that the version of the license manager is compatible with the runtime system.
-6781	`Incompatible License Manager (flmprg) version. The minimum version required is version-num.` The license manager is too old and is not compatible with the current runtime system. Call the support center to get a new version of the license manager.
-6783	`The license number 'license-num' is invalid. Please, contact your vendor.` The license number could not be validated by the license server. Call the support center to get a new license number.
-6784	`The license 'license-num' has expired. Please, contact your vendor.` The license is time limited and it has expired. Call the support center to get a new license number.
-6785	`CPU limit exceeded. Please, contact your vendor.` The license is CPU limited and the system has more CPUs as allowed. Call the support center to get a new license number.
-6786	`Report Writer token creation failed.` Check permissions on the lock/token directory (in FGLDIR or FLMDIR).
-6787	`This GRW license requires a DVM license with a valid maintenance date.` GRW licenses with the option 'DVM under maintenance' require that the DVM maintenance key expiration date not be expired. Update the DVM maintenance key.
-6788	`Cannot get GRW report token information.` Contact your support center.
-6789	`The installed license is invalid and cannot be used by this product.` The current license is not valid for the product you have installed. Contact your support center to get a license corresponding to the current installed product.

Number	Description
-6802	`Can not open Database dictionary 'schema-name'. Run schema extraction tool.` The schema file does not exist or cannot be found. If the schema file exists, verify that the filename is spelled correctly, and that the file is in the current directory or the FGLDBPATH environment variable is set to the correct path. If the file does not exist, run the database schema extraction tool to create a schema file.
-6803	`A grammatical error has been found at 'line-number', expecting token-name.` This is a generic message for errors.
-6804	`'form-name' form compilation was successful.` This is an information message indicating that the form was compiled without problem.
-6805	`Open Form 'form-name', Bad Version: version-1, expecting: version-2.` You have compiled your form with a version of the form compiler that is not compatible with that used for compiling the other source code. Compile your form file and related source code files using the same or compatible versions of the compilers.
-6807	`The label 'label-name' could not be used as column-title.` The form file defines an invalid TABLE column title. Check for column titles which are not corresponding to column positions.
-6808	`The widget 'widget-name' can not be defined as array.` The form file defines an item which is used as a matrix column. Review your form definition.
-6809	`The layout tag 'tag-name' is invalid, expecting: token-name.` The form compiler detected an invalid layout tag specification. Review your form definition.
-6810	`The attribute 'attribute-name' is invalid for item type 'type-name'.` The form compiler detected an invalid attribute definition for this item type. Review your form definition and check for invalid attributes.
-6811	`Syntax error near 'token-1', expecting token-2.` A general syntax error message that indicates the location of the problem code and what code was expected. Review your code and make the necessary corrections.
-6812	`Unterminated char constant.`

Number	Description
	The form compiler detected an unterminated character constant. Review your form definition and check for missing quotes or double-quotes.
-6813	`The element 'element-name' conflicts with group-box 'group-name'.` You have used the same name for an element and for a group-box. Review your form definition and ensure that the names used are unique.
-6814	`All members of the SCREEN RECORD 'screen-record-name' must reference the same Table or ScrollGrid.` The shown screen record references multiple tables or scrollgrids in your form file. Review your form definition and use one unique table for a given screen record.
-6815	`Invalid indentation in between braces.` The LAYOUT section of your form defines an invalid indentation. Review your form definition and check for corresponding indentations.
-6817	`TABLE container defined without a SCREEN RECORD in the INSTRUCTION section.` The minimum value of the defined attribute must be lower than the maximum value. Review your code and make the necessary corrections.
-6818	`Min value must be lower that Max value.` The minimum value of the defined attribute must be lower than the maximum value. Review your code and make the necessary corrections.
-6819	`Number of elements in the SCREEN RECORD must match the number of columns in TABLE container.` The elements defined in the screen record differs from the columns used for the TABLE container. Review your form definition and add missing table columns to the screen record, order does not matter.
-6820	`ScrollGrid and/or Group layout tags cannot be nested.` The form definition has nested ScrollGrid and/or Group layout tags. These tags cannot be nested. Review your form definition and make the necessary corrections.
-6821	`HBOX tags cannot be used for ARRAYS.` The form definition is using an HBOX tag for an array, which is not permitted. Review your form definition and make the necessary corrections.
-6822	`Escaped graphical characters are not accepted in GRID sections.` You try to use Text User Interface graphics in the new GRID container.

Number	Description
	This is not allowed, use GROUPs instead.
-6823	`Close tag does not have a matching tag above.` The form definition has a close tag without a prior matching open tag. Open tags and close tags must match. Review your form definition file and make the necessary corrections.
-6824	`The table 'table-name' is empty.` The form layout defines a table layout tag identified by tablename , but nothing was found directly under this table which could be a column or a column title. Append columns to the table layout region.
-6825	`The tag 'tag-name' overlaps with table 'table-name'.` In the form layout, tagname overlaps the layout region of tablename and makes it invalid. Move or remove tagname , or redefine the layout region of tablename .
-6826	`Checked value must be different from unchecked value for field 'field-name'.` The VALUECHECKED and VALUEUNCHECKED attributes have the same value. This makes no sense because these attributes define the values corresponding to the checked and unchecked states of a checkbox. Use different values for these attributes.
-6827	`Duplicated item key found for field 'field-name'.` The ITEMS attribute of field fieldname defines item keys with the same value. Check ITEMS attribute and use unique key values. Note that '' and NULL are equivalent.
-6828	`The attribute attribute-name must belong to a column of a TABLE.` A form item uses an attribute that references a form field which is not defined or does not belong to the TABLE. Check the ATTRIBUTES section for invalid column references.
-6829	`The column column-name referenced by the attribute-name attribute must belong to the TABLE.` A form item uses an attribute that references a form field which is not defined or does not belong to the TABLE. Check the ATTRIBUTES section for invalid column references.
-6830	`Not implemented (yet): feature-name` The feature or syntax you are using is not implemented yet. This feature cannot be used in the Genero version you have installed.
-6831	`At least one member of the SCREEN RECORD 'screen-record-name' must not be a PHANTOM field.`

Number	Description
	A screen record is defined with form fields that are all defined as PHANTOM fields. At least on screen record field must not be a PHANTOM field.
-6832	`Repeated screen tags 'tag-name' are misaligned, must align on X or Y.` The layout defines multiple tags with the same name, but these are not properly aligned in the X or Y direction. Edit the form file and make sure that repeated tags are correctly aligned.
-6833	`Invalid TREE definition: the field 'field-name' must be an EDIT or LABEL.` The form defines a TREE container with the field column defined with a wrong item type. Replace the item type by EDIT or LABEL.
-6834	`Invalid TREE definition: the field 'field-name' must be defined for the SCREEN RECORD.` The form defines a TREE container with an invalid field set. Check that mandatory fields such as node name, parent id and node id fields are defined.
-6835	`The fields specified in the THRU option appear in the reverse order.` The form defines a screen record by using the THRU or THROUGH keyword, but the first field is defined after the last field in the ATTRIBUTES section. Exchange the field names specified in the screen record definition, or review the declaration order in ATTRIBUTES.
-6836	`Invalid TREE definition: the attribute 'attribute-name' conflicts with id or parentid.` The .per form defines a TREE with invalid configuration. You have probably used the same field for the named attribute and for IDCOLUMN or PARENTIDCOLUMN. Review the form definition and configure the TREE properly. You must use dedicated columns for the attributes mentioned.
-6837	`Invalid AGGREGATE definition: must be located below a table column.` The .per form defines an AGGREGATE form item with a field tag that is not aligned under a table column field tag. Review the table layout and make sure that all aggregate fields are properly aligned and placed below column tags.
-6838	`This area is reserved for AGGREGATEs.` The .per form defines a TABLE with aggregate fields, but not all aggregate fields are declared with the AGGREGATE item type. Review the field definitions in the ATTRIBUTES section.

Number	Description
-6839	`The screen tag 'tag-name' can not be defined in a TABLE.` The .per form defines a TABLE with columns using different field tag names, an no AGGREGATE field is defined. Review columns of the TABLE, each field tag of a given column must use the same tag name, except if you want to define an AGGREGATE field.
-6840	`Columns with AGGREGATE must have type EDIT.` The form field table column corresponding to the aggregate field must be of type EDIT.
-6841	`FORM not contain TOPMENU or TOOLBAR.` The form layout includes an external form specification file containing a TOPMENU or a TOOLBAR. Remove these sections from the included form file.
-6842	`FORM is out of date.` The form layout includes an external form specification file the was compiled with a older version.
-6843	`A resizable SCROLLGRID requires the definition of exactly one template.` A resizable SCROLLGRID (WANTFIXEDPAGESIZE=NO) must define a single row template.
-6844	`None form-field in resizable SCROLLGRID.` Elements in a resizable SCROLLGRID (WANTFIXEDPAGESIZE=NO) can only be form fields.
-6845	`The display field label 'field-name' has already been defined.` The field item tag is defined several times in different containers of the LAYOUT section.
-6846	`The screen tag '`*`tagname`*`' can not be defined in a SCROLLGRID` A SCROLLGRID contains a layout tag such as <TABLE > or <TREE >. This is not allowed.
-6847	`TABINDEX has to be unique` Some elements of the form define the same TABINDEX. Review the ATTRIBUTE section and make sure that all TABINDEX values are unique.
-6848	`All TABLE columns must have the same size.` In a TABLE or TREE container, all columns must be defined with the same width and height in the LAYOUT section.
-8000	`Dom: Node not found.` The node could not be found in the current document. Review your code.
-8001	`Dom: Invalid Document.`

Number	Description
	The document passed to the DOM API is not a valid document. Review your code.
-8002	`Dom: Invalid usage of NULL as parameter.` NULL cannot be used at this place. Review your code.
-8003	`Dom: A node is inserted somewhere it doesn't belong.` You try to insert a node under a parent node which does not allow this type of nodes. Check for the possible nodes and review your code.
-8004	`Sax: Invalid hierarchy.` The SAX handler encountered an invalid hierarchy. Make sure parent/child relations are respected.
-8005	`Deprecated feature: feature-name` The feature you are using will be removed in a next version. A replacement for the feature is normally available.
-8006	`The string resource file 'file-name' cannot be found.` The string file shown could not be found. Check if file exists and if path is valid.
-8007	`The string resource file 'file-name' cannot be read.` The string file shown could not be read. Check if file exists and if user has read permissions.
-8008	`There is no string text defined for the 'key-name' string key.` The runtime system could not find a string resource corresponding to the shown key. Check if the key is defined in one of the resource files.
-8009	`String resource syntax error near 'token-name', expecting token.` The string file compiler detected a syntax error. Check for invalid syntax in the **.str** file.
-8012	`Duplicate string key 'key-name' ( file-name : line ) IGNORE LINE.` The string file compiler detected duplicated string keys. Review the .str file and remove duplicated keys.
-8013	`The string file 'file-name' can not be opened for writing.` The string file compiler could not write to the specified string file. Make sure the user has write permissions and file name is valid.
-8014	`The string file 'file-name' can not be read.`

Number	Description
	The runtime system could not read from the specified string file. Make sure the user has read permissions.
-8015	`Field ( field-name ) in ON CHANGE clause not found in form.` The field used in the ON CHANGE clauses was not found in the form specification file. Make sure the field name of the ON CHANGE clause matches a valid form field.
-8016	`You cannot have multiple ON CHANGE clauses for the same field.` It is not possible to specify multiple ON CHANGE clauses using the same field. Remove unnecessary ON CHANGE clauses.
-8017	`SFMT: Invalid % index used.` The format string is not valid. Check for invalid % positions.
-8018	`SFMT: Format error.` The format string is not valid. Check for invalid % positions.
-8020	`Multiple ON ACTION clauses with the same action name appear in the statement.` It is not possible to specify multiple ON ACTION clauses using the same action name. Remove unnecessary ON ACTION clauses.
-8021	`Multiple ON KEY clauses with the same key name appear in the statement.` It is not possible to specify multiple ON KEY clauses using the same key. Remove unnecessary ON KEY clauses.
-8022	`Dom: Cannot open xml-file.` The file could not be loaded. Check file name and user permissions.
-8023	`Dom: The attribute 'attribute-name' does not belong to node 'node-type'.` You try to set an attribute to a node which does not have such attribute. This is not allowed, review your code.
-8024	`Dom: Character data can not be created here.` You try to create a text node under a node which does not allow such nodes. This is not allowed, review your code.
-8025	`Dom: Cannot set attributes of a character node.` You try to set attributes in a text node.

Number	Description
	This is not allowed, review your code.
-8026	`Dom: The attribute 'attribute-name' can not be removed: the node 'node-type' belongs to the user-interface.` You try to remove a mandatory attribute from an AUI node. You can only change the value of this attribute, try 'none' or an empty string.
-8027	`Sax: can not write.` The SAX handlers could not write to the destination file. Make sure the file path is correct and the user has write permissions.
-8029	`Multiple inclusion of the source file 'file-name'.` The preprocessor detected that the specified file was included several times by the same source. Remove unnecessary file inclusions.
-8030	`The full path to the source file 'file-name' is too long.` The preprocessor does not support very long file names. Rename the file.
-8031	`The source file 'file-name' cannot be read.` The preprocessor could not read the file specified. Make sure the use has read permissions.
-8032	`The source file 'file-name' cannot be found.` The preprocessor could not find the file specified. Make sure the file exists.
-8033	`Extra token found after 'directive-name' directive.` The preprocessor detected an unexpected token after the shown directive. Review your code and make the necessary corrections.
-8034	`feature-name : This feature is not implemented.` This preprocessor feature is not supported. Review your code and make the necessary corrections.
-8035	`The macro 'macro-name' has already been defined.` The preprocessor found a duplicated macro definition. Review your code and make the necessary corrections.
-8036	`A &else directive found without corresponding &if, &ifdef or &ifndef directive.` The preprocessor detected an unexpected **&else** directive.

Number	Description
	Review your code and make the necessary corrections.
-8037	`A &endif directive found without corresponding &if, &ifdef or &ifndef directive.` The preprocessor detected an unexpected **&endif** directive. Review your code and make the necessary corrections.
-8038	`Invalid preprocessor directive & name found.` The preprocessor directive shown in the error message does not exist. Review your code and check valid macros.
-8039	`Invalid number of parameters for macro-name.` The number of parameters of the preprocessor macro shown in the error message does not match de number of parameters in the definition of this macro. Review your code and check for the number of parameters.
-8040	`Lexical error: Unclosed string.` The compiler detected an unclosed string and cannot continue. Review your code and make the necessary corrections.
-8041	`Unterminated condition &if or &else.` The preprocessor found an un-terminated conditional directive. Review the definition of this directive.
-8042	`The operator '##' can only be used with identifiers and numbers. token is not allowed.` The preprocessor found an invalid usage of the ## string concatenation operator. Review the definition of this macro.
-8043	`Could not run FGLPP, command used: command` The compiler could not run the preprocessor command shown in the error message. Make sure the preprocessor command exists.
-8044	`Lexical error: Unclosed comment.` The compiler detected an unclosed comment and cannot continue. Review your code and make the necessary corrections.
-8045	`This type of statement can only be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY, CONSTRUCT or MENU statement.` This statement has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, END CONSTRUCT, or END MENU. Review your code and make the necessary corrections.

Number	Description
-8046	`This type of statement can only be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT statement.` This statement has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT. Review your code and make the necessary corrections.
-8047	`Invalid use of 'dialog'. Must be used within an INPUT, INPUT ARRAY, DISPLAY ARRAY or CONSTRUCT statement.` The predefined keyword DIALOG has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END INPUT ARRAY, END DISPLAY ARRAY, or END CONSTRUCT. Review your code and make the necessary corrections.
-8048	`An error occurred while preprocessing the file 'file-name'. Compilation ends.` The Genero BDL preprocessor could not parse the whole source file and stopped compilation. Review the source code and check for not well formed & preprocessor macros.
-8049	`The program cannot ACCEPT (INPUT|CONSTRUCT|DISPLAY) at this point because it is not immediately within (INPUT|INPUT ARRAY|CONSTRUCT|DISPLAY ARRAY) statement.` ACCEPT XXX has not been used within a valid interactive statement, which must be terminated appropriately with END INPUT, END PROMPT, or END INPUT ARRAY. Review your code and make the necessary corrections.
-8050	`Dom: Invalid XML data found in source.` ACCEPT DISPLAY has not been used within a valid DISPLAY ARRAY statement, which must be terminated with END DISPLAY ARRAY. Review your code and make the necessary corrections.
-8051	`Sax: Invalid processing instruction name.` The om.SaxDocumentHandler.processingInstruction() does not allow invalid processing instruction names such as 'xml'. <?xml ..?> is not a processing instruction, it is reserved to define the XML file text declaration. You must use another name.
-8052	`Illegal input sequence. Check LANG.` The compiler encountered an invalid character sequence. The source file uses a character sequence which does not match the locale settings (LANG). Check source file and locale settings.
-8053	`Unknown preprocessor directive 'directive-name'.` The preprocessor directive shown in the error message is not a known directive.

Number	Description
	Check for typo errors and read the documentation for valid preprocessor directives.
-8054	`Unexpected preprocessor directive.` The preprocessor encountered an unexpected directive. Remove the directive.
-8055	`The resource file 'file-name' contains unexpected data.` The XML resource file shown in the error message does not contain the expected nodes. For example, you try to load a ToolBar with ui.Interface.loadActionDefaults(). Check if the XML file contains the node types expected for this type of resource.
-8056	`XPath: Unclosed quote at position integer.` The XPath parser found an unexpected quote at the given position. Review the XPath expression.
-8057	`XPath: Unexpected character 'char' at position pos.` The XPath parser found an unexpected character at the given position. Review the XPath expression.
-8058	`XPath: Unexpected token/string 'token-name' at position pos.` The XPath parser found an unexpected token or string at the given position. Review the XPath expression.
-8059	`SQL statement or language instruction with vendor proprietary syntax.` The compiler found an SQL statement which is using a database specific syntax. This statement will probably not run on other database servers as the current. Review the SQL statement and use standard/common syntax and features.
-8060	`Spacer items are not allowed inside a SCREEN sections.` The form contains spacer items in a SCREEN section, while these are only allowed in LAYOUT. Review the form specification file.
-8061	`A TABLE row should not be defined on multiple lines.` All columns of a row in a TABLE container must be in a single line. Use a SCROLLGRID if you want to show row cells on multiple lines.
-8063	`The client connection timed out, exiting program.` The runtime system could not establish the connection with the front-end after a given time. This can for example happen during a file transfer, when the front-end takes too much time to answer to the runtime system. Check that your network connection is working properly.
-8064	`File transfer interrupted.`

Number	Description
	An interruption was caught during a file transfer. File could not be transferred, you need to redo the operation.
-8065	`Network error during file transfer.` An socket error was caught during a file transfer. Check that your network connection is working properly.
-8066	`Could not write destination file for file transfer.` The runtime system could not write the destination file for a transfer. Make sure the file path is correct and check that user has write permissions.
-8067	`Could not read source file for file transfer.` The runtime system could not read the source file to transfer. Make sure the file path is correct and check that user has read permissions.
-8068	`File transfer protocol error (invalid state).` The runtime system encountered a problem during a file transfer. A network failure has probably raised this error.
-8069	`File transfer not available.` File transfer feature is not supported. Make sure the front-end supports file transfer.
-8070	`The localized string file 'file-name' is corrupted.` The shown string resource file is invalid (probably invalid multibyte characters corrupt the file). Check for locale settings (LANG), make sure the .str source uses valid characters and recompile it.
-8071	`'symbol-name' is already defined.` The form file defines several elements of the same type with the same name. Review the form file and use unique identifiers.
-8072	`Statement must terminate with ';'.` An ESQL/C preprocessor directive is not terminated with a semicolon. Add a semicolon to the end of the directive.
-8073	`Invalid 'include' directive file name.` An include preprocessor directive is using an invalid file name. Check the file name.
-8074	`A &elif directive found without corresponding &if, &ifdef or &ifndef directive.` The preprocessor found an **&elif** directive with no corresponding **&if** .

Number	Description
	Add the **&if** directive before the **&elif**, or remove the **&elif** .
-8075	`The compiler plugin name could not be loaded.` fglcomp could not load the plugin because it was not found. Make sure the plugin exists and can be loaded.
-8076	`The compiler plugin name does not implement the required interface.` fglcomp could not load the plugin because the interface is invalid. Check if the plugin corresponds to the version of the compiler.
-8077	`The attribute 'attribute-name' has been defined more than once.` The variable attribute shown in the error message was defined multiple times. Review the variable definition and remove duplicated attributes.
-8078	`The attribute 'attribute-name' is not allowed.` The variable attribute shown in the error message is not allowed for this type of variable. Review the possible variable attributes.
-8079	`An error occurred while parsing the XML file.` The runtime system could not parse an XML file, which is probably not using a valid XML format. Check for XML format typos and if possible, validate the XML file with a DTD.
-8080	`Could not open xml file.` The specified XML file cannot be opened. Make sure the file exists and has access permissions for the current user.
-8081	`Invalid multibyte character has been encountered.` A compiler found an invalid multibyte character in the source and cannot compile the form or module. Check locale settings (LANG) and verify if there are no invalid characters in your sources.
-8082	`The item 'item-name' is used in an invalid layout context.` The form item name is used in a layout part which does not support this type of form item. This error occurs for example when you try to define a BUTTON as a TABLE column. Review your form definition file and use correct item types.
-8083	`NULL pointer exception.` The program is using calling a method thru an object variable which is NULL. You must assign an object reference to the variable before calling a method.

Number	Description
-8084	`Can't open socket: description` The channel object failed to open a TCP socket. See the description for more details. Make sure the IP address and port are correct.
-8085	`Unsupported mode for 'open socket'.` You try to open a channel with an unsupported mode. See channel documentation for supported modes.
-8086	`The socket connection timed out.` Socket connect could not be established and timeout expired. Check all network layers and try again.
-8087	`File error in BYTE or TEXT readFile or writeFile.` File I/O error occurred while reading from or writing to a file. Verify the file name, content and access permissions.
-8088	`The dialog attribute 'attribute-name' is not supported.` A dialog instruction was declared with an ATTRIBUTES clause containing an unsupported option. Review the ATTRIBUTES clause and remove unsupported option.
-8089	`Action 'action-name' not found in dialog.` You try to use and action name that does not exist in the current dialog. Verify if name of the action is defined by an ON ACTION clause.
-8090	`Field 'field-name' already used in this DIALOG.` The DIALOG instruction binds the same field-name or screen-record multiple times. Review all sub-dialog blocks and check the field-names / screen-records.
-8091	`The clause 'clause-name' appears more than once.` You have defined the same dialog control block multiple times. For example, AFTER ROW was defined twice. Remove the un-necessary control blocks.
-8092	`At least one field for this INPUT ARRAY must be editable.` An INPUT ARRAY is executed on fields that are read-only. At least one field must be editable and active. Review the form specification file or check that at least one field is active.
-8093	`Multi-range selection is not available in this context.` You try to use multi-range selection but it is not possible in the current dialog type. Disable this feature.

Number	Description
-8094	`Multi-range selection is not available in this context.` You try to use multi-range selection but it is not possible in the current dialog type. Disable this feature.
-8095	`Cannot change selection flag for this range of rows.` An attempt of selection flag modification with DIALOG.setSelectionRange() failed because the range is out of bounds or because there is no multi-range selection available in this context. Make sure you can use multi-range selection, and check the start and end index of the range.
-8096	`General SQL Warning, check SQLCA.SQLERRD[2] or SQLSTATE.` The last SQL statement has generated an SQL warning setting the SQLCA.SQLAWARN flags. Program execution can continue. However, you should take care and check the native SQL code and the SQL message in SQLERRMESSAGE.
-8097	`Value too large to fit in a TINYINT.` The TINYINT data type can accept numbers with a value range from -128 to +127. To store numbers that are outside this range, redefine the column or variable to use the SMALLINT or INTEGER type.
-8098	`ON FILL BUFFER conflicts with DISPLAY ARRAY as a tree.` The DISPLAY ARRAY instruction is using a treeview as decoration, but it implements also an ON FILL BUFFER trigger to do paged mode. The paged mode is not possible when using a treeview, because all rows of visible nodes are required (i.e. the dialog cannot display a tree only with a part of the dataset). To populate dynamically the array for a treeview, use the ON EXPAND to add new nodes and ON COLLAPSE to remove nodes.
-8099	`The form 'form-name' is incompatible with the current runtime version. Rebuild you forms.` The .42f form was probably compiled with an earlier version as the current runtime system. Recompile the form with the fglform compiler corresponding to the current fglrun.
-8100	`Attempt to access a closed dialog.` A call to a DIALOG class method is done with a dialog object that has terminated. Review the program logic and call the DIALOG methods only for active running dialogs.
-8101	`The TABLE column tag 'tag-name' appears multiple times in the row definition.` A TABLE column can only be used once in the row definition, you have probably repeated the same screen tag by mistake.

Number	Description
	Modify the TABLE row definition in the layout section in order to use each column only once.
-8102	`Syntax error in preprocessor directive.` The source file contains a preprocessor macro with an invalid syntax. Check the preprocessor manual page and fix the syntax error.
-8103	`The source and destination file name of a file transfer must not be NULL or empty.` The program is doing an fgl_getfile() or fgl_putfile() and the source or destination file name is NULL or empty. Provide a valid file name for both source and destination parameters.
-8104	`Cannot read from TUI: system-error .` A program running in text mode (FGLGUI=0) failed to read from console input stream. Check the console/terminal settings.
-8105	`Not found.` This message displayed by the runtime system when a record was not found. It can be displayed in different contexts, for example when searching a record in a list with the built-in search feature.
-8106	`Field ( field-name ) in ON ACTION INFIELD not found in form.` The field name used in an ON ACTION INFIELD action handle could not be found in the form. Make sure you are using the correct field name and field prefix (table name or screen record name).
-8107	`FGL_LENGTH_SEMANTICS environment variable is invalid. Valid values are BYTE and CHAR` The value specified in the FGL_LENGTH_SEMANTICS environment variable must be BYTE or CHAR.
-8108	`Subdialog dialog-name: already active` The sub-dialog is already in use.
-8109	`JSON parse error: description` Verify the input string passed to the JSON parsing function. See the description for more details.
-8110	`JSON stringify error: description` The JSON serialization failed. See the description for more details.
-8111	`Can not happen: description`

Number	Description
	The runtime system encounters an unexpected situation. The message is displayed to the user, but the program flow will continue. This unexpected situation must be fixed by programmers.
-8112	`Illegal argument.` The runtime system intruction, function or object method does not expect the value passed as argument. This can for example occur when calling the `Array.sort()` method with an invalid array-record member name.
-8113	`The actions DETAILACTION and DOUBLECLICK must be different.` The `DETAILACTION` and `DOUBLECLICK` attributes are used in `DISPLAY ARRAY` to configure a table decoration and behavior. These attributes cannot define the same action.
-8114	`Completer item list too long. The list must not contain more than 50 items.` The array passed to the setCompleterItems() dialog method is too long, reduce the list.
-8115	`Character to boolean conversion error.` The array passed to the setCompleterItems() dialog method is too long, reduce the list.
-8116	`Illegal context.` The current instruction is used on a wrong context.
-8117	`'##' cannot appear at start of macro expansion.` The preprocessor operator ## must join two identifiers (a ## b).
-8118	`'##' cannot appear at end of macro expansion.` The preprocessor operator ## must join two identifiers (a ## b).
-8119	`'#' is not followed by a macro parameter.` The preprocessor operator # must be followed by a parameter of the macro.
-8120	`File transfer: copy file to file-name failed.` The runtime system could not copy the specified file.
-8121	`File transfer: remove file file-name failed.` The runtime system could not delete the specified file.
-8122	`File transfer: touch file file-name failed.` The runtime system could not touch the specified file.
-8123	`\x used with no following hex digits.` The `\xNN` character code is malformed.
-8124	`hex escape sequence out of range.`

Number	Description
	The `\xNN` character code contains an invalid hexadecimal value.
-8125	`File transfer: create symbolic link file-name failed.` The file transfer required a symbolic link that could not be created.
-8126	`Image to font mapping: Font file file-name not found.` The font file could not be found, check FGLIMAGEPATH environment variable.
-8127	`Image to font mapping: Format error in file file-name.` The image to font mapping file contains errors.
-8128	`Image to font mapping: Cannot open file file-name.` The image to font mapping file could not be found, check FGLIMAGEPATH environment variable.
-8200	`apidoc: parameter name 'param-name' is invalid.` The compiler has detected a comment error while extracting the source documentation: The @param variable name is not in the list of parameters in the next FUNCTION definition. Check the function parameter name.
-8201	`apidoc: tag missing: @param param-name.` The compiler has detected a comment error while extracting the source documentation: There is a missing @param tag that should describe a parameter of the next FUNCTION definition. Check the function parameter name.
-8202	`apidoc: invalid tag name @ tag-name.` The compiler has detected a comment error while extracting the source documentation: The @ tag-name tag is not a known tag name. Check for typo errors in the tag name.
-8300	`Cannot load java shared library. Reason: system-error` The runtime system could not load the JVM shared library (or DLL). Make sure that a JRE is installed on the machine and check the environment (LD_LIBRARY_PATH on UNIX™ or PATH on Windows™).
-8301	`Cannot create java VM.` The runtime system could load the JVM shared library (or DLL), but could not initialize the Java™ VM with a call to JNI_CreateJavaVM(). Check that the Java™ requirements and resources needs to create a Java™ VM.
-8302	`Array element type is not a Java type.` The fglcomp compiler detected a Java™ Array definition which is not using a Java™ type for the elements.

Number	Description
	Review the DEFINE statement and use a Java™ type.
-8303	`Java is not supported.` The platform you are using does not support a recent Java™ version required by Genero. You cannot use the Java™ interface in this operating system, you must review your source code and remove all Java™ related parts.
-8304	`Cannot assign a value to final variable 'variable-name'.` The program tries to set a Java™ class variable which is not writable. Review the program logic.
-8305	`The Java variable 'variable-name' can not be used here.` The program tries to use a Java™ class variable in an invalid context. For example, a Java™ class variable is used in an INPUT instruction. Review the program logic and use a regular Genero BDL variable.
-8306	`Java exception thrown: java-exception-text.` A Java™ exception has been thrown while executing Java™ code. Check the exception text and review the code.
-8307	`Java object required.` A Java™ object reference is expected by the instruction. This error typically occurs in a CAST() or INSTANCEOF(). Check the expression used in the instruction and make sure it references a Java object.
-8400	`module.name has private access.` An instruction references a module function or module variable which is declared as private. Make the function or variable public in the imported module.
-8401	`Reference to name is ambiguous.` A function or variable referenced without the module prefix, but exists in several imported modules. This error can also be printed by the compiler for Java™ calls. Add the module prefix before the object name to remove the ambiguity.
-8402	`Cyclic IMPORT FGL involving module.` Some modules are importing each other and introduce a cyclic reference which is impossible to resolve. Extract common language elements into a new module.
-8403	`Module name does not exist.` The module name to be imported could not be found.

Number	Description
	Make sure the module name matches the file name.
-8404	`Module name has not been imported.` A statement is referencing module name which has not been imported. Import the module before usage.
-8405	`category-name qualifier-name.symbol-name has not been defined.` The symbol identified by qualifier-name.symbol-name cannot be found. For example, a START REPORT or SUBDIALOG is referencing a report or sub-dialog symbol with module prefix, but the symbol is not found in the specified module. You must import the module defining the referenced symbol.
-8406	`The function 'function-name' has not been defined. This conflicts with IMPORT FGL.` The function name is referenced in the compiled module, but none of the imported modules define that function. You must import the module containing the function.
-8407	`The type of the parameter 'param-name' is not an SQL type: cannot be inserted into a temporary table used for this report.` The REPORT parameter name is defined with a BDL type that has no SQL equivalent and thus cannot be used to create the temporary table needed to sort rows for a two-pass report. Define the parameter with an SQL-compatible type (CHAR, VARCHAR, INTEGER, DECIMAL, etc).
-8408	`ON ACTION action-name conflicts with ON action-name.` The dialog block defines conflicting ON ACTION and ON triggers, defining the same actions. For example, an ON ACTION delete is defined within a dialog block that is also defining an ON DELETE trigger. Review the dialog actions, if you want to use ON triggers defining actions.
-8409	`The action action-name shadows another action with the same name.` The dialog defines ON ACTION blocks using the same action name at different levels (dialog, sub-dialog and field level). Use different action names when a conflict occurs.
-8410	`The symbol 'symbol-name' is not a DIALOG.` The symbol referenced is not defined as a DIALOG subdialog block.
-8500	`The Genero Mobile pcode size limit has been reached.` Contact your vendor for details.
-8501	`Modules compiled with Genero require a Genero license at runtime.`

Number	Description
	Contact your vendor for details.
-9000	`Value not allowed for this XML attribute.` Remove the value for this attribute or see the "Mapping between simple BDL and XML data types" section.
-9001	`Value mandatory for this XML attribute.` Set a value to the XML attribute. See the "Mapping between simple BDL and XML data types" section.
-9002	`Cannot set the XML attribute, because only one XSD attribute is allowed per definition.` Select the unique appropriate XSD data type.
-9003	`XML Attribute only allowed on a BDL TYPE.` Remove the XML attribute or change your BDL DEFINE instruction into a BDL TYPE definition.
-9004	`XML Attribute is not allowed on a type definition.` Remove the XML attribute or change your BDL TYPE definition into a BDL DEFINE instruction.
-9005	`XML Attribute XSTypeNamespace cannot be set without attribute XSTypeName.` Add a XSTypeName attribute.
-9006	`XML Attribute is only allowed on a simple data type definition.` Remove the XML attribute or change your RECORD or ARRAY into a simple BDL data type.
-9007	`XML Attribute is only allowed on a BDL RECORD definition.` Change your BDL variable definition into a RECORD.
-9008	`XML Attribute is only allowed on a one dimentional array definition.` Remove the XML attribute or use a one dimentional array.
-9009	`Attributes XMLAttribute, XMLElement, XMLAny and XMLBase are exclusives.` Choose only one of the above available choices.
-9010	`Attributes XMLChoice, XMLAll, XMLSequence, XMLSimpleContent and XSComplexType are exclusives.` Choose only one of the above available choices.
-9011	`Attribute XSTypeName has been defined twice with the same value XML attribute and the same XSTypeNamespace value, but not the same definition.`

Number	Description
	Define a unique (XSTypeName,XSTypeNamespace) couple for your program.
-9012	`XMLName or XMLNamespace not allowed on nested XMLChoice variable.` Remove the XMLName and XMLNamespace attributes.
-9013	`XMLName or XMLNamespace not allowed on nested XMLSequence variable.` Remove the XMLName and XMLNamespace attributes.
-9014	`Unrecognized XML attribute value.` Review the available values for this XML attibute.
-9015	`XML Attribute is only supported on a member of a record.` Remove the XML attribute.
-9016	`XML Attribute is only supported on a record's member when XMLChoice is defined.` Remove the XML attribute.
-9017	`XML Attribute is only supported on a record's member when XMLSimpleContent is defined.` Remove the XML attribute.
-9018	`XML Attribute not supported on this simple type.` Remove the XML attribute or change your BDL type definition.
-9019	`Attribute XMLTypeNamespace cannot be set without attribute XMLTypeName.` Set XMLTypeName attribute.
-9020	`XMLSimpleContent attribute supports only XMLAttribute and XMLAnyAttribute attributes.` Remove the unallowed XML attributes.
-9021	`Attribute XMLBase has been defined more than once in the BDL record.` Set only one XMLBase attribute.
-9022	`Attribute XMLSelector has been defined more than once in the BDL record.` Set only one XMLSelector attribute.
-9023	`XML Attribute cannot be set with other attributes.` Remove all the other XML attributes.
-9024	`Attribute XMLSelector is missing in the BDL record.`

Number	Description
	Set the XMLSelector attribute on one of the record member.
-9025	`Attribute XMLBase is missing in the BDL record.` Set the XMLBase attribute on one of the record member.
-9026	`Nested XML attribute cannot be defined on a BDL TYPE.` Remove the Nested XML attribute.
-9027	`Nested XML attribute cannot be defined on root variable.` Remove the Nested XML attribute.
-9028	`Invalid parameter.` See the documentation about the function paramaters.
-9029	`Parameters of a published RPC Web Service operation must be a Record or NULL.` Review your parameters definition.
-9030	`Parameters of a published DOC Web Service operation must be a Record, an Array or NULL.` Review your parameters definition.
-9031	`XML Attribute is not allowed on a BDL record's member.` Remove the XML attribute or set it at the appropriate place.
-9032	`XML Attribute can only be set on a ARRAY defined inside a RECORD.` Remove the XML attribute or set it at the appropriate place.
-9033	`XML Attribute cannot be defined at first level of a variable.` Remove the XML attribute or set it at the appropriate place.
-9034	`Attributes 'XMLAttribute' are not allowed on nested sequence or choice.` Remove the XMLAttribute attribute.
-9035	`RPC Web Functions cannot have XMLList set on one of the parameters.` Put your BDL ARRAY inside a BDL RECORD.
-9036	`Attribute XMLName is mandatory on BDL variable when used as SOAP Header.` Add the XMLName attribute.
-9037	`RPC Web Functions cannot have XMLNamespace set on one of the parameters.`

Number	Description
	Remove the XMLNamespace attribute.
-9038	`XSComplexType attribute allows only attributes with one optional nested list or nested record.` Set only one XMLOptional attribute for all nested record members.
-9039	`XMLName or XMLNamespace not allowed on nested XMLAll.` Remove XMLName and XMLNamespace.
-9040	`Nested XML Attribute is not allowed on an array.` Remove the XML attribute
-9041	`XMLBase Attribute allows only one additional XSD attribute.` Set a unique XSD attribute.
-9042	`XML Attribute value is not allowed on a BDL record's member.` Set the appropriate value to the specified XML attribute.
-9043	`Unsupported facet constraint for the BDL type.` Check the available facet constraint in "Mapping between simple BDL and XML data types" section.
-9044	`Invalid value for facet constraint 'constraint-name'.` Check the available facet constraint value. See XML facet constraint attributes on page 2529.
-9045	`Facet constraint attributes cannot be defined without a XSD simple type attribute.` Add the appropriate XSD attribute.
-9046	`Facet XSDLength and XSDMinLength or XSDMaxLength cannot be used together.` Select only one of the above attributes.
-9047	`XML Attribute not allowed on BDL objects.` Remove the XML attribute.
-9048	`Attribute XMLName cannot be set with XMLAny or XMLAnyAttribute.` Remove the XMLName attribute.
-9049	`XML Attribute not allowed on members of xmlchoice='inherited' records.` Remove the XML attribute.
-9050	`Parameter with public qualifier not allowed.` Remove the PUBLIC instruction.

Number	Description
-9051	`Parameters of published Web Service operations must be variables in global or modular scope.` Move your variables to a GLOBALS instruction or to modular scope.
-9052	`A published Web service header must be a variable in global or modular scope.` Move your Web service header to a GLOBALS instruction or to modular scope.
-9053	`Web service function with private qualifier not allowed.` Remove the PRIVATE instruction.
-9054	`Web service function must be a string literal.` You cannot use a variable for your web service function name.
-9055	`XML Attribute is not allowed on an array definition.` Remove the XML attribute.
-9056	`Attribute XMLAny has been defined more than once per BDL record.` Use only one XMLAny attribute in a BDL RECORD.
-9057	`Attribute XMLAnyAttribute has been defined more than once per BDL record.` Use only one XMLAnyAttribute attribute in a BDL RECORD.
-9058	`Attribute XMLList and XMLAnyAttribute are exclusives.` Use only one of the above XML attributes.
-9059	`Element of BDL array with XMLAnyAttribute must be a BDL record containing three variables for the namespace, name, value of type STRING.` Example: DEFINE arr DYNAMIC ARRAY OF RECORD ns, name, value STRING END RECORD
-9060	`XML Attribute is only allowed on dynamic arrays.` Change your BDL ARRAY into a DYNAMIC ARRAY.
-9061	`XML Attribute cannot be set inside a nested record.` Remove the XML attribute.
-9062	`Attribute XMLAttribute is not allowed after attribute XMLAnyAttribute.` Move the record member with XMLAnyAttribute attribute to the last position.
-9063	`A published Web service fault must be in global or modular scope.` Move your variables to a GLOBALS instruction or to modular scope.

Number	Description
-9064	`Attribute XMLName is mandatory on the BDL variable when used as Fault.` Set the XMLName attribute.
-9065	`Colon not allowed for XML attribute value.` Remove the colon.
-9066	`XML Attribute is only allowed on a root variable.` Remove the attribute or move it to the root variable.

-9067	`Bad W3CEndPointReference definition.`
	Review your RECORD definition. It should match this structure:
	`RECORD ATTRIBUTES(W3CEndpointReference)`
	`  address STRING, -- The location of the Web Service (for ex: URL)`
	`  ref RECORD`
	`  ... (other members defining the state)`
	`  END RECORD`
	`END RECORD`
	See com.WebService.CreateStatefulWebService on page 2012.

-9068	`Invalid state BDL variable, only simple variables or W3CEndpointReference record allowed.` Check that "state" parameter TYPE of function com.WebService.CreateStatefulWebService is correct. Its type must be a simple type definition or a W3CEndPointReference RECORD.
-9069	`Registered HTTP variable error.` Check that the BDL variable match the definition set in com.WebService.registerInputHTTPVariable or com.WebService.registerOutputHTTPVariable.
-10098	`Incorrectly formed hexadecimal value.` You try to load data with LOAD or locate a BYTE variable with a file contained malformed hexadecimal values. Check the file content and fix the typos before loading again.
-10099	`Invalid delimiter. Do not use '\' or hex digits (0-9, A-F, a-f).` You try to LOAD or UNLOAD data with an invalid field delimiter. Change the field delimiter to a valid character such as | (pipe) or ^ (caret).
-15500	`Internal runtime error occurred in WS server program.` Contact your support center.
-15501	`Cannot create WS operation because the given function is not defined.`

Number	Description
	Verify that the name of the BDL function of fgl_ws_server_publishFunction() is correct.
-15502	`Invalid WS-function declaration, no parameters allowed.` Verify that the BDL function has no input and no output parameters.
-15503	`Operation name is already used in the current web service.` You must change the name of the Web-Function operation in the function fgl_ws_server_publishFunction().
-15504	`WS server port already used by another application.` You must change the port number in the function Fgl_ws_server_start().
-15505	`Some BDL data types are not supported by XML.` Verify that all exposed functions don't contain one of the following data types: • DATETIME beginning with MINUTE • DATETIME beginning with SECOND • INTERVAL beginning with YEAR and/or MONTH
-15511	`Invalid fgl_ws_set/getOption() parameter.` Verify that the option flag of the fgl_ws_setOption()/ fgl_ws_getOption() function exists.
-15512	`WS input record not defined.` Verify that the name of the input record on the fgl_ws_server_publishFunction() exists.
-15513	`WS output record not defined.` Verify that the name of the output record on the fgl_ws_server_publishFunction() exists.
-15514	`The port value from the FGLAPPSERVER environment variable or from the parameter of the fgl_ws_server_start() function is not a numeric one.` Verify that the port value contains only digits. See fgl_ws_server_start()
-15515	`No application server has been started at specified host.` Verify that FGLAPPSERVER contains the right host and port where the application server is listening.
-15516	`No more licenses available.` Contact your support center.
-15517	`Current runner version not compatible with the Web Services Extension.` Install the right version of the Genero BDL.
-15518	`The input namespace of your Web function is missing.` Add a valid input namespace in fgl_ws_server_publishFunction().

Number	Description
-15519	`The output namespace >namespace of your Web function is missing.` Add a valid output namespace in fgl_ws_server_publishFunction().
-15520	`Cannot load a certificate or private key file.` Verify that each ws.ident.security FGLPROFILE entries contain a valid security identifier.
-15521	`Cannot find a certificate in the Windows key store.` Verify that each ws.ident.security FGLPROFILE entries contain a valid Windows™ security identifier.
-15522	`Cannot load the Certificate Authorities file.` Verify that the security.global.ca FGLPROFILE entry contains the correct Certificate Authorities filename.
-15523	`Cannot create the Certificate Authorities from the Windows key store.` Verify that you have enough rights to access the Windows™ key store.
-15524	`Cannot set the cipher list.` Verify that all ciphers in the list are valid ones and supported by openssl.
-15525	`Unable to reach the HTTP proxy.` Verify that the proxy.http.location FGLPROFILE entry contains the correct HTTP proxy address.
-15526	`Unable to reach the HTTPS proxy.` Verify that the proxy.https.location FGLPROFILE entry contains the correct HTTPS proxy address.
-15527	`Unknown HTTP proxy authenticate identifier.` Verify that the proxy.http.authenticate FGLPROFILE entry contains a valid HTTP authenticate identifier.
-15528	`Unknown HTTPS proxy authenticate identifier.` Verify that the proxy.https.authenticate FGLPROFILE entry contains a valid HTTP authenticate identifier.
-15529	`Cannot create a HTTP authenticate configuration.` Verify that all authenticate logins and passwords are correctly set.
-15530	`Cannot create an encrypted HTTP authenticate configuration.` Verify that all authenticate logins and encrypted passwords are correctly set.
-15531	`Cannot create a server configuration.` Verify that all ws.ident.url FGLPROFILE entries are correctly set.

Number	Description
-15532	`Unknown server configuration security identifier.` Verify that all ws.ident.security FGLPROFILE entries contain a valid Security identifier.
-15533	`Unknown server configuration authenticate identifier.` Verify that all ws.ident.authenticate FGLPROFILE entries contain a valid HTTP Authenticate identifier.
-15534	`Invalid self object.` Contact your support center.
-15535	`Cannot perform operation due to invalid parameters.` Check all parameters against the built-in classes documentation.
-15536	`Service registration failed, see SQLCA.SQLERRM for more details.` Check the following : • A service of the same name already exists • The namespace of the service is missing • A header cannot have the same name and namespaces as an operation
-15537	`Cannot create web service, see SQLCA.SQLERRM for more details.` Check that the service has a valid name and namespace.
-15538	`Cannot create Web operation, see SQLCA.SQLERRM for more details.` Check that operation name and namespace are valid according to the style (Document or RPC).
-15539	`Cannot publish Web operation, see SQLCA.SQLERRM for more details.` Check that input or output headers have previously been created.
-15540	`Published BDL function not found, see SQLCA.SQLERRM for more details.` Check that BDL function to be publish exists.
-15541	`Published BDL function not correctly defined, see SQLCA.SQLERRM for more details.` Check that BDL function has no input or output parameters.
-15542	`Input parameter of published operation error. See SQLCA.SQLERRM for more details.` Contact your support center.
-15543	`Output parameter of published operation error. See SQLCA.SQLERRM for more details.` Contact your support center.

Number	Description
-15544	`Web Service header configuration error, see SQLCA.SQLERRM for more details.` Verify that a one-way function do not have an output header.
-15545	`Service is already registered. You cannot modify a service after it has been registered.` Check that you do not call a service modifier method on a service after registration.
-15546	`Invalid option.` Check the option name according to documentation.
-15547	`Unsupported web service operation.` Verify if a Document style operation does not perform SOAP Section5 encoding.
-15548	`Bad URI.` Check that URI passed to a HttpRequest or TcpRequest is valid.
-15549	`HTTP runtime exception, see SQLCA.SQLERRM for more details.` Contact your support center.
-15550	`XML runtime exception, see SQLCA.SQLERRM for more details.` Contact your support center.
-15551	`WSDL generation failed.` Contact your support center.
-15552	`Charset conversion exception, see SQLCA.SQLERRM for more details.` Change server charset response via a HTTP accept header or change you application locale.

-15553	`TCP runtime exception, see SQLCA.SQLERRM for more details.`
	If detailed message is 'The TCP connection has been interrupted', then check that your network was working properly and that the INT_FLAG was not set to TRUE.
	When working with a Web Service application, this can the result of a COM error. Check in FGLWSDEBUG to see whether it was shut down on the client or server side.
	For example:
	`WS-DEBUG (IO ERROR)`
	`Class: TCPConnection::atomicReceive()`
	`Msg: TCP input stream shut down.`
	`Code: 104`
	`WS-DEBUG END=`
	You can find the 104 code in `/usr/include/asm-i386/errno.h` (depending on your system).

Number	Description
	In this example it correspond to: `#define ECONNRESET 104 /* Connection reset by peer */` • Review the WSDL and see if what we send to the server is correct • Review the server log and see why it has ended the connection
-15554	`Index is out of bound.` Check your index maximum value.
-15555	`Unsupported request-response feature.` Check the streaming operations order or for invalid usage. For example, in function readTextRequest(), the incoming request can be read only once, so processing the incoming message while sending the response is not allowed.
-15556	`No request was sent.` Check that you called one of the doRequest(), doXmlRequest() or doTextRequest() method before to call getResponse() or getAsyncResponse().
-15557	`Request was already sent.` Check that you do not call twice one of the doRequest(), doXmlRequest() or doTextRequest() method.
-15558	`Waiting for a response.` Check that you do not perform a new request before reading the response of previous one.
-15559	`No stream available.` Check that you do not call a method to read on a stream that has not yet been created.
-15560	`Streaming is over.` Check that you do not read a streaming response that was closed.
-15561	`Streaming in progress.` Check that you do not call twice beginXmlResponse() without a call to endXmlResponse().
-15562	`Streaming not yet started.` Check that you do not call endXmlRequest() or endXmlResponse() without a beginXmlRequest() or beginXmlResponse().
-15563	`Streaming already started.` Check that you do not call twice beginXmlRequest() or beginXmlResponse().
-15564	`Unexpected peer stream was shutdown.` The peer closed connection during reading operation.
-15565	`Cannot return incoming request, see SQLCA.SQLERRM for more details.`

Number	Description
	Check detailed message.
-15566	`Operation failed, see SQLCA.SQLERRM for more details.` Check the parameter for invalid data.
-15567	`Parameter cannot be NULL.` Check that the parameter is not NULL
-15568	`BDL callback function not found, see SQLCA.SQLERRM for more details.` Check that BDL callback function exists.
-15569	`BDL callback function requires one input and one output parameter, see SQLCA.SQLERRM for more details.` Check BDL callback parameters according to documentation.
-15570	`Web Service fault error. See SQLCA.SQLERRM for more detail.` Contact your support center.
-15571	`Stateful Service error. See SQLCA.SQLERRM for more detail.` Contact your support center.
-15572	`Access denied lock error.` Either the file is already locked, or the application does not have the write access right to the given path.
-15573	`HTTP Multipart error : description.` One of the methods of the COM multipart API has failed. See the description for more details. Contact your support center if the error detail does not provide the information needed to fix the error.
-15574	`Cannot load Certificate Authorities from path : path.` The certificate could not be found according to the current FGLPROFILE configuration. Check the certificate authority settings as described in: HTTPS configuration on page 2440.
-15575	`Incoming request has been closed : reason` The GAS has disconnected the web service server, for example while calling the `com.WebServiceEngine.GetHTTPServiceRequest` or `com.WebServiceEngine.HandleRequest` methods. Use a `TRY/CATCH` block to trap this error, as described in com.WebServiceEngine.GetHTTPServiceRequest on page 2027.
-15576	`Invalid TCP IP version.` The FGLPROFILE configuration parameter `ip.global.version` defines a value different from valid possible values (4 and 6).

Number	Description
-15577	`Unknown network interface name : name.` The FGLPROFILE configuration parameter `ip.global.v6.interface.name` defines a network interface that does not exist.
-15578	`Request canceled by user.` The HTTP request initiated by a `com.HTTPRequest.getResponse()` method has been canceled by the user.
-15598	`XML deserialization error.` The WSDL contract does not match the BDL variable definition. Check that BDL variables are correctly generated according to the WSDL.
-15599	`Internal error, should not happen.` Contact your support center.
-15600	`Operation failed.` Check method for invalid parameters according to documentation.
-15601	`Name cannot be NULL.` Check that name parameter is not NULL.
-15602	`Namespace cannot be NULL.` Check that namespace parameter is not NULL.
-15603	`Prefix cannot be NULL.` Check that prefix parameter is not NULL.
-15604	`Value cannot be NULL.` Check that parameter is not NULL according to documentation.
-15605	`Node cannot be NULL.` Check that node parameter is not NULL.
-15606	`Text cannot be NULL.` Check that text parameter is not NULL.
-15607	`Target of a processing instruction cannot be NULL.` Check that target parameter is not NULL.
-15608	`Name of an entity reference cannot be NULL.` Check that entity name parameter is not NULL.
-15609	`XPath expression cannot be NULL.` Check that xpath parameter is not NULL.

Number	Description
-15610	`Filename cannot be NULL.` Check that filename parameter is not NULL.
-15611	`Document cannot be NULL.` Check that document parameter is not NULL.
-15612	`DTD string cannot be NULL.` Check that dtd parameter is not NULL.
-15613	`Stax cannot be NULL.` Check that stax parameter is not NULL.
-15614	`Malformed XML name.` Check that xml name is well-formed.
-15615	`Malformed XML string.` Check that xml string is well-formed.
-15616	`Malformed XML prefix.` Check that xml prefix is well-formed.
-15617	`Malformed XML namespace.` Check that xml namespace is well-formed.
-15618	`Bad validation type.` Check validation type parameter.
-15619	`No XML schema found.` Check that a valid XML schema is used for validation.
-15620	`No DTD schema found.` Check that a DTD schema is present in XML document.
-15621	`Feature or option cannot be NULL.` Check that parameters are not NULL.
-15622	`Feature or option is unsupported.` Check option or feature name according to documentation.
-15623	`Feature or option value is invalid.` Check option or feature validity according to documentation.
-15624	`Node is not part of the document.` Check that node belong to the same XML document.

Number	Description
-15625	`Node does not have the correct parent node.` Check that node to remove belongs to the right parent node.
-15626	`Node is already linked to another node.` Check that node is not already attached to another node.
-15627	`Cannot add a node to itself.` Check that node to add is not itself.
-15628	`Index is out of bounds.` Check index maximum value.
-15629	`StaxWriter runtime exception: reason` See SQLCA.SQLERRM for more details and check the reason for the error.
-15630	`StaxReader runtime exception: reason` See SQLCA.SQLERRM for more details and check the reason for the error.
-15631	`Serializer runtime exception: reason` See SQLCA.SQLERRM for more details and check the reason for the error.
-15632	`Document loading runtime exception, check xml.DomDocument.getErrorDescription() for more details.` Check detailed message of dom document.
-15633	`Document saving runtime exception, check xml.DomDocument.getErrorDescription() for more details.` Check detailed message of dom document.
-15634	`Invalid encoding.` Check encoding value.
-15635	`PublicID of a DTD cannot be set with a SystemID.` Check DTD node creation
-15636	`Undefined namespace prefix in the XPath expression.` Check an undeclared prefix used in XPath expression.
-15637	`XPath expression error.` Check XPath expression.
-15638	`A namespace in the XPath namespace list is missing.` Check for an undeclared namespace used in XPath expression
-15639	`XPath function has two mandatory parameters.`

Number	Description
	Check parameters according to documentation.
-15640	`Internal XPath error.` Contact your support center.
-15641	`Invalid XPath namespace.` Check namespace value passed to XPath method.
-15642	`Unable to load schema.` Check XML schema parameters in DomDocument.setFeature().
-15643	`Schemas are malformed or inconsistent.` Check XML schema validity in DomDocument.setFeature().
-15644	`URI is malformed.` Check that URI is well-formed according to documentation.
-15645	`Protocol layer needs a new try to complete operation.` Sax writer close operation requires a new request to complete previous one.
-15646	`Charset conversion error.` Check fglrun LANG and system locale.
-15647	`Unable to load xml security library.` Contact your support center.
-15648	`Xml security operation failed. See SQLCA.SQLERRM for more detail.` Check detailed message.
-15649	`URL cannot be null.` Check if XML-Security URL is NULL.
-15650	`CryptoX509 cannot be null.` Verify that CryptoX509 object has been correctly instantiated.
-15651	`CryptoKey cannot be null.` Verify that CryptoKey object has been correctly instantiated.
-15652	`Bad signature transformation.` Check transformation URL validity passed to appendReferenceTransformation()
-15653	`Bad signature digest.` Check digest URL validity passed to createReference().
-15654	`Bad signature node.`

Number	Description
	Check XML-Signature node passed to CreateFromNode().
-15655	`Bad key type.` Check key identifier URL.
-15656	`Bad key usage.` Verify usage of CryptoKey object passed to setKeyEncryptionKey() or setKey().
-15657	`Bad XPathFilter2 type, only intersect, subtract or union allowed.` Verify type used in a XPathFilter2 transformation.
-15658	`Bad derived key URL.` Check derived key identifier URL.
-15699	`Internal error, should not happen.` Contact your support center.
-15700	`Called operation failed, see SQLCA.SQLERRM for more details.` See SQLCA.SQLERRM for details on why the operation failed.
-15701	`Invalid parameter.` Check that your security library function has the correct parameters.
-15702	`File access denied.` Check that your security library function has the permissions to access to the file.
-15703	`File does not exist.` Check that the file exist on your system for the security library function to access.
-15704	`Algorithm not supported.` Check that the algorithm is in the supported list for security library function. See security.Digest.CreateDigest on page 2294.
-15705	`Invalid current object.` Check that the context for security library function is correctly initialized. See security.Digest.CreateDigest on page 2294.
-15799	`Internal security error.` Contact your support center.

Web services

Create a Web service client or server with Genero BDL.

The Genero APIs for creating Web services can be found in the Library section of this manual. See The com package on page 2009 and The xml package on page 2103.

•

General

These topics provide you with an introduction to Genero Web Services and the information needed to get working with the latest version of the software.

- Introduction to Web Services on page 2400
- SOAP Web Services basics on page 2404
- RESTful Web Services basics on page 2416
- Getting started and examples on page 2416
- Debugging on page 2416
- Platform-specific notes on page 2416
- Known issues on page 2419
- Legal Notices on page 2419

Introduction to Web Services

This topic provides an introduction to Web Services with the Genero Web Services Package (GWS). It is intended to help those using GWS for the first time to understand basic Web Services concepts, and to quickly start their development with the Genero tools.

Concepts

Web services are a standard way of communicating between applications over an intranet or Internet. They define how to communicate between two entities:

- A server that exposes services
- A client that consumes services

Server usage example

A server exposes a"StockQuotation" service that responds to an operation "getQuote". For the "getQuote" operation, the input message is a stock symbol as a string, and the output message is a stock value as a decimal number.

The "getQuote" operation is a function written in Genero BDL, and it is published on the server. This function retrieves the stock value for the stock symbol passed in, and returns it.

Client usage example

The Web service client application calls the function as if it were a local function. It passes the stock symbol in to the function, and stores the returned value in a variable. If the Web Service operation is named **WebService_StockQuotation_getQuote** and the local variable is **svalue**, the Web Service is called as follows:

```
LET svalue = WebService_StockQuotation_getQuote( "MyStockSymbol" )
```

Service Oriented Architecture (SOA) and web services

Service Oriented Architecture (SOA) is a philosophy of how to connect systems and exchange data to solve business problems. Rather than concentrating on a specific task or transaction, SOA addresses how to use data from various sources, reduce human work, and mitigate the effects of change in a business process and its supporting systems.

The SOA defines the services to be provided; Web Services are the means of implementing those services. Web Services provide a ***platform-neutral technology*** to connect multiple systems in a flexible manner, where the platform-neutrality helps insulate the SOA from changes to the underlying systems.

Web Services work by answering requests for information and returning well defined, structured XML documents. Because XML is simple text and Web Services can be invoked via the hypertext transfer protocol (HTTP), it does not matter what platform runs the Web Service, or what platform receives the XML document.

An SOA's resilience to change is accomplished by adhering to good Web Services design practices:

- Build a Web Service that performs a specific task
- Have a rigid structure for the data

Web Services tell exactly how to ask for the information in an XML document written using the Web Services Descriptive Language (WSDL). This self-describing document describes the service the Web Service will perform and how to form the request for its data. Each Web Service must have an associated WSDL document, so that developers and applications know what to expect from the Web Service, and how to invoke it.

Migrating to SOA and web services

Developing an SOA and moving to Web Services is an iterative and evolutionary process. It requires work and diligent design. When switching to Web Services from another integration method, it is recommended to initially focus on shorter term business benefits, targeting an SOA and Web Services project that has tangible goals with measurable benefits.

Once an SOA contains some useful services, these services can be arranged together in a workflow that automates a business process. Web Services can be reused to answer new questions, and implemented as new business services in an SOA. A well-defined Web Service does not contain business logic or business process information. Because each Web Service in an SOA can be called individually to perform a specific task, they can be arranged (orchestrated) together to perform many different business functions. As a result, companies with a mature SOA in place can change business processes through configuring of the orchestration software as opposed to programming individual links between systems.

Planning a web service

When creating a Web Service, you not only have to think of the task at hand, but you should also consider growth. You likely want the Web Service to be flexible; to be able to handle different types of input. Prepare the Web Service for what is *probable*. Developers should think bigger than the needs of a single application. You should think of reusing existing services, and think how your services can be reused by others.

Security will likely play a larger role than it did previously with existing in-house application infrastructures using programmed links between systems; you will need to become versed in security issues.

Keep the goals of SOA in mind when designing and coding Web Services: **Flexibility**. **Reusability**. **Interoperability**.

Genero web services extension

The Genero Web Services Extension (GWS) is an extension to the Genero Business Development Language. It installs within the Genero Business Development Language directory. The fglgws package includes both Genero Business Development Language and Genero Web Services.

The Genero Application Server is required to manage your Web Services in a deployment environment. It is not required for Web services development, unless you are interested in testing deployment issues.

> **Important:** When programming a Web service, your applications must include `IMPORT com` at the top of each module. This imports the Genero Web Services Extension library named `com`:

```
IMPORT com
```

Web services standards

Web services are platform-independent and programming language-independent. The World Wide Web consortium defines the Web services standards. For more information about these standards, refer to the "Web services" section of their web site at http://www.w3.org. The Genero Web Services package supports the WSDL 1.1 specification of March 15, 2002 as well as some previous specifications.

The standards involved in what is commonly called "Web services" include XML, XML Schema, SOAP, WSDL, and HTTP.

XML

XML (eXtensible Markup Language) defines a machine-independent way of exchanging data. For example, an XML representation of the following BDL data structure:

```
DEFINE Person
RECORD Attribute (XMLName="Person")
   FirstName  VARCHAR(32) Attribute (XMLName="FirstName"),
   LastName   VARCHAR(32) Attribute (XMLName="LastName"),
   Age        INTEGER Attribute (XMLName="Age")
END RECORD
```

Could be:

```
<Person>
 <FirstName>John</FirstName>
 <LastName>Smith</LastName>
 <Age>35</Age>
</Person>
```

The record definition allows you to specify XML attributes for data types. This feature was added with Genero 2.00.

XML schema

XML Schema defines the elements, entities, and content model of an XML document. For example, for the example document shown in the topic XML on page 2402, the schema could say that the XML document contains an element "Person", and that each "Person" contains one and only one element "FirstName", "LastName", and "Age". The XML Schema has additional capabilities, such as data type control and content restrictions.

An XML Schema allows an XML document to be validated for correctness.

SOAP

SOAP (Simple Object Access Protocol) is a high-level communication protocol between a server and the client. It defines the XML data flow between the server and the client. The "StockQuote" service mentioned in the Concepts section exchanges messages using the following syntax:

Request

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <getQuote>
         <stockSymbol>MyCompany</stockSymbol>
```

```
      </getQuote>
   </soap:Body>
</soap:Envelope>
```

Response

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <getQuoteResponse>
         <stockValue>999.99</stockValue>
      </getQuoteResponse>
   </soap:Body>
</soap:Envelope>
```

SOAP relies on a lower-level protocol for the transport layer.

Genero Web Services use SOAP over HTTP, and can also perform low-level XML and TEXT over HTTP communications on the client side. This allows communication between applications using the core Web technology, taking advantage of the large installed base of tools that can process XML delivered plainly over HTTP, as well as SOAP over HTTP.

WSDL

The WSDL (Web Services Description Language) file describes the services offered by a server. It contains:

- The description of the operations offered by the server, and each operation's input and output messages.
- The location of the SOAP server.
- Internal connection and protocol details (transport layer, encoding, namespaces, and so on).

A WSDL description is sufficient to provide all the information required to communicate with the SOAP server.

Genero Web Services package provides a tool, fglwsdl, that enables Genero client applications to obtain the WSDL description of a Web Service.

HTTP

HTTP (Hypertext Transfer Protocol) is the set of rules for exchanging files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web.

Web services style options

The Web Services Style options available for created Genero Web services are WS-I (Web Services Interoperability organization) compliant:

- **RPC Style Service (RPC/Literal)** is generally used to execute a function, such as a service that returns a stock option.
- **Document Style Service** (**Doc/Literal**) is generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document.
- **RPC Style Service (RPC/Encoded)** is the legacy style, now provided for backwards compatibility. You most likely will not be using the RPC/Encoded style to create new Web Services.

SOAP Web Services basics

Topics in this section are only relevant for Web Services based on the SOAP protocol.

Migration notes
Migrating GWS server applications
What you need to know when migrating GWS server applications.

- Migrating GWS server runners only on page 2404
- Migrating GWS server runners and using new APIs on page 2404
- Operation publication restrictions on page 2405

Migrating GWS server runners only

There is no need to create a special runner for Genero Web Services 2.x. Instead, the GWS 2.x library is imported into your applications. If you want to migrate your existing 1.x GWS Server application to 2.x to avoid the need for a special runner, as well as to take advantage of any bug fixes, take the following steps:

1. Add the following statement at the top of any .4gl module where you have used GWS 1.3x functions:

```
import com
```

2. Compile and re-link your GWS Server application (.42r).

This imports the new GWS **com** library, and ensures that any GWS 1.3x functions that you have used will be compatible. Your existing Genero 1.3x Client applications, as well as third-party Client applications, will continue to work.

Migrating GWS server runners and using new APIs

If you want to take advantage of the new features and simplify future migrations, you can migrate your Genero Web Services (GWS) Server runner and also use the new GWS 2.x APIs. All the 1.3x publishing functions for all the operations in your application must be replaced with 2.x publishing functions. Since this does not change the interface, all existing Genero 1.3x Client applications, as well as third-party Client applications, will continue to work.

Since 1.3x only supports RPC-Encoded style services, you must use the RPC style functions of the new 2.x APIs as the replacement functions, with **setInputEncoded** and **setOutputEncoded** set to true. And, you cannot add XML attributes to the records used as Web Service function parameters.

To replace the fgl_ws_server_publishfunction() statement in an existing GWS Server application; for example:

```
CALL fgl_ws_server_publishfunction(
  "EchoInteger",
  "http://tempuri.org/webservices/types/in", "echoInteger_in",
 "http://tempuri.org/webservices/types/out", "echoInteger_out",
 "echoInteger")
```

1. Add this statement at the top of each module:

```
import com
```

2. Define variables for the WebService and WebOperation objects:

```
DEFINE serv  com.WebService
DEFINE op    com.WebOperation  -- Operation of a WebService
```

3. Create the GWS Server object:

```
LET serv = com.WebService.CreateWebService(
  "EchoInteger",
  "http://tempuri.org/webservices")
```

4. Use the 2.x publishing functions for each operation:

```
LET op = com.WebOperation.CreateRPCStyle(
  "echoInteger",
  "EchoInteger",
  echoInteger_in,
  echoInteger_out)
CALL op.setInputEncoded(true)
CALL op.setOutputEncoded(true)
CALL serv.publishOperation(op,NULL)
```

5. Compile and re-link your GWS Server application (.42r)

GWS 2.x also allows your Server application (.42r) to contain multiple services. If you would like 2.x and 1.3x GWS to coexist in the same .42r executable, replace the existing publishing 1.3x functions.

Operation publication restrictions

If you use a variable as the name of the function to publish, you will have an error message at compile time.

For example:

```
com.WebOperation.CreateRPCStyle(test,"Add",add_in,add_out)
```

Where **test** is a string variable, add_in and add_out are input and output records.

At compile time, you get the error message:

```
error:(-9054) Web service function must be a string
```

The function name in parameter can only be a string literal not a string variable.

Since version 2.21, FGL has introduced the concept of PUBLIC/PRIVATE function, there is a risk for a user to publish private functions. Private functions are not always available at runtime.

As a workaround you can add a switch depending on the function name value in order to call the appropriate publication API with the name in a string literal such as following sample:

```
CASE function_name
  WHEN "Operation1"
    LET op = com.WebOperation.CreateDocStyle(
               "Operation1","Operation1",op1_in,op1_out)
  WHEN "Operation2"
    LET op = com.WebOperation.CreateDocStyle(
               "Operation2","Operation2",op2_in,op2_out)
  OTHERWISE
    DISPLAY "ERROR"
END CASE
```

In Java™ or in .NET you cannot publish different numbers of operations for a same service, everything is done at compile time. For instance, when you publish a web service in Java™, only the public methods will be published as operation of the service. There is no way to add or remove some methods at runtime. The only way you have is to create another Java™ class.

Be aware that if you dynamically change the service operations names you are creating a different service, which might be confusing for the web service client.

Enhance the GWS server application to be WS-I compliant (recommended)

> **Important:** You must be able to change all the Client applications that access your migrated Genero Web Services (GWS) Server.

If you use the Literal styles now available in GWS 2.x for your Web Service, your application will be WS-I compliant. However, the migration techniques still use the RPC/Encoded style (Only RPC/Encoded was supported in GWS 1.3x.). If you can change all the client applications that access your migrated GWS Server, we recommend that you enhance the GWS Server application to be WS-I compliant.

1. Replace the publishing functions in the GWS Server application, but omit the **setInputEncoded** and **setOutputEncoded** lines. The resulting style will be Literal.
2. The enhanced GWS Server will have a new RPC/Literal WSDL that must be used to regenerate the client stub with the fglwsdl tool:

```
fglwsdl -o NewClientstub http://localhost:8090/MyCalculator?WSDL
```

3. Compile that new client stub, and re-link it with the GWS Client application. This operation must be repeated for each Client application accessing that service.
4. Third party Client applications must also be changed to use the new WSDL.

Migrating GWS client applications
Migration from version 1.3x to 2.2x

If you use a Genero 2.2x runner for the GWS Client application, you must:

1. Regenerate the GWS Client stubs using the **-compability** option of the fglwsdl tool, so the function prototypes will be compatible:

```
fglwsdl -compatibility -o NewClientstub http://localhost:8090/
MyCalculator?WSDL
```

2. Compile the GWS Client stubs and re-link the Client application (.42r).

Migration from version 2.0x to 2.2x

You must regenerate all client stubs into your application using the fglwsdl tool.

This is mandatory because the generated code is based on the low-level COM and XML APIs and is completely different from versions prior to 2.1x; otherwise, you won't be able to execute the code.

Migration from version 2.1x to 2.2x

It is recommended to regenerate all client stubs into your application using the fglwsdl tool.

Migration from version 2.xx to 2.4x

It is recommended to regenerate all client stubs into your application using the fglwsdl tool.

If you have modified the server location at runtime via the generated global variable in your client application, you MUST apply following modification:

- Prior to version 2.40, you had something like following:

```
LET Calculator_CalculatorPortTypeLocation = "http://host:port/Calculator"
```

- Starting with version 2.40, you must have something like following:

```
LET Calculator_CalculatorPortTypeEndPoint.Address.Uri =
 "http://host:port/Calculator"
```

See Change client behavior at runtime.

Migration from version 2.xx to 3.xx

It is recommended to regenerate all client stubs into your application using the fglwsdl tool.

> **Important:** It is mandatory to regenerate the client stubs, to support fault response with HTTP error code of 200.

See Change client behavior at runtime.

WebService engine options

In the class **com.WebServiceEngine**, two options have been renamed and two options moved to a new class.

Renamed options

The **http_invoketimeout** and **tcp_connectiontimeout** options have been respectively renamed into **readwritetimeout** and **connectiontimeout**, as they are now available for either HTTP or TCP protocol. While the old option names remain for backward compatibility, using the new option names is **strongly** recommended.

Moved options

xml_ignoretimezone and **xml_usetypedefinition** options were part of the **com**.**WebServiceEngine** class. They have been moved to the class **xml.Serializer**, which groups functions on serialization.

I4GL migration guide

Migrate an I4GL web service provider to Genero

This section explains how to migrate a I4GL web service provider to a Genero application providing the same web service in order to let all clients, already accessing that service, unmodified (excepted for the hostname of course).

> **Note:** The migration will be based on the SOA zipcode demo in the I4GL package.

Step 1: Use the I4GL function and the I4GL .4cf configuration file

Use the I4GL .4cf configuration file to get all information about the I4GL web service

For example, the I4GL zipcode demo has following .4cf configuration file :

```
[SERVICE]
 TYPE               = publisher
 INFORMIXDIR        = /dbs/32bits/ifx/11.70.uc2
 DATABASE           = i4glsoa
 CLIENT_LOCALE      = en_US.8859-1
 DB_LOCALE          = en_US.8859-1
 INFORMIXSERVER     = ol_moscou1170uc2
 HOSTNAME           = moscou.strasbourg.4js.com
 PORTNO             = 9876
 I4GLVERSION        = 7.50.xC4
 WSHOME             = /dbs/32bits/ifx/11.70.uc2/AXIS2C
 WSVERSION          = AXIS1.5
 TMPDIR             = /tmp/zipcodedemo
 SERVICENAME        = ws_zipcode
 [FUNCTION]
  NAME           = zipcode_details
  [INPUT]
   [VARIABLE]NAME = pin TYPE = CHAR(10)[END-VARIABLE]
  [END-INPUT]
  [OUTPUT]
   [VARIABLE]NAME = city TYPE = CHAR(100)[END-VARIABLE]
   [VARIABLE]NAME = state TYPE = CHAR(100)[END-VARIABLE]
  [END-OUTPUT]
 [END-FUNCTION]
 [DIRECTORY]
  NAME           = /home/f4gl/fg/i4gl
  FILE           = soademo.4gl,
 [END-DIRECTORY]
[END-SERVICE]
```

Then simply copy your I4GL function without any modification into a new Genero file and add the Genero **IMPORT com** instruction at the beginning of the file.

For example, the I4GL soa demo contains the zipcode_details service (soademo.4gl)

```
IMPORT com

FUNCTION zipcode_details(pin)
  DEFINE state_rec RECORD
         pin CHAR(10),
         city CHAR(100),
         state CHAR(100)
       END RECORD,
       pin CHAR(10),
       sel_stmt CHAR(512);

  LET sel_stmt= "SELECT * FROM statedetails WHERE pin = ?";
  PREPARE st_id FROM sel_stmt;
  DECLARE cur_id CURSOR FOR st_id;
  OPEN cur_id USING pin;
  FETCH cur_id INTO state_rec.*;
  CLOSE cur_id;
  FREE cur_id;
  FREE st_id;
  RETURN state_rec.city, state_rec.state

END FUNCTION
```

Note: you may need some minor code modification for compatibility.

Step 2: Create a BDL RECORD for the input parameters

Add a new modular BDL record where all members map to one of your I4GL web service input parameter, and keep the parameter order as defined in I4gl .4cf file.

You must then specify the web service input message name via the Genero XML attribute called XMLName, and assign it to the FUNCTION NAME as defined in the I4GL .4cf file.

For example, in the I4GL zipcode demo there is only one parameter: **pin** . So add the following record at the beginning of the Genero file :

```
DEFINE zipcode_details_in RECORD ATTRIBUTES(XMLName="zipcode_details")
  pin CHAR(10)
  END RECORD
```

Note: Genero Web Services supports complex data type as input parameters.

Step 3: Create a BDL RECORD for the output parameters

Add another modular BDL record where all members map to one of your I4GL web service output parameter, and keep the parameter order as defined in I4GL .4cf file.

You must then specify the web service output message name via the Genero XML attribute called XMLName, and assign it to the FUNCTION NAME as defined in the I4GL .xcf file concatenated to response.

For example, in the I4GL zipcode demo there are two parameters: **city** and **state**. So add following record at the beginning of the Genero file:

```
DEFINE zipcode_details_out RECORD
 ATTRIBUTES(XMLName="zipcode_detailsresponse)
  city CHAR(100),
  state CHAR(100)
```

```
  END RECORD
```

Note: Genero Web Services supports complex data type as output parameters.

Step 4: Create a BDL wrapper function

Create a Genero BDL wrapper function without any parameters that will then use the input and output record created at Step 2 and 3 to call the I4GL function passing it the parameters retrieved from the records.

For example, in the I4GL zipcode demo there are 1 input and 2 output parameters. So the BDL wrapper function must use these records to call the I4GL function as following :

```
FUNCTION zipcode_details_g()
 CALL zipcode_details(zipcode_details_in.pin)
    RETURNING zipcode_details_out.city,zipcode_details_out.state
END FUNCTION
```

Step 5: Publish the wrapper function as a Genero web service

Use the COM APIs to publish the I4GL function as a web service based on I4GL .4cf configuration file to get a compatible Genero Web service.

To create a new BDL function in charge of the service publication, you will need the following elements of the I4GL .4cf configuration file:

- The name of the service that is defined in the SERVICENAME entry
- The namespace of the service that is defined as http://www.ibm.com/ concatenated to the FUNCTION NAME
- The name of the function to be published that is defined in the FUNCTION NAME entry

For example, the I4GL zipcode demo has one function published as a Doc/Literal service.

```
FUNCTION create_zipcode_details_web_service()
  DEFINE serv com.WebService
  DEFINE op   com.WebOperation

  #
  # Create the web service based on the entries of the .4cf file
  #   SERVICENAME: The name of service is 'ws_zipcode'
  #   FUNCTION NAME: The namespace of the service is built from
  #                  the base url 'http://www.ibm.com/' concatenated to
  #                  the NAME of the I4GL function 'zipcode_details'
  #
  LET serv = com.WebService.CreateWebService("ws_zipcode",
              "http://www.ibm.com/zipcode_details")

  #
  # Create and publish the Doc/Literal web function based on
  #  step 2, step 3 and step 4
  #  and from the FUNCTION NAME defined in the .4cf file
  #
  LET op = com.WebOperation.CreateDOCStyle("zipcode_details_g",
            "zipcode_details",
            zipcode_details_in,
            zipcode_details_out)
  CALL serv.publishOperation(op,NULL)

  #
  # Register the service into the SOAP engine
  #
  CALL com.WebServiceEngine.RegisterService(serv)
```

```
END FUNCTION
```

Note: I4GL supports only Doc/Literal services.

Note: Genero Web Services can contain several BDL functions in the same service. In other words, you can group several I4GL services into the same Genero service.

Step 6: Create the server

I4GL uses Axis as server for its services, but Genero has its own server programmable via the COM library. Create a new file and add the `IMPORT com` instruction at beginning of the server file, then simply create the main loop in BDL that will process any incoming HTTP request.

The port of the service defined in the I4GL .4cf configuration file (via the PORTNO entry) can be reused by setting the FGLAPPSERVER environment variable to the same value before to run the server. However, only on development or for tests, on production Genero Web services requires an application server called GAS in charge of load balancing. See the GAS documentation for more details about port configuration for deployment purpose.

For example, to migrate the I4GL zipcode demo, the service must be created in the server before run the main loop as following :

```
MAIN
  DEFINE ret INTEGER
  DEFER INTERRUPT

  # Create zipcode_details service
  CALL create_zipcode_details_web_service()

  # Start the server on port set in FGLAPPSERVER
  #    (to be set to same value as PORTNO defined in the .4cf file)
  CALL com.WebServiceEngine.Start()

  # Handle any incoming request in a WHILE loop...
  # See <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//IBM//DTD DITA IBM Concept//EN" "ibm-
concept.dtd">
<concept id="c_gws_server_tutorial_009" xml:lang="en-us">
<title>Step 5: Start the GWS server and process requests</title>
<shortdesc>Once you have registered the Web Service(s), you are ready to
 start the Genero Web
Services (GWS) Server and process the incoming SOAP requests.</shortdesc>
<prolog></prolog>
<conbody>
<p> The GWS Server is located on the same physical machine where the
 application is being executed
(In other words, where <cmdname>fglrun</cmdname> executes).</p>
<p>This is the <codeph>MAIN</codeph> program block of your application.</p>
<section><title>Define a variable for status</title><p>Define a variable
to hold the returned status of the         request:<codeblock>MAIN
  DEFINE ret INTEGER</codeblock></p><p>Call the function that you created,
 which defined and registered the service and its
operations:<codeblock>  CALL createservice()</codeblock></p></section>
<section><title>Start the GWS Server</title><p>Use the <codeph>Start</
codeph> class method of the <xref href="c_gws_ComWebServiceEngine.dita"
><codeph>WebServiceEngine</codeph></xref> class to start the
server.<codeblock>  CALL com.WebServiceEngine.Start()</codeblock></p></
section>
<section id="process_request">
<title>Process the requests</title>
<p>This example uses the <codeph>ProcessServices</codeph> method of the
 <xref
```

```
href="c_gws_ComWebServiceEngine.dita"><codeph>WebServiceEngine</codeph></
xref> class to process each
incoming request. It returns an integer representing the status. The
 parameter specifies the timeout
period (in seconds) for which the method should wait to process a service.
 The value -1 specifies an
infinite waiting
time.<codeblock>  WHILE TRUE
    # Process each incoming requests (infinite loop)
    LET ret = com.WebServiceEngine.ProcessServices(-1)
    CASE ret
      WHEN 0
        DISPLAY "Request processed."
      WHEN -1
        DISPLAY "Timeout reached."
      WHEN -2
        DISPLAY "Disconnected from application server."
        EXIT PROGRAM
      WHEN -3
        DISPLAY "Client Connection lost."
      WHEN -4
        DISPLAY "Server interrupted with Ctrl-C."
      WHEN -10
        DISPLAY "Internal server error."
        EXIT PROGRAM
      WHEN -15
        DISPLAY "Server was not started."
        EXIT PROGRAM
      OTHERWISE
        DISPLAY "ERROR: ", STATUS, SQLCA.SQLERRM
    END CASE
    IF int_flag&lt;&gt;0 THEN
      LET int_flag=0
      EXIT WHILE
    END IF
  END WHILE

  DISPLAY "Server stopped"

END MAIN</codeblock></p><note type="note">For testing purposes only, the GWS
 Server can be started in <xref
href="c_gws_server_tutorial_015.dita#c_gws_server_tutorial_015">standalone
 mode</xref>. In a
production environment, the Genero Application Server (GAS) is required to
 manage your application.
For deployment, the GWS Server application must be added to the GAS
 configuration. See <cite>Adding
Applications</cite> in the <cite>Genero Application Server User Guide</
cite>. </note></section>
</conbody>

</concept>


END MAIN
```

> **Note:** With Genero Web Services, one server can contain several services. In other words, you can put all your I4GL services into one server.

Step 7: Configure the database

Based on the DATABASE entry in the I4GL .4cf configuration file, use the Genero instruction to connect to the informix database at server startup.

For example, in the I4GL zipcode demo the service access the database called: **i4glsoa** . So add following instruction at the beginning of the server file created in step 6:

```
DATABASE i4glsoa
MAIN
...
END MAIN
```

Step 8: Compile and run the Genero service

Compile and link the 2 Genero files created above and run your Genero service. It will be directly available for any client, and provide the WSDL when requested via a HTTP GET with WSDL as query string.

Example

The Genero web service is accessible on URL: **http://hostname:9876/ws_zipcode** and can return the WSDL on URL: **http://hostname:9876/ws_zipcode?WSDL**.

```
$ fglcomp -M genero_service.4gl
$ fglcomp -M genero_server.4gl
$ fgllink -o genero_zipcode genero_service.42m genero_server.42m
$ export FGLAPPSEVER=9876
$ fglrun genero_zipcode.42r
```

Note: The hostname depends on the machine your Genero application is started.

Note: For deploying the service on production you will need the Genero application server (GAS) to load-balance the service. See the GAS documentation about Web Services when deployment is required.

Step 9: Disable Axis support of MTOM/XOP and WS-Addressing

I4GL is based on Axis web service for the SOAP layer and sends by default requests in MTOM/XOPand with support of WS-Addressing. However, Genero Web services doesn't support MTOM/XOP and WS-Addressing, therefore you have to unset both features on your Axis installation if you still want your I4GL client applications to communicate with the Genero web service after migration.

For example, the Axis installation contains a file called axis.xml where following two lines have to be removed:

```
<parameter name="enableMTOM" locked="false">true</parameter>
<module ref="addressing"/>
```

Migrate an I4GL web service consumer to Genero

This section explains how to migrate a I4GL web service consumer to a Genero application accessing the same web service. **Notice** that the migration will be based on the soa demo in the I4GL package.

Step 1: Generate the Genero web service stub from an I4GL WSDL

Use the I4GL WSDL located on the Axis server to generate the Genero web service client stub via the tool called **fglwsdl**.

For exampl, the WSDL file of the I4GL zipcode demo is located on $INFORMIXDIR/AXIS2C/services/ws_zipcode/zipcode_details.wsdl. So do following command:

```
$ fglwsdl -noFacets zipcode_details.wsdl
```

It will generate these two Genero files:

- ws_zipcode_zipcode_detailsservice.4gl

  - It contains the Genero functions to connect to the server in SOAP over HTTP.
  - Take a look into that file if you are interested into Genero HTTP and XML low-level APIs.
- ws_zipcode_zipcode_detailsservice.inc
  - It contains the Genero data types and variables used for XML serialization.
  - Take a look into that file if you are interested into Genero XML to BDL variable mapping.

    Note: Option -noFacets is required for this demo because the I4GL CHAR data type will be generated as string in Genero what can lead to XML serialization error if not present.

Step 2: Modify the Genero .inc stubs to fix wrong I4GL WSDL

The I4GL WSDL contains namespace declaration for all I4GL web service data types, but in practice the I4GL axis server doesn't care about namespaces, but Genero does. So you have to open the generated Genero .inc file and remove all attributes called **XMLNamespace** and **XSTypeNamespace** .

For example, the generated .inc file from the I4GL WSDL must be modified as following:

```
#----------------------------------------------------------------------------
# File: ws_zipcode_zipcode_detailsservice.inc
# GENERATED BY fglwsdl 101601
#----------------------------------------------------------------------------
# THIS FILE WAS GENERATED. DO NOT MODIFY.
#----------------------------------------------------------------------------

GLOBALS
...
#
# TYPE : tzipcode_details
#
TYPE tzipcode_details RECORD
  ATTRIBUTES(XMLSequence,XSTypeName="zipcode_details")
  #,XSTypeNamespace="http://www.ibm.com/zipcode_details")
   pin STRING ATTRIBUTES(XMLName="pin")
END RECORD
#----------------------------------------------------------------------------
#
# TYPE : tzipcode_detailsresponse
#
TYPE tzipcode_detailsresponse RECORD
  ATTRIBUTES(XMLSequence,XSTypeName="zipcode_detailsresponse")
  #,XSTypeNamespace="http://www.ibm.com/zipcode_details")
   city STRING ATTRIBUTES(XMLName="city"),
   state STRING ATTRIBUTES(XMLName="state")
END RECORD
...
#----------------------------------------------------------------------------
#
# Operation: zipcode_details
#
# FUNCTION: zipcode_details_g()
#   RETURNING: soapStatus
#   INPUT: GLOBAL zipcode_details
#   OUTPUT: GLOBAL zipcode_detailsresponse
#
# FUNCTION: zipcode_details(p_pin)
#   RETURNING: soapStatus ,p_city ,p_state
#
# FUNCTION: zipcode_detailsRequest_g()
#   RETURNING: soapStatus
#   INPUT: GLOBAL zipcode_details
#
```

```
# FUNCTION: zipcode_detailsResponse_g()
#   RETURNING: soapStatus
#   OUTPUT: GLOBAL zipcode_detailsresponse
#
#--------------------------------------------------------------------------------
# VARIABLE : zipcode_details
DEFINE zipcode_details tzipcode_details
  ATTRIBUTES(XMLName="zipcode_details")
  #,XMLNamespace="http://www.ibm.com/zipcode_details")
#--------------------------------------------------------------------------------
# VARIABLE : zipcode_detailsresponse
DEFINE zipcode_detailsresponse tzipcode_detailsresponse
  ATTRIBUTES(XMLName="zipcode_detailsresponse")
  #,XMLNamespace="http://www.ibm.com/zipcode_details")
END GLOBALS
```

> **Note:** Genero Web Services provides a lots of XML mapping attributes.

Step 3: Include the generated stub into your I4GL application

Add in all I4GL files calling a web service the generated .inc stub with a GLOBALS instruction.

For example, in the I4GL zipcode demo, only the **clsoademo.4gl** file uses web services. So add following line at beginning of the file :

```
GLOBALS "ws_zipcode_zipcode_detailsservice.inc"
MAIN
...
END MAIN
```

> **Note:** This allows access to the Genero global variables and data types used in the web service call, so as the Genero global **wsError** record to retrieve error codes if any.

Step 4: Modify the I4GL web service function call

The Genero Web service function name is defined in the generated .4gl file and must be used instead of the I4GL function name.

For example, in the I4GL zipcode demo, the web service function name is **cons_ws_zipcode** and must be renamed to **zipcode_details** as following:

```
FUNCTION func_cons_ws_zipcode()
   DEFINE state_rec RECORD
            pin CHAR(10),
            city CHAR(100),
            state CHAR(100)
         END RECORD;
   #
   # Genero web service status returning
   # whether web function call was successful or not
   #
   DEFINE soapstatus INTEGER

   #
   # I4GL web service function name is 'cons_ws_zipcode'
   # CALL cons_ws_zipcode("97006")
   #   RETURNING state_rec.city, state_rec.state
   # Genero web service function name is 'zipcode_details'
   CALL zipcode_details("97006")
     RETURNING soapstatus, state_rec.city, state_rec.state
   ...
END FUNCTION
```

Note: In Genero Web Services there is an additional returned parameter, **soapstatus**. If it contains 0 the operation was a success, otherwise an error occurred.

Step 5: Handle Genero web services errors

I4GL web service errors are returned on a non conventional SOAP fault what cannot be handled in Genero. However the errors are handled through the additional returned parameter **soapstatus** that must be checked after each web service call. If its value is not zero, an error has occurred and can be retrieved via the global Genero **wsError** record defined in the above generated .inc file.

Example

In the Genero Web Service you must check the soap status after each web service call:

```
FUNCTION func_cons_ws_zipcode()
   DEFINE state_rec RECORD
             pin CHAR(10),
             city CHAR(100),
             state CHAR(100)
          END RECORD;
   #
   # Genero web service status returning
   #  whether web function call was successful or not
   #
   DEFINE soapstatus INTEGER

   # Genero web service function call
   CALL zipcode_details("97006")
     RETURNING soapstatus, state_rec.city, state_rec.state
   # Check soap status for errors after zipcode_details call
   IF soapstatus<>0 THEN
      # Display error information from the server
      DISPLAY "Error:"
      DISPLAY "  code :",wsError.code
      DISPLAY "  ns   :",wsError.codeNS
      DISPLAY "  desc :",wsError.description
      DISPLAY "  actor:",wsError.action
   ELSE
      # Display results
      DISPLAY "\n ------------------------ \n"
      DISPLAY "SUPPLIED ZIP CODE: 97006 \n"
      DISPLAY " ------------------------ \n"
      DISPLAY "RESPONSE FROM WEB SERVICE \n"
      DISPLAY " ------------------------ \n"
      DISPLAY " CITY:",state_rec.city
      DISPLAY "\n STATE:",state_rec.state
      DISPLAY "\n ======================== \n"
   END IF
   ...
END FUNCTION
```

Step 6: Compile and run the Genero client

Then simply compile your modified I4GL application for Genero and execute it. Your application will then connect to the web service passing and returning the parameters as it were only simple BDL function calls.

For example, to compile your I4GL web service application for Genero, you must do the following commands:

```
$ fglcomp -M ws_zipcode_zipcode_detailsservice.4gl
$ fglcomp -M clsoademo.4gl
$ fgllink -o clsoademo.42r clsoademo.42m
 ws_zipcode_zipcode_detailsservice.42m
```

```
$ fglrun clsoademo.42r
```

Step 7: Disable Axis support of MTOM/XOP and WS-Addressing

I4GL is based on Axis web service for the SOAP layer and sends by default requests in MTOM/XOP and with support of WS-Addressing. However, Genero Web services doesn't support MTOM/XOP and WS-Addressing, therefore you have to unset both features on your Axis installation if you want your Genero client application to communicate with an I4GL web service provider.

For example, the Axis installation contains a file called axis.xml where following two lines have to be removed:

```
<parameter name="enableMTOM" locked="false">true</parameter>
<module ref="addressing"/>
```

Remark: Standalone Axis server is buggy

The I4GL standalone axis server adds an extra CR LF after the body of the SOAP HTTP post response what leads the Genero client to return the error message : `Body content bigger than expected`. This is not allowed as defined in HTTP [RFC2616].

> **Important:** Notice however that Axis works as expected if loaded from Apache server.

RESTful Web Services basics

Topics in this section are only relevant for Web Services based on the SOAP protocol.

Getting started and examples

Genero Web Services code examples are located in **FGLDIR/demo/WebServices**, where **FGLDIR** is the Genero BDL installation directory.

Debugging

The Genero Web Services library gives you the ability to log the data your Web Service application is receiving from or sending to another application by turning on the debug mode.

Debug information is written to the standard error stream of the console; if needed, it can be redirected to a file.

To turn on the debugging feature, set the FGLWSDEBUG environment variable before starting the application.

The level of debugging depends on the value set for the FGLWSDEBUG variable.

Possible values are described in the FGLWSDEBUG environment variable definition.

> **Note:** To debug a Web Service application managed by the Application Server, you have to modify the value of the FGLWSDEBUG environment variable in the Application Server configuration file. For more information, refer to the *Genero Application Server Manual* documentation.

Platform-specific notes

IBM® AIX®

- The "IBM® C++ Runtime Environment Components for AIX®" must be installed in order to use Genero Web Services Extension 2.0. See the IBM® support center for more information about downloading the component.

  > **Note:** If not installed, you will get the following error message:
  >
  > ```
  > Could not load C extension library 'com'. Reason: A file or directory
  > in the path name does not exist.
  > ```

- Due to an IBM® issue on 64-bit platforms, the openssl library is unable to open the system /dev/urandom device to generate a PRNG number.

  You must install the **Entropy Gathering Daemon (a.k.a EGD)** if you need security in your GWS application, and especially if you access a server in HTTPS.

GMI / iOS Web Services limitations
Some Web Services classes are not supported on iOS devices (GMI).

Web Services COM package

The following com classes are not supported in GMI:

- `com.Util`
- `com.TCPRequest`
- `com.TCPResponse`
- `com.WebService`
- `com.WebOperation`
- `com.WebServiceEngine` (except for `SetOption()`/`GetOption()` methods, for option `SoapModuleURI` only)
- `com.HTTPServiceRequest`

The following methods have a different behavior:

- `com.HTTPRequest.setVersion()` has no effect, the iOS HTTP stack supports HTTP 1.1 only.
- `com.HTTPRequest.getAsyncResponse()` is not working asynchronously, it works like `com.HTTPRequest.getResponse()`.
- `com.HTTPRequest.setAutoReply()` has no effect, the iOS HTTP stack does not provide an auto reply option.
- `com.HTTPRequest.setMaximumResponseLength()` has no effect, the iOS HTTP stack does not provide a maximum response length option.
- `com.HTTPRequest.setConnectionTimeout()` and `com.HTTPRequest.setTimeout()`: the max of both settings is used as timeout by the iOS HTTP stack.

Web Services XML package

The following xml classes are not supported in GMI:

> **Note:** These classes are currently not supported on iOS, as OpenSSL cannot be used to implement these classes on iOS. OpenSSL is used to implement these classes for other platforms.

- `xml.CryptoKey`
- `xml.CryptoX509`
- `xml.Signature`
- `xml.Encryption`
- `xml.KeyStore`

For all other classes of the xml package, methods using an URL parameter accept only a file URI:

- `xml.DomDocument.Load()`
- `xml.DomDocument.save()`
- `xml.StaxWriter.writeTo()`
- `xml.StaxReader.readFrom()`

If the parameter is not a file URI, these methods can raise runtime exceptions such as -15629, -15630, -15632, -15633.

SOAP protocol

GMI is not able to handle SOAP errors or faults, as the iOS API does not allow a retrieval of an HTTP response body if the server uses an HTTP code of 500. As a result, GMI will not get notified about what went wrong during a remote procedure call.

If the server returns a SOAP error or fault, GMI will raise exception -15559. Modify your call of a remote web service as follows:

```
DEFINE wsstatus INTEGER
...
   LET wsstatus = Webservice_Function_g()
   IF wsstatus==-15559 THEN
      MESSAGE "Handle generic SOAP error or fault"
   END IF
```

Web Services configuration options

GWS configuration entries of FGLPROFILE are not supported on iOS.

FGLPROFILE entries are described here: FGLPROFILE entries on page 2510.

Long running HTTP request popup

If the `com.HTTPRequest.getResponse()`/`getAsyncResponse()` methods take more than 5 seconds to complete, the GMI will show a typical iOS popup message to ask the user if the request must be canceled. If the user cancels the request, the runtime system raises an exception. A progress bar is displayed if the Content-Length for a request is available (i.e. self made uploads and most downloads)

HTTP request compression for POST/PUT

HTTP request compression for POST/PUT is not supported on iOS devices.

Multipart HTTP request

On iOS, multipart HTTP requests are not supported. See com.HTTPRequest.setMultipartType on page 2066.

Changing the SOAP client behavior at runtime

The following features have a limited usage on iOS devices:

- HTTP version protocol definition is ignored.
- Connection timeout and read/write timeout are identical.

FGLPROFILE settings for Web Services client

FGLPROFILE settings for Web Services are not supported on iOS:

- Settings for logical names is not supported.
- Settings for HTTP proxy configuration is ignored: Uses device settings.
- Settings for client authentication to server is not supported: Use HTTPRequest API instead.
- Settings for client authentication to proxy is not supported: Uses device settings.
- Settings for server certificate authority is not supported: Uses device KeyChain.

GMA / Android™ Web Services requirements

Requirements to use Web Services on Android platforms.

V3 SSL Certicates

The SSL certificates for secured servers must be of type V3: Android does not support other types of SSL certificates. When creating your own self-signed certificates (to be installed in the "Install from storage" Keystore of Android), make sure that type V3 is used.

Known issues

Forcing RPC style convention when no input message

In RPC style, the convention defines names for input messages and output messages, but if there is no input message, its name cannot be redefined.

To workaround this issue, respect RPC style convention in wsdl, or force RPC convention (on client and server side) by using the `-fRPC` option of the `fglwsdl` tool.

Variable names conflicts with library names

The `fglwsdl` tool can generate variable names conflicting with `IMPORT` library names.

For example:

```
DEFINE xml xml.DomDocument
```

will conflict with the `xml` library, if the code defines also the instruction:

```
IMPORT xml
```

Legal Notices

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software developed by CollabNet (http://www.Collab.Net/).

This product includes software developed by the University of California, Berkeley and its contributors.

This product includes software developed or owned by Caldera International, Inc

Concepts

These topics cover various Genero Web Services concepts.

- High-level and low-level web services APIs on page 2420
- SOAP features on page 2420
- Stateful web services on page 2422
- Encryption, base64 and password agent with fglpass tool on page 2429
- HTTP compression on page 2432
- SOAP multipart style requests in GWS on page 2434

High-level and low-level web services APIs

With Genero, we distinguish two type of APIs to implement web services.

- High-level web services are SOAP web services managed on server side with the high-level APIs `WebService` and `WebOperation`, or if you have generated code via the fglwsdl tool (client and server side).
- Low-level web services are implemented with `HTTPRequest`, `HTTPResponse` and `HTTPServiceRequest` classes, where you have to write all the HTTP code of your services by hand.

SOAP features

SOAP 1.1 and 1.2

Since 2.40, Genero Web Services (GWS) supports SOAP 1.2. GWS is able to communicate with Web services provided with SOAP 1.1 or SOAP 1.2.

Server side

A Genero Web Services server can deliver a service in SOAP 1.1 or SOAP 1.2 using the com.WebService.setFeature function.

For example in $FGLDIR/demo/WebServices/calculator/server/calculatorServer.4gl, the calculator server offers the service in SOAP 1.1 and SOAP 1.2.

```
LET serv = com.WebService.CreateWebService("Calculator",serviceNS)
CALL serv.setFeature("Soap1.1",TRUE)
CALL serv.setFeature("Soap1.2",TRUE)
```

Client side

A GWS client can consume a service in SOAP 1.1 or SOAP 1.2.

For example:

- To create a client that consumes the Calculator service in SOAP 1.1 use command:

```
fglwsdl -soap11 -o ws_calculator http://localhost:8090/Calculator?WSDL
```

- To create a client that consumes the Calculator service in SOAP 1.2 use command:

```
fglwsdl -soap12 -o ws_calculatorSoap12 http://localhost:8090/Calculator?
WSDL
```

Be aware to generate different clients for each SOAP versions. Even if the same operations are provided, the services are using different protocols so the underlying generated stubs are also different.

SOAP Fault

Since 2.40, Genero Web Services supports SOAP fault.

For backward compatibility, fglwsdl tool provides option **-ignoreFaults** to disable SOAP fault management.

Server side

A Genero Web Services server can throw a SOAP fault if any processing error is encountered.

To generate a SOAP fault you need to:

- create the fault variable with com.WebService.createFault()
- add it to your operation with com.WebOperation.addFault
- use it with com.WebServiceEngine.SetFaultDetail

For example in $FGLDIR/demo/WebServices/calculator/server/calculatorServer.4gl, the calculator server has a **divide_by_zero** SOAP fault. The SOAP fault is raised when you try to divide a number by zero. To generate a SOAP fault proceed as follow:

Create a SOAP fault

You define the variable to send as a SOAP fault. It can be a simple string like in this example or a complex type. Remember to assign a XMLName to the variable.

```
DEFINE divide_by_zero STRING ATTRIBUTES (XMLName="DividedByZero")
```

Then you inform the service that it can use this fault variable using function com.WebService.createFault().

```
LET serv = com.WebService.CreateWebService("Calculator",serviceNS)
CALL serv.createFault(divide_by_zero,FALSE)
```

Add the SOAP fault to an operation

A SOAP fault can be used by an operation to inform the client that an error has occurred. An operation can use different SOAP faults but **only one at a time**.

```
LET op =
 com.WebOperation.CreateRPCStyle("divide","Divide",divide_in,divide_out)
CALL op.addFault(divide_by_zero,NULL)
```

Here, the SOAP fault is added to the "divide" operation.

Send the SOAP fault

Set the values to the fault variable. The fault message is be sent to the client at the end of the operation processing.

```
LET divide_by_zero = "Cannot divide "||divide_in.a||" by zero"
CALL com.WebServiceEngine.SetFaultDetail(divide_by_zero)
```

Client side

If a SOAP fault occured the operation returns the SOAP fault number in the operation status. The SOAP fault number is defined in the generated stubs as a BDL constant prefixed with the string **FaultID**_.

> **Note:** A SOAP fault can occur in case of HTTP error 200 and 500.

For example in $FGLDIR/demo/WebServices/calculator/client/ws_calculator.inc, **Divide** operation has a SOAP fault that informs the client when a number is divided by zero.

```
# List of Soap fault constants
CONSTANT FaultID_DividedByZero = 1
...
# VARIABLE : DividedByZero
DEFINE DividedByZero STRING ATTRIBUTES(XMLName="DividedByZero",
  XMLNamespace="http://tempuri.org/")
...
# Operation: Divide
# FAULT #1: GLOBALS DividedByZero
```

You can test the operation status code accordingly and display the SOAP fault message.

For example in $FGLDIR/demo/WebServices/calculator/client/calculatorClient.4gl, when the divide operation status is 1, DividedByZero message is displayed.

```
ON ACTION divide
   CALL Divide(op1, op2) RETURNING wsstatus, result, remaind
   CASE wsstatus
      WHEN 0
         DISPLAY BY NAME result,remaind
         DISPLAY "OK" TO msg
      WHEN FaultID_DividedByZero
         DISPLAY DividedByZero TO msg
      OTHERWISE
         DISPLAY wsError.description TO msg
   END CASE
```

Stateful web services

Concept

A stateful service is a service that maintains a context between a web services client and server. It enables the service to keep trace of previous requests from that context, in order to manage different states in the web service server.

Genero Web Services supports two kinds of stateful services:

- Based on the WS-Addressing 1.0 specification to define the XML format used to convey the context from client to the server
- Based on an HTTP session cookie to convey the context from client to the server

The Genero Web Service engine uses a BDL variable defined at stateful service creation via createStatefulWebService() as service context. Use that variable to hold a service state in a database.

It is up to the BDL programmer to create, store and remove the service state in the database.

The SOAP engine is responsible for:

- Deserializing the state variable when getting an new incoming request. *The programmer can then read the state variable for any published BDL web service operation and restore the service state corresponding to that variable.*
- Serializing a new instance of the state variable in a web service response for all BDL web service operations set as session initiator via initiateSession(). *The programmer must instantiate a new state by filling the state variable and storing it into a database for further use.*

WS-Addressing 1.0 stateful services

A stateful service based on WS-Addressing uses the WS-Addressing EndpointReference type as state variable and is independent from the transport layer used. (See WS-Addressing 1.0 EndpointReferenceType). The session state is conveyed from the client to the server as WS-Addressing 1.0 reference parameters.

Server side
Perform these steps to create a WS-Addressing stateful service.
Step 1: Declare a W3CEndpointReference record to be used as state variable

This record **MUST** have:

- A mandatory member of type STRING, where you can define a different service end point URL, otherwise the current server URL will be used.
- A sub record to contain one or more BDL variables used as state variables and defined as reference parameter in the WS-Addressing 1.0 specification.

For example:

```
DEFINE EndpointReferenceState RECORD ATTRIBUTES(W3CEndpointReference)
  address STRING, # Mandatory
  ref RECORD # Sub-record Reference parameters containing one
             # or more state variables
    OpaqueID STRING ATTRIBUTES(XMLName="OpaqueID"), # Unique ID to
               #identify the service state in the database
    Expiration DATE ATTRIBUTES(XMLName="Expiration",
      XMLNamespace="http://tempuri.org") # Session state expiration date
  END RECORD
END RECORD
```

You can use a unique ID of a database table to manage the web services sessions in place of OpaqueID.

Step 2: Create a stateful WS-Addressing enabled web service with W3CEndpointReference record as a parameter

The Genero Web Service extension provides a new Web service constructor called createStatefulWebService() to perform stateful services. This function works as the stateless constructor, but expects a W3CEndpointReference record as parameter.

For example:

```
DEFINE serv com.WebService
LET serv = com.WebService.CreateStatefulWebService(
  "StatefulWSAddressingService","http://4js.com/services",
  EndpointReferenceState) # Create a stateful service
  # with a W3CEndpointReference state variable
CALL serv.setFeature("WS-Addressing1.0","REQUIRED") # enable
  # support of WS-Addressing 1.0
```

Step 3: Publish a web service operation returning the W3CEndpointReference state variable and set it as session initiator

You must define which web service operation will initiate the session on your service and return the W3CEndpointReference state variable.

All other web service operations (not defined as session initiator) will return an error if they don't get reference parameters defined in the W3CEndpointReference state variable as WS-Addressing 1.0 headers.

For example:

```
DEFINE op com.WebOperation
LET op = com.WebOperation.CreateDocStyle("GetInstance",
  "GetInstance",NULL,EndpointReferenceState)
CALL op.initiateSession(TRUE)
CALL serv.publishOperation(op,NULL)
```

There is no restriction regarding the input parameter of the web service initiator function, but the output parameter must be the same W3CEndpointReference record passed to the service creation constructor.

It is not required to have a web operation which initiate the session in the same service, but then you have to return the same W3CEndpointReference record in another web service to instantiate the session, such as a Factory service that instantiates all sessions for other stateful services.

Step 4: Create the BDL session initiator function and instantiate a new session

In your BDL function declared as session initiator, you have to:

- Handle the creation of the session
- Fill the state variable before to return from the function

- Store the new session in a database based on the state variable (in order to keep the session across consecutive requests from the same client).

For example:

```
FUNCTION GetInstance()
  LET EndpointReferenceState.address = NULL
  # Use default end point location
  LET EndpointReferenceState.ref.OpaqueID = com.Util.CreateUUIDString()
  # Generate an unique string (can come from a database table id)
  LET EndpointReferenceState.ref.Expiration = CURRENT  + INTERVAL HOUR TO
 HOUR (1)
  # Create expiration date in one hour to discard request after that date
  ... Store OpaqueID into database or use directly a database table entry
  ... to hold the session
END FUNCTION
```

Step 5: Restore the session in any BDL web operation from the W3CEndpointReference record

In any publish BDL web function, the SOAP engine deserializes the WS-Addressing 1.0 reference parameter headers into the W3CEndpointReference sub-record so that you can retrieve the session from the state variable.

For example:

```
FUNCTION MyFunction()
  IF EndpointReferenceState.ref.OpaqueID IS NULL THEN
    CALL com.WebServiceEngine.SetFaultString("Invalid session id")
    RETURN
  ELSE
    ... Restore the service session based on the OpaqueID state
    ... variable from the database
  END IF
  ... Process the operation
END FUNCTION
```

Client side
Perform these steps to communicate with a stateful web service based on WS-Addressing 1.0.
Step 1: Generate the client stub from your WS-Addressing stateful service

Use the fglwsdl tool as usual. It will detect that the service returns a W3CEndpointReference and generate the appropriate code.

The WSDL imports the WS-Addressing 1.0 schema, so the fglwsdl tool requires an access to the W3C server. Use the option -proxy if you need to connect via a proxy server.

For example:

```
$ fglwsdl -o ws_stub http://localhost:8090/StatefulWSAddressingService?WSDL
```

The generated .inc file contains a variable of type tWSAGlobalEndpointType to be used to transmit the WS-Addressing 1.0 reference parameters.

Example of a global variable name

```
DEFINE
 StatefulWSAddressingService_StatefulWSAddressingServicePortTypeEndpoint
   tGlobalWSAEndpointType
```

Step 2: Create the MAIN application

In your main application:

1. Import the XML library. This is due to the support of WS-Addressing 1.0 with **IMPORT XML**.
2. Import the generated .inc file with **GLOBALS "ws_stub.inc"**
3. Manage the WS-Addressing 1.0 reference parameters representing the session state (if your client has to handle several instances of a same service).

For example:

```
IMPORT XML # Import the XML library required for WS-Addressing 1.0

GLOBALS "ws_stub.inc" # Import service global definition

TYPE InstanceType DYNAMIC ARRAY OF xml.DomDocument
  # End point WSA reference parameters

DEFINE instance1,instance2,instance3 InstanceType
   # Store the different sessions the client will have to manage

MAIN
  ...
END MAIN
```

Step 3: Instantiate a new session by calling the web service operation set as session initiator

Call the BDL function generated from the WSDL that is defined as session initiator on the server. This function returns a **W3CEndpointReference** parameter that contains the WS-Addressing 1.0 reference parameters representing the new instance created on server side.

If your application handles several instances, you will have to copy and store those parameters in your application to identify a service instance for further requests.

As the WS-Addressing 1.0 reference parameters are defined as any XML document, they are represented as a dynamic list of xml.DomDocument in BDL.

For example:

```
DISPLAY "Creating a new instance ..."
LET wsstatus = GetInstance_g() # call the service session initiator
                               # web function
IF wsstatus == 0 THEN
   FOR ind=1 TO
     ns1GetInstanceResponse.return.ReferenceParameters._LIST_0.getLength()
      LET instance1[ind]=

 ns1GetInstanceResponse.return.ReferenceParameters._LIST_0[ind].clone()
        # copy the service returned WS-Addressing 1.0 reference parameters
   END FOR
ELSE
     ... handle soap errors
END IF
```

When creating a new instance, ensure that the **Parameters** member of the generated global variable of type tWSAGlobalEndpointType has been set to NULL, otherwise the server will complain.

Step 4: Call any web service operation with previously returned WS-Addressing 1.0 reference parameters

Before calling any web service operation, you must set the WS-Addressing 1.0 reference parameters returned by a session initiator function to identify the session to the server.

For example:

```
LET StatefulWSAddressingService_StatefulWSAddressingServicePortTypeEndpoint.
Address.Parameters.* = instance1.*
# assign WS-Addressing 1.0 reference parameters dynamic array by reference
CALL MyFunction("Hello") RETURNING wsstatus,ret
# Call web operation MyFunction of instance 1
```

Stateful services based on HTTP cookies

A stateful service based on HTTP cookies uses the HTTP transport protocol and its ability to convey cookies, used as session context. **Notice** that it works only if the communication path between the client and the server is performed in HTTP, otherwise it is recommended to use WS-Addressing stateful services.

Server side
Perform these steps to create an HTTP cookie based stateful service.
Step 1: Declare any BDL simple variable to be used as state variable

For example:

```
DEFINE ServiceState STRING # Unique ID to identify the service state in the
 database
```

For instance, you can use a unique ID of a database table to manage the web services sessions.

Step 2: Create a stateful web service with state variable as parameter

The Genero Web Service extension provides a new Web service constructor called createStatefulWebService() to perform stateful services. This function works as the stateless constructor, but expects a simple state variable as parameter.

Example

```
DEFINE serv com.WebService
LET serv =
 com.WebService.CreateStatefulWebService("StatefulCookieService",
           "http://4js.com/services",ServiceState)
# Create a stateful service with a simple BDL variable as state
 variable
```

Step 3: Publish a web service operation defined as session initiator

Define which web service operation will initiate the session on your service and instantiate a new session. All other web service operations (not defined as session initiator) will return an error if they don't get an HTTP cookie called **GSESSIONID**.

For example:

```
DEFINE op com.WebOperation
LET op =
 com.WebOperation.CreateDocStyle("GetInstance","GetInstance",NULL,NULL)
CALL op.initiateSession(true)
CALL serv.publishOperation(op,NULL)
```

There is no restriction on the web service session initiator function regarding to the input and output parameters.

Step 4: Create the BDL session initiator function and instantiate a new session

In your BDL function declared as session initiator, you must:

- Handle the creation of the session.
- Fill the state variable before to return from the function.
- Store the state variable in a database based on the state variable (in order to keep the session across consecutive requests from a same client).

For example:

```
FUNCTION GetInstance()
  # Generate an unique string (can come from a database table id)
  LET ServiceState = com.Util.CreateUUIDString()
      ... Store ServiceState value into database or use directly a
          database table entry to hold the session
END FUNCTION
```

Step 5: Restore the session in any BDL web operation from the state variable

In any publish BDL web function, the SOAP engine deserializes the HTTP Cookie called **GSESSIONID** from the HTTP layer into the state variable. You can then retrieve the session in BDL via that state variable.

For example:

```
FUNCTION MyFunction()
  IF ServiceState IS NULL THEN
    CALL com.WebServiceEngine.SetFaultString("Invalid session id")
    RETURN
  ELSE
    ... Restore the service session based on the ServiceState
        variable from the database
  END IF
  ... Process the operation
END FUNCTION
```

Step 6: Deployment recommendation

When deploying stateful web services based on HTTP cookies, the complete server path will be added into the cookie when first instantiated, so you must pay attention to that URL. In other words, you MUST always call the service via the complete URL containing the service name inside. For instance if your service is named **MyService** and if you GAS configuration file is called **Server.xcf**, the stateful service is accessible at URL: **http://localhost:6394/ws/r/group/Server/MyService**.

Client side
Perform the following steps to communicate with a stateful web service based on HTTP cookies.
Step 1: Generate the client stub from your stateful service

Use the fglwsdl tool as usual.

For example:

```
$ fglwsdl -o ws_stub http://localhost:8090/StatefulCookieService?WSDL
```

The generated .inc file contains a variable of type tGlobalEndpointType to be used to transmit the HTTP Cookie.

Example of a global variable name

```
DEFINE StatefulCookieService_StatefulCookieServicePortTypeEndpoint
 tGlobalEndpointType
```

Step 2: Create the MAIN application

In your main application:

- Import the generated .inc file with **GLOBALS "ws_stub.inc"**.
- Manage the HTTP cookies representing the session state (if your client has to handle several instances of a same service).

For example:

```
GLOBALS "ws_stub.inc" # Import service global definition

# Store the different sessions the client will have to manage
# in a string
DEFINE instance1,instance2,instance3 String

MAIN
  ...
END MAIN
```

Step 3: Instantiate a new session by calling the web service operation set as session initiator

Call the BDL function generated from the WSDL that was defined as session initiator on the server. This function returns a new HTTP Cookie saved into the **Binding.Cookie** member of the global service variable of type tGlobalEndpointType. If your application handles several instances, you will have to copy and store that cookie in your application to identify a service instance for further requests.

For example:

```
DISPLAY "Creating a new instance ..."
LET wsstatus = GetInstance_g() # call the service session
                               # initiator web function
IF wsstatus == 0 THEN
  # copy the service returned HTTP cookie
  LET instance1 =

 StatefulCookieService_StatefulCookieServicePortTypeEndpoint.Binding.Cookie
ELSE
  ... handle soap errors
END IF
```

When creating a new instance, ensure that the **Binding.Cookie** member of the generated global variable of type tGlobalEndpointType has been set to NULL, otherwise the server will complain.

Step 4: Call any web service operation with previously returned HTTP cookie

Before calling any web service operation, set the HTTP cookie returned by a session initiator function to identify the session to the server.

For example:

```
# use instance1
LET
 StatefulCookieService_StatefulCookieServicePortTypeEndpoint.Binding.Cookie
 =
 instance1
# Call web operation MyFunction of instance 1
CALL MyFunction("Hello") RETURNING wsstatus,ret
```

Step 5: Troubleshooting

If your Genero application doesn't set the HTTP cookie when accessing a stateful service via the GAS, it is possible that you didn't use the complete URL when accessing the service.

For instance if your service is named **MyService** and if you GAS configuration file is called **Server.xcf**, the stateful service is accessible at URL: **http://localhost:6394/ws/r/group/Server/MyService**.

Encryption, base64 and password agent with fglpass tool

For security reasons, it is recommended that you avoid storing clear passwords in a file. The Genero Web Services enables the password encryption of a HTTP Authenticate entry in the FGLPROFILE file. The encrypted password is decrypted by the Genero Web Services engine when required.

The fglpass tool

The Genero Web Services package provides a command line tool called **fglpass**. The **fglpass** tool can encrypt a password from a X.509 certificate or a RSA private key. The encrypted password is displayed on the console in a Base64 form, composed only of alphanumeric characters, and therefore easily usable in any text file.

See fglpass for more details.

Encrypt a HTTP authenticate password

1. Find the HTTP Authenticate entry with the password you want to encrypt:

```
authenticate.myentry.login    = "mylogin"
authenticate.myentry.password = "mypassword"
```

2. Add the certificate and its private key in the FGLPROFILE file as follows:

```
security.mykey.certificate = "MyCertificate.crt"
security.mykey.privatekey  = "MyPrivateKey.pem"
```

3. Encrypt the password with **fglpass**:

```
$ fglpass -c MyCertificate.crt
Enter password :mypassword
```

The **fglpass** output looks like the following:

```
BASE64 BEGIN
dBy3E5JCVxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w
/onWviG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG/
SFJmlVH63kuyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4=BASE64 END
```

> **Note:** The encrypted password corresponds to the big suite of alphanumeric characters between BASE64 BEGIN and BASE64 END. The long line of text is wrapped for display purposes only.

4. Replace the clear password with the encrypted one, and specify the key used to encrypt it (*mykey* in our case):

```
authenticate.myentry.login = "mylogin"
authenticate.myentry.password.mykey = "dBy3E5JCVxuoxsR+
aOBVfp1j0SwQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w
/onWviG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG/
SFJmlVH63kuyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4="
```

> **Note:** Do not forget to put quotes around the base64 form; otherwise the '=' character is interpreted during the loading of FGLPROFILE. The long line of text is wrapped for display purposes only.

Encrypt a HTTP authenticate password using a certificate in the Windows™ key store

1. Find the HTTP Authenticate entry with the password you want to encrypt:

```
authenticate.myentry.login    = "mylogin"
authenticate.myentry.password = "mypassword"
```

2. Add the subject of the certificate registered in the Windows™ key store:

```
security.mykey.subject = "Georges"
```

3. Encrypt the password with **fglpass**:

```
$ fglpass -s Georges
Enter password :mypassword
```

The **fglpass** output looks like this:

```
BASE64 BEGIN
dBy3E5JCVxuoxsR+aOBVfp1j0SwQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w
/onWviG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG/
SFJmlVH63kuyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4=
BASE64 END
```

> **Note:** The encrypted password corresponds to the big suite of alphanumeric characters between BASE64 BEGIN and BASE64 END. The long line of text is wrapped for display purposes only.

4. Replace the clear password with the encrypted one, and specify the key used to encrypt it (*mykey* in our case):

```
authenticate.myentry.login          = "mylogin"
authenticate.myentry.password.mykey = "dBy3E5JCVxuoxsR+
aOBVfp1j0SwQPt+hdjpMKriWvO2xMd5rFnFEwv+sPPd4w
/onWviG0M5mqubBeS7QUlt/ZK0D1aO9/R5RVa5wylQu//6vxfyd8NG/
SFJmlVH63kuyXfiVfq6bHo5+nlQZpVjSHfF2msET3S9HTpZUt4NblP4="
```

> **Note:** Do not forget to put quotes around the base64 form; otherwise the '=' character is interpreted during the loading of FGLPROFILE. The long line of text is wrapped for display purposes only.

Use the password agent

The **fglpass** tool can be started as an agent, to help any BDL application who requires a password to grant access to a private key, by getting it without having to type it. You simply need to enter the password once for each private key at the agent startup, and then **any BDL application started on the same machine and with the same user name as the agent itself can get rid of entering the different passwords**.

Of course, authentication and data encryption are performed between the BDL application and the agent to guarantee passwords confidentiality, and the passwords are also stored encrypted in the agent memory.

1. To start the password agent at port number **4242** and to serve the BDL applications with the passwords of the private key **RSAKey1.pem** and **DSAKey2.der**, specify the option `-agent`, followed by a colon, followed by the port number where it will be reachable, followed by the list of private keys the agent will handle for all BDL applications.

```
fglpass -agent:4242 RSAKey1.pem DSAKey2.der
```

2. The agent will ask you to silently enter the password of the different keys*(the passwords are not displayed to the console when being typed)*. In this example, you have:

```
Enter pass phrase for RSAKey1.pem:
```

Followed by:

```
Enter pass phrase for DSAKey2.der:
```

3. Once all keys have been treated, it displays following message to notify that the agent is ready to serve.

```
Agent started
```

4. To enable one BDL application to use the password agent capability, set the entry called security.global.agent in the FGLPROFILE file with the port number of the agent.

   In our example, with value 4242:

```
security.global.agent = "4242"
```

Encrypt a password

The **fglpass** tool can encrypt a password using an RSA key or certificate, and then encode it in BASE64 form. This allows you to easily add a protected password in the FGLPROFILE file for future use by any BDL application.

1. To encrypt a password from an RSA key and encoded in BASE64, enter:

```
fglpass -e -k RSAPub.pem
```

2. You are prompted to enter the password you want to encrypt.

```
Enter password :hello
```

   The **fglpass** tool outputs the BASE64 form of the encrypted password on the console.

```
BASE64 BEGIN
Pzk/fNRhetdJDZz5kjNg7P0XET4XsW6bys/fi0DvugxRPh9d/s41oAws65
JY0EPb2zytQjxZ/dwaaRzJPYoQmA==
BASE64 END
```

   Note: The BASE64 encrypted password is the string between the BASE64 BEGIN and BASE64 END.

Decrypt a password

The **fglpass** tool can decrypt a BASE64 encoded and encrypted password using the RSA private key that was used to encrypt it or that is associated to a certificate containing the public part of that private key.

1. To decrypt a BASE64 encoded and encrypted password from a RSA private key, enter:

```
fglpass -d -k RSAPriv.pem
```

2. If the RSA key is protected with a password, you are asked to silently enter that password *(the password is not displayed to the console when being typed)*.

```
Enter pass phrase for RSAPriv.pem:
```

3. You are prompted to enter the BASE64 encoded and encrypted password you want to decrypt.

```
Enter password :Pzk/fNRhetdJDZz5kjNg7P0XET4XsW6bys/fi0
DvugxRPh9d/s41oAws65JY0EPb2zytQjxZ/dwaaRzJPYoQmA==
```

   The fglpass tool outputs the password in clear text on the console.

```
hello
```

Encode a file in BASE64 form

The **fglpass** tool can encode a file in BASE64 form.

1. To encode the file **MyFile** in BASE64, enter:

```
fglpass -enc64 MyFile
```

The **fglpass** tool outputs the BASE64 form of the file to the console.

```
BASE64 BEGIN
c2VjdXJpdHkuZ2xvYmFsLmFnZW50ICAgICAgPSAiNDI0MiINCmNyeXB0by
5pZDEua2V5ICAgICAgICAgICAgID0gIlJTQTEwMjRLZXkucGVtIg0KY3J5
cHRvLmlkMi5rZXkgICAgICAgICAgPSAiUlNBMjA0OEtleS5wZW0iDQ
pjcnlwdG8uaWQzLmtleSAgICAgICAgICA9ICJEU0ExMDI0S2V5LnBl
bSINCmNyeXB0by5pZDQua2V5ICAgICAgICAgID0gIlJTQTUxMlByb3
RlY3RlZC5wZW0iDQpjcnlwdG8uaWQ1LmtleSAgICAgICAgICA9ICJE
U0E1MTJSZWFsbHlQcm90ZWN0ZWQucGVtIg0K
BASE64 END
```

Note:

- The BASE64 encoded file is the string between BASE64 BEGIN and BASE64 END.
- You can redirect the output of **fglpass** tool to a file. For example:

```
fglpass -enc64 MyFile > Base64filename
```

Decode a BASE64 form encoded file

The **fglpass** tool can decode a BASE64 encoded file.

1. To decode a file encoded in BASE64 form, enter:

```
fglpass -dec64 Base64filename
```

The **fglpass** tool outputs the file in clear form on the console.

```
security.global.agent      = "4242"
crypto.id1.key             = "RSA1024Key.pem"
crypto.id2.key             = "RSA2048Key.pem"
crypto.id3.key             = "DSA1024Key.pem"
crypto.id4.key             = "RSA512Protected.pem"
crypto.id5.key             = "DSA512ReallyProtected.pem"
```

Note:

- You don't have to remove the BASE64 BEGIN and BASE64 END tags, if they are present in the file, because the **fglpass** tool detects and removes them automatically.
- You can redirect the output of the **fglpass** tool to a file. For example:

```
fglpass -dec64 Base64MyFile > MyFile2
```

HTTP compression

HTTP compression is a capability that can be built into web servers and web clients to make better use of available bandwidth, and provide greater transmission speeds between both.

There are a variety of places where you can set up HTTP compression.

- You can set up the Web services client to send and receive compressed requests. See Compression and a Web services client on page 2433.
- You can enable compression for the Web server. Refer to your Web server documentation for details.
- You can enable compression in the Genero Application Server. Compression is enabled by default in `$FGLASDIR/etc/imt.cfg`. Refer to the *Genero Application Server User Guide* for more information.

- You can set up the Web services server to send and receive compressed requests. See Compression and a Web services server on page 2434.

Compression and a Web services client
Send and receive compressed requests from a Web services client.

When you create a low-level Web service and do not have any stubs created by `fglwsdl`, you need to manage it by setting the HTTP headers.

Important: HTTP request compression for POST/PUT is not supported on GMI mobile devices.

Send a compressed request

The method used to set up the client for sending a compressed request depends on whether the Genero Web Services client is a high-level or low-level Web services client. A *high-level client* is a Genero Web Services client that includes the stub files created by the fglwsdl tool. A *low-level client* is a Genero Web Services client that does not utilize stub files created by the fglwsdl tool.

Regardless of the type of client, the server must be set up to handle such compression, otherwise the request will be rejected.

Send a compressed request from a high-level client

A *high-level client* is a Genero Web Services client that includes the stub files created by the fglwsdl tool.

Set the variable Binding.CompressRequest to either "gzip" or "deflate".

```
LET EchoDocStyle_EchoDocStylePortTypeEndpoint.Binding.CompressRequest =
 "gzip"
```

The `Binding.CompressRequest` variable is defined in the stub file, specifically the client's global (`inc`) file.

```
#
# Global Endpoint user-defined type definition
#
TYPE tGlobalEndpointType RECORD   # End point
        Address RECORD # Address
           Uri STRING                # URI
        END RECORD,
        Binding RECORD # Binding
          Version STRING,             # HTTP Version (1.0 or 1.1)
          Cookie STRING,              # Cookie to be set
          ConnectionTimeout INTEGER,  # Connection timeout
          ReadWriteTimeout INTEGER,   # Read write timeout
          CompressRequest STRING      # HTTP request compression mode (gzip
 or deflate)
       END RECORD
     END RECORD

#
# Location of the SOAP endpoint.
# You can reassign this value at run-time.
#

DEFINE EchoDocStyle_EchoDocStylePortTypeEndpoint tGlobalEndpointType
```

Send a compressed request from a low-level client

A *low-level client* is a Genero Web Services client that does not utilize stub files created by the fglwsdl tool.

Set the `Content-Encoding` field in the request header to either "gzip" or "deflate".

This example sets the `Content-Encoding` field to "gzip", where the request is a `com.HTTPRequest` object.

```
CALL request.setHeader("Content-Encoding","gzip")
```

Accept a compressed response

A Genero Web Services client can accept a compressed request if it sets the `Accept-Encoding` field in the header to "gzip, deflate". These values represent supported compression schema names (called content-coding tokens) separated by commas.

This example sets the `Accept-Encoding` field with the `setHeader` method, where the request is a `com.HTTPRequest` object.

```
CALL request.setHeader("Accept-Encoding","gzip, deflate")
```

Compression and a Web services server

Send and receive compressed requests from a Web services server.

If the Genero Web Services client accepts compression, the Genero Web Services server will reply with a compressed response.

To disable compression, you must disable compression in the Genero Application Server `$FGLASDIR/etc/imt.cfg` file. See the `Genero Application Server User Guide` for more information.

SOAP multipart style requests in GWS

This topic describes multipart support with Genero Web Services

What is multipart style in SOAP?

Multipart style SOAP is the ability to send and receive a SOAP request in multiple pieces. The sending of attached files in separate parts of the SOAP request is one example of a multipart style SOAP request.

Multipart SOAP on the client

When using a WSDL with multipart style, `fglwsdl` generates a client-side stub handling multipart requests. For more details, see Multipart in the client stub on page 2460.

Multipart SOAP on the server

Multipart style is not yet supported with the high-level WS API of Genero.

- It is not possible to write a GWS server handling multipart style SOAP requests with the high-level API.
- When generating code from a WSDL using multipart style, the `fglwsdl` will produce a warning message: `WARNING : Unable to manage MIME Mutlipart binding on message '`*`name`*`'`, where *name* is the name of the message in XML.

Implementing multipart using the low-level APIs

If required, you can implement a WS server handling multipart with the low-level APIs of Genero Web Services. For more details, see com.HTTPServiceRequest.getRequestMultipartType on page 2044.

Security

These topics covers security and Genero Web Services.

- Encryption and authentication on page 2435
- Accessing secured services on page 2438
- HTTPS configuration on page 2440
- Certificates in practice on page 2441
- Examining certificates on page 2443
- Troubleshoot common issues on page 2446
- The Diffie-Hellman key agreement algorithm on page 2447

Encryption and authentication

A scenario involving a person (Georges) and his bank guides you through the concepts of secured communication, certificates, and certificate authorities.

- Secured communications on page 2435
- Certificates on page 2436
- Certificate authorities on page 2437
- Certificates and private keys storage on page 2438

Secured communications

Secured communications are important. If an application wants to send or receive messages from a financial, business, or personnel application on the web, it must be able to authenticate the origin of the message, ensure that no malicious application has altered the original message, and ensure that no third party application can intercept the message.

Suppose that a person named Georges wants to send a message to his bank to transfer some money on the Internet. In this scenario, he faces the following concerns:

1. The **privacy** of the message, since it includes his account number and the transfer amount.
2. The **integrity** of the message, since someone might try to modify the original message or substitute a different message in order to transfer the money to another account.
3. The **authentication** of the message, since the bank must ensure that the message was sent from the right person.

Message privacy

To keep a message private, use a cryptographic algorithm - a technique that transforms a message into an encrypted form unreadable except by those it is intended for. Once it is in this form, the message may only be interpreted through the use of a secret key. There are two kinds of cryptography algorithms: symmetric and asymmetric.

Symmetric means the sender and the receiver of a message have to share the same key used to encrypt a clear message into an encrypted form, and then to decrypt it back into the original message. If that key is kept secret, nobody other than the sender and the receiver can read the message. However, the task of choosing a private key before communicating can be problematic.

Asymmetric means that there are two different keys working as a key-pair. One key is used to encrypt a message, and the second one is used to decrypt the encrypted message back into its original form. This solves the problem of key sharing in the symmetric cryptography algorithm, and makes it possible to receive secure messages, simply by publishing the key used to encrypt messages (the *public key*), and keeping secret the key used to decrypt messages (the *private key*). Anyone can encrypt a message using the public key, but only the owner of the private key can read it.

> **Important:** The use of an asymmetric key-pair (public and private key), allows Georges to send private messages to his bank, simply by using the bank's public key to encrypt a message. Only the owner of the corresponding private key (the bank in this scenario) is able to read it.

Message integrity

To guarantee the integrity of a message, send a concise summary of the original message. The receiver of the message can create its own summary and compare it to the sender's summary. If they are similar, the message is considered intact, meaning that no third party has modified the original message.

Such a summary is called a *message digest* and is based on hash algorithms that produce a fixed-length representation of variable-length messages. Message digests are designed to make it very difficult (if not impossible) to determine the original message from a summary.

The message digest must be sent to the receiver in a secure way to assure the message integrity. This is achieved with a digital signature authenticating the sender and containing the sender's message digest.

> **Important:** The use of message digests allows Georges' bank to verify that no one has modified the original message he sent.

Message authentication

To authenticate a message, add a digital signature to that message.

A *digital signature* is another message, created by encrypting the message digest, along with some other information, with the sender's private key. Anyone with the corresponding public key can decrypt the digital signature. If an application is able to decrypt it, it means the owner of the private key was able to encrypt it, proving that the message comes from this sender and not from someone else.

Once the sender has been authenticated, the receiver can compare the message digest integrated into the digital signature to the one it created from the message it receives, in order to check the message integrity.

> **Important:** The use of digital signatures allows Georges' bank to verify that the message really comes from him.

Certificates

An SSL certificate is a kind of digital identity card that associates the public key with a unique digital thumbprint identifying an individual, a server, or any other entity.

Now that Georges is able to send a secured message to his bank, there is still a problem. How can Georges be sure that the server he is connected to is really the bank's server and not a malicious server?

Georges must be sure that the public key he is using to encrypt his message corresponds to the bank's private key. Similarly, the bank needs to verify that the message signature it receives corresponds to Georges' signature.

To identify a remote peer, use a *certificate* - a kind of digital identity card that associates the public key with a unique digital thumbprint identifying an individual, a server, or any other entity (known as the *subject*). It also includes the identification and signature of the Certificate Authority that issued the certificate, and the period of time during which the certificate is valid. It may have additional information (or extensions) as well as administrative information for the Certificate Authority's use, such as a serial number.

A standard X.509 certificate contains the following standard fields:

- Certificate version
- Serial number of the certificate
- The distinguished name of the certificate issuer
- The distinguished name of the certificate owner
- The validity period of the certificate
- The public key
- The digital signature of the issuer

- Signature algorithm used
- Zero or more certificate extensions

Note:

1. An example of a distinguished name is: **CN=Georges,E=georges@mycompany.com,OU=Sales,O=My Company Name,C=FR,S=France**
2. The CN (Common Name) of the distinguished name of the certificate owner corresponds to the certificate subject, and identifies the owner of that certificate.

Certificate authorities

When a certificate authority signs a certificate, it is validating that the certificate is valid.

Each time Georges sends a message to his bank, he will present his own certificate to the bank, and will get the bank's certificate back. But as every one can create a certificate in the name of Georges, a higher authority that confirms the validity of a certificate is necessary. The bank must be sure it is Georges' certificate, and that no one else has taken his identity. Similarly, Georges needs an authority that confirms that the certificate coming from the server is really the bank's certificate.

The solution to validating a certificate is to sign it with a trusted certificate called **certificate authority**. This is a certificate in which an application creates total confidence concerning the validity of the certificates it has signed. Before signing a certificate, a certificate authority must proceed with a strict identification of the owner of that certificate.

Note: The private key associated to a Certificate Authority must be managed with care, as it is the entity in charge of the validity of all other certificates it has signed.

There are several companies (such as *VeriSign*, *GlobalSign* or *RSA Security*) that have established themselves as certificate authorities and provide the following services over the Internet:

- Verifying certificate requests
- Processing certificate requests
- Issuing and managing certificates

Note: It is also possible to create your own Certificate Authority, but it is up to you to manage it securely.

Root Certificate Authority

A Certificate Authority signed by itself is called a Root Certificate Authority, meaning that the certificate issuer is the same as the certificate subject. Most of the time, such a certificate belongs to a company established as a Certificate Authority, and is used to sign certificate requests coming from different companies that want their own Certificate Authority. If a client certificate is signed by a Certificate Authority previously signed by a Root Certificate Authority, the client certificate can be validated by the Root Certificate Authority even if the Certificate Authority is not present.

For example, if a company wants to buy a Certificate Authority from VeriSign, VeriSign signs that Certificate Authority with its own Root Certificate Authority. The company can then create certificates with the Certificate Authority provided by VeriSign and connect to secure servers without providing them their own Certificate Authority. The secure server, of course, has to know the VeriSign Root Certificate Authority.

Certificate chains

A certificate authority may issue a certificate for another certificate authority. This means that when an application wants to examine the certificate of the issuer, it must check all parent certificates of that issuer until it reaches one it which it has confidence.

The certificate chain corresponds to the number of parent certificate authorities allowed to validate a certificate.

Certificate Authority List

A Certificate Authority List is a list of all certificate authorities considered as trusted by one application, classified by order of importance. Each of these certificates allows the authentication of a certificate presented to that application from a remote peer.

> **Note:** With most applications, the Certificate Authority List is a concatenated file of all certificate authorities.

Certificates and private keys storage

The entire concept of security is based on the publication of the public key, and the privacy of the associated private key. For maximum security, it is critical to restrict the access of the private key to the owner of the certificate and associated private key.

> **Note:** Some companies provide systems to manage certificates and private keys in complete security.

UNIX™ systems

As the UNIX™ system is already able to restrict the access of a file to only one person, simply restrict access to the private key to the owner of that key to achieve a good level of security. This provides enough security to allow a Genero Web Services client to perform secured communications in the name of the certificate and private key owner, because access to the private key file is granted only if the correct user has logged in.

Windows™ systems

The Windows™ system doesn't provide a reliable and sufficiently strong file access rights policy to secure a file. However, Windows™ has an integrated **key store** system to manage certificates and private keys. It allows the registration and the storage of X.509 certificate authorities, as well as personal X.509 certificates and their associated private keys accessible only if the correct user has logged in. It is recommended that you store the certificate and associated private key in the Windows™ key store instead of in files on the disk.

Accessing secured services

Security and authentication are important. Genero Web Services provides various communications options for a client to connect to a Web Service.

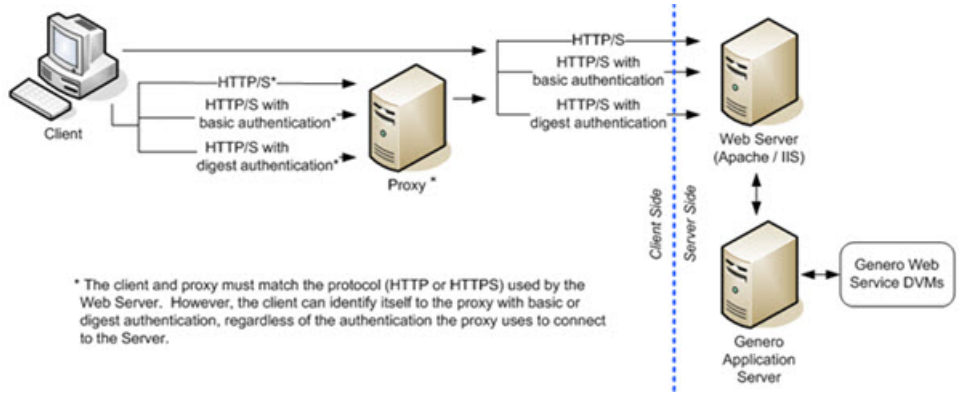


Figure 104: Communications options for a client to connect to a Web Service

HTTP	Client connects to a Web Server (or a Web Service) using HTTP as the communication protocol. (**No security** , **No authentication**).
HTTP with Basic Authentication	Client connects to a Web Server using HTTP as the communication protocol, but a valid login and

	password are required from the Web Server to grant access to the Web Service. (**No security** , **Weak Authentication)**. The login and password are sent in clear text on the communication layer.
HTTP with Digest Authentication	Client connects to the Web Server using HTTP as the communication protocol, but a valid login and password are required from the Web Server to grant access to the Web Service. (**No security** , **Authentication**). The login and password are encoded using a digest algorithm, requiring additional information from the Web Server. This means that the first connection will always fail, but it is necessary in order to return Web Server additional information back to the client.
HTTPS	Client connects to a Web Server using HTTPS as the communication protocol. (**Security** , **No authentication**). The communication channel is encrypted by SSL.
HTTPS with Basic Authentication	Client connects to a Web Server using HTTPS as the communication protocol, but a valid login and password are required from the Web Server to grant access to the Web Service. (**Security** , **Weak Authentication**). The login and password are sent in clear text on the communication layer, but the communication channel is encrypted by SSL.
HTTPS with Digest Authentication	Client connects to the Web Server using HTTPS as the communication protocol, but a valid login and password are required from the Web Server to grant access to the Web Service. (**Security** , **Authentication**). The login and password are encoded using a digest algorithm, requiring additional information from the Web Server. This means that the first connection will always fail, but it is necessary in order to return Web Server additional information back to the client. The communication channel is encrypted by SSL.

To improve communication speed with the cache mechanism, or to restrict internet access to specific clients, Genero Web Services allows a client to connect via proxies. The proxy is in charge of dispatching the client request to the server, and uses the same protocol as that used by the server. So, when a client connects via a proxy to access a HTTP server, the configuration of the HTTP proxy is used, and when the client communicates in HTTPS, the HTTPS proxy configuration is used.

HTTP proxy	Client connects via a proxy using HTTP as the communication protocol.
HTTP proxy with Basic Authentication	Client connects via a proxy using HTTP as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are sent in clear text on the communication layer between client and proxy.
HTTP proxy with Digest Authentication	Client connects via a proxy using HTTP as the communication protocol, but a valid login and password are required from the proxy to dispatch

	the request to the Web Service. The login and password are encoded using a digest algorithm, requiring additional information from the proxy. This means that the first connection will always fail, but it is necessary in order to return proxy additional information back to the client.
HTTPS proxy	Client connects via a proxy using HTTPS as the communication protocol. The communication channel is encrypted by SSL.
HTTPS proxy with Basic Authentication	Client connects via a proxy using HTTPS as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are sent in clear text on the communication layer between client and proxy, but the communication channel is encrypted by SSL.
HTTPS proxy with Digest Authentication	Client connects via a proxy using HTTPS as the communication protocol, but a valid login and password are required from the proxy to dispatch the request to the Web Service. The login and password are encoded using a digest algorithm, requiring additional information from the proxy. This means that the first connection will always fail, but it is necessary in order to return proxy additional information back to the client. The communication channel between client and proxy is encrypted by SSL.

HTTPS configuration

If no HTTPS is provided, Genero Web Services (GWS) does the HTTPS request transparently.

For GWS, use an implicit certificate when no HTTP configuration is provided. For stronger security, you can provide HTTPS configuration with your own certificates and CA list.

The implicit client certificate

For the implicit certificate, no configuration is required. GWS creates a temporary certificate for the HTTPS request. The temporary certificate is valid for the application session.

The explicit client certificate

For the explicit certificate, configure your certificate with `fglprofile` entries.

For access to a specific site, specify `security.ident.certificate` and `security.ident.privatekey`.

If you use the same certificate across all sites, specify `security.global.certificate` and `security.global.privatekey`.

Certificate authorities

Certificate authorities are provided by the system (the operating system keystore). If they are not provided by the system, they are looked for in *`FGLDIR`*`/web_utilities/certs`. Genero Web Services will laod the CA from the directories listed in the `fglprofile` entry "`security.global.ca.lookuppath`". This entry is a list of directories, separated by a semicolon.

You can configure your CA list with the `fglprofile` entry `security.global.ca`.

Mobile platforms

On mobile platforms, no HTTPS configuration is required, because the Web Service library uses the SSL certificates installed in the key database of the device (Keystore for Android™ and Keychain for iOS).

See also GMA / Android Web Services requirements on page 2419.

Certificates in practice

Procedures and tools for creating, importing, and viewing certificates and keys.

- The OpenSSL (openssl) tool on page 2441
- Create a root certificate authority on page 2441
- Create a certificate authority on page 2442
- Create a certificate on page 2442
- Create a certificate authority list on page 2442
- Import a certificate and its private key into the Windows key store on page 2443
- Import a certificate authority into the Windows key store on page 2443
- View a certificate on page 2443
- HTTPS configuration on page 2440

The OpenSSL (openssl) tool

The `openssl` command line tool creates certificates for the configuration of secured communications.

It requires a configuration file with the default parameters such as the key size or the private key name. **OpenSSL** is provided with a default configuration file `openssl.cnf`.

The `openssl` tool looks for the `openssl.cnf` file in the directory where it is executed; it stops if the file is not present. To use the `openssl` tool from any directory, set the `OPENSSL_CONF` environment variable to specify the location of the configuration file.

For information on how the `openssl` tool works, refer to the `openssl` documentation at http://www.openssl.org/docs/apps/openssl.html.

Create a root certificate authority

This procedure allows you to create a root certificate authority.

1. Create the root certificate authority serial file:

```
$ echo 01 > MyRootCA.srl
```

2. Create a CSR (Certificate Signing Request):

```
$ openssl req -new -out MyRootCA.csr
```

   This creates a **privkey.pem** file containing the RSA private key of that certificate and protected by a password.

3. Remove the password of the private key (Optional):

```
$ openssl rsa -in privkey.pem -out MyRootCA.pem
```

   Note: Removing the password of a certificate authority's private key is not recommended.

4. Create a self-signed certificate from the Certificate Signing Request for a validity period of 365 days:

```
$ openssl x509 -trustout -in MyRootCA.csr -out MyRootCA.crt
 -req -signkey MyRootCA.pem -days 365
```

Note: If you want an official Root Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with **openssl**).

Create a certificate authority
This procedure allows you to create a certificate authority.

1. Create a CSR (certificate signing request):

```
$ openssl req -new -out MyCA.csr
```

   This creates a **privkey.pem** file containing to the RSA private key of that certificate and protected by a password.
2. Remove the private key password (Optional):

```
$ openssl rsa -in privkey.pem -out MyCA.pem
```

   Note: Removing the password of a certificate authority's private key is not recommended.
3. Create a certificate from the Certificate Signing Request and trusted by the Root Certificate Authority:

```
$ openssl x509 -in MyCA.csr -out MyCA.crt -req -signkey MyCA.pem
 -CA MyRootCA.crt -CAkey MyRootCA.pem -days 365
```

   Note: If you want an official Certificate Authority, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with **openssl**).

Create a certificate
This procedure allows you to create a certificate.

1. Create the certificate serial file:

```
$ echo 01 > MyCA.srl
```

2. Create a CSR (Certificate Signing Request):

```
$ openssl req -new -out MyCert.csr
```

   This command creates a **privkey.pem** file containing the RSA private key of that certificate and protected by a password.
3. Remove the private key password (Optional):

```
$ openssl rsa -in privkey.pem -out MyCert.pem
```

4. Create a certificate from the Certificate Signing Request and trusted by the Certificate Authority:

```
$ openssl x509 -in MyCert.csr -out MyCert.crt -req -signkey MyCert.pem
 -CA MyCA.crt -CAkey MyCA.pem -days 365
```

   Note: If you want an official Certificate, you must send the CSR file to one of the self-established Certificate Authority companies on the Internet (instead of creating it with **openssl**).

Create a certificate authority list
This procedure allows you to create a certificate authority list using the `openssl` command.

Concatenate all certificate authorities by order of importance, listing the most important first:

```
$ openssl x509 -in MyCA1.crt -text >> CAList.pem
$ openssl x509 -in MyCA2.crt -text >> CAList.pem
```

```
$ openssl x509 -in MyCA3.crt -text >> CAList.pem
```

Import a certificate and its private key into the Windows™ key store
This procedure allows you to import a certificate and its private key.

1. Create a certificate.

   See Create a certificate.
2. Create a specific PKCS12 file containing the certificate and its private key in one file:

   ```
   $ openssl pkcs12 -export -inkey MyCert.pem -in MyCert.crt -out MyCert.p12
   ```

   Note: The .p12 generated file is protected by a password and can then be transported without any risk.
3. On a Windows™ system, open this .p12 file and follow the instructions provided.

   Note: If you select strong verification during the importation process, a popup displays each time an application accesses the private key asking the user whether the application is allowed to use it.

Import a certificate authority into the Windows™ key store
This procedure allows you to import a certificate authority.

1. Create a certificate authority.

   See Create a certificate Authority.
2. Open the .crt certificate file
3. Click **Install Certificate** and follow the instructions provided.

   Windows™ automatically places the certificate in the certificate authority list of the key store.

View a certificate
This procedure allows you to view a certificate using the `openssl` command.

To view a certificate, enter the x509 command:

```
openssl x509 -in MyCompanyCA.crt -noout -text
```

Examining certificates

When you receive a URL in https, you are asked to either accept a certificate or the certificate has already been accepted. In the second case, you can still check the server certificate.

- Check the server certificate using FireFox on page 2443
- Check the server certificate using Internet Explorer on page 2444
- Selecting the certificate to add on page 2444
- Missing certificates on page 2445

Check the server certificate using FireFox
This procedure allows you to check the server certificate using FireFox.

1. Type the https URL.
2. Once the page is displayed, click on the padlock.

   The Page Info for the certificate displays.
3. In the Security tab, click on the View button.

   The Certificate Viewer opens.
4. In the Details tab, view the Certificate Hierarchy.

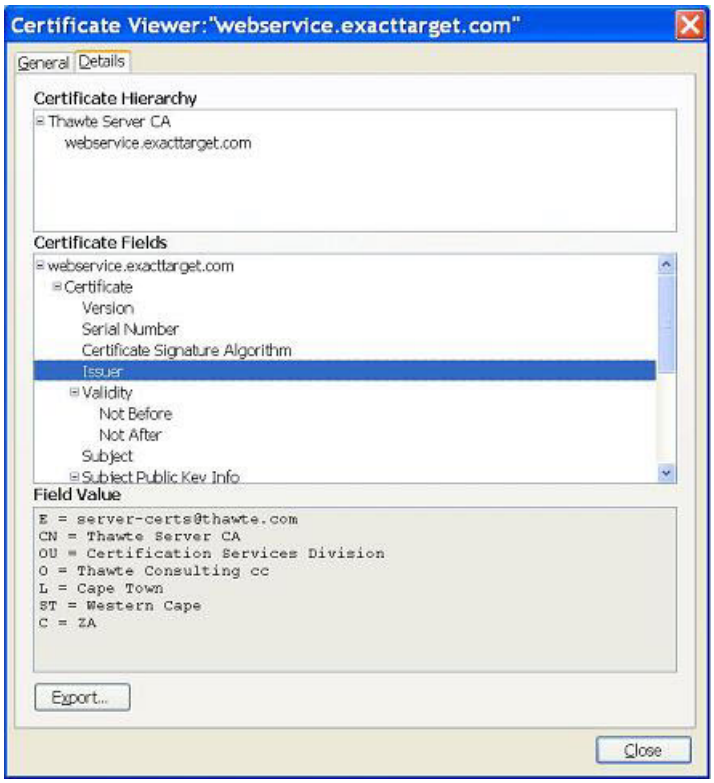


Figure 105: Certificate Viewer; Details tab.

Check the server certificate using Internet Explorer

This procedure allows you to check the server certificate using Internet Explorer.

1. Type the https URL.
2. Once the page is displayed, click on the padlock.

   The Certificate window displays.
3. On the Certification Path, view the certificate hierarchy.

Selecting the certificate to add

The certificate authority (CA) is the authority that validates the server. The certificate to add to the CA list is the authority certificate, not the server certificate.

There are default certificates known by browsers like:

- VeriSign: http://www.verisign.com/support/roots.html
- Thawte: http://www.thawte.com/roots/index.html

Get the server issuer certificate (and all the parents, grandparents, and so on).

For example, if your server is validated by Thawte, add the Thawte certificate to the list.

To check whether your certificate is the CA certificate, search for the CNs (Common Names) in the .cer files. The CA Subject entry should be the Issuer CN in the server certificate.

```
openssl x509 -in server.pem -noout -subject
```

gives:

subject= /C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=CertificationServices Division/ **CN=Thawte Server CA** /emailAddress=server-certs@thawte.com

To convert a .cer certificate to the .pem format used by Genero Web Services:

```
openssl x509 -inform DER -in server.cer -outform PEM -out server.crt
```

Missing certificates

Sometimes the CA hierarchy described in the server certificate is incomplete or needs another certificate (default ones use by browsers or private ones).

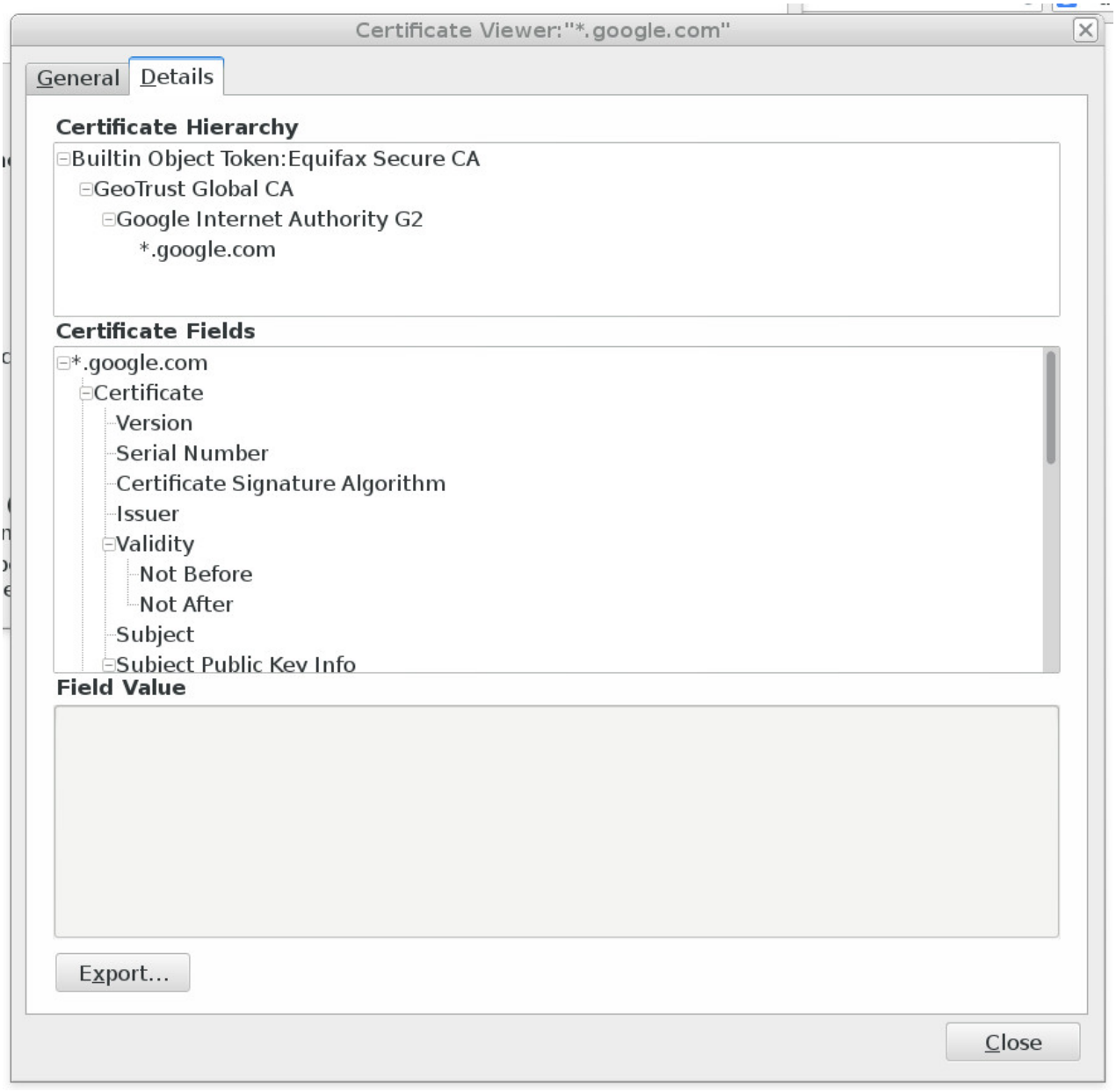


Figure 106: Certificate Viewer in Firefox Web Browser; Details Tab

When this occurs, you will have this kind of error message when you set FGLWSDEBUG:

```
WS-DEBUG (Security error)
Error with certificate at depth: 3
 issuer = /C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification
 Authority
```

```
 subject = /C=US/O=VeriSign, Inc./OU=Class 3 Public Primary Certification
 Authority
 err 19:self signed certificate in certificate chain
WS-DEBUG END
```

This means openssl is looking for a third ancestor that is not listed in the hierarchy above. In this example, gatewaybeta.fedex.com only has two ancestors, and none are named "Class 3 Public Primary Certification Authority". You need to download the root certificates from VeriSign and add "Class 3 Public Primary Certification Authority" in your CA list.

Troubleshoot common issues

You may encounter known (and common) issues when completing the Genero Web Services tutorials or when adding Web services of your own. These issues and their solutions are presented in the following topics.

HTTP 401 error message

An HTTP 401 error message means the server is asking for, but not receiving, authentication (login and password).

This means **authenticate.xxx.login** and **authenticate.xxx.password** are not correctly configured. The login and password should be provided in the FGLPROFILE.

Solution:

1. Open the FGLPROFILE used by the application.
2. Add entries for **authenticate.xxx.login** and **authenticate.xxx.password** .
3. Save your changes.

Error: Peer certificate is issued by a company not in our CA list

When a client needs to connect to a server with https, the client needs to trust the server it is talking to. So the client needs to included the server CAs (certificate authorities list) to its trusted CAs.

This error means the client CA list is missing a certificate authority in its CA list.

To display the client CA list, use the following command:

```
openssl x509 -in ClientCAList.pem -noout -text
```

Solution:

1. Add the missing CA list to the client CA list.

   ```
   openssl x509 -in MyCompanyCA.crt -text >> ClientCAList.pem
   ```

Theory:

Usually certificates work in pairs: a public key and a private key.

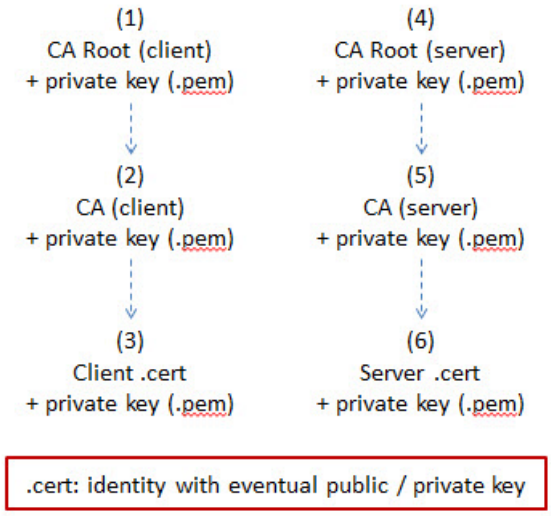


Figure 107: Certificates working in pairs: a public key and a private key

This means that the client has a certificate that can be signed by an authority signed itself by a root authority. Likewise, the server has a certificate that can be signed by an authority signed itself by a root authority. In some instances, a certificate can be signed by itself.

Things to remember:

- The server certificate should have its hostname as CN (Common Name). For example: if you want to access the server https://www.mycompany.com the CN should be www.mycompany.com.
- In client CA list you should have all the CA of the server. In this example you need the server CA (5) and the server CA Root (4). If the server is self-signed then add the server certificate (6) to the client CA list.
- Sometimes, the needed CAs are not listed in the certificates hierarchy. Setting environment variable FGLWSDEBUG=3, will give you information about the missing CA.

The Diffie-Hellman key agreement algorithm

Diffie-Hellman is a key-agreement algorithm. It allows two peers to agree on the same symmetric key, the shared secret, without exchanging confidential data.

The Diffie-Hellman key agreement algorithm is a method that allows two devices to communicate over a network by establishing a shared secret without exchanging any secret data. Knowing the used key-agreement algorithm, the two devices only need to exchange their public key. Then, using the other peer's public key and its own private key, each device performs the algorithm specific key generation operation to obtain the shared secret. The shared secret is a ready-to-use symmetric key for further signed or encrypted exchanges between the two peers.

Genero Web Services provides several shared secret type for signature, encryption, or key encryption purposes. Using the Diffie-Hellman key agreement algorithm, one of the following types of shared secrets can be computed:

- Symmetric AES128 encryption key
- Symmetric AES192 encryption key
- Symmetric AES256 encryption key
- Symmetric TripleDES encryption key
- Symmetric key wrap AES128 key encryption key
- Symmetric key wrap AES192 key encryption key
- Symmetric key wrap AES256 key encryption key
- Symmetric key wrap TripleDES key encryption key
- Symmetric HMAC-SHA1 signature key

In the Diffie-Hellman key agreement algorithm, two shared constants (called parameters) are used in addition to the private and public key. These two parameters are:

- The modulus (called P): A very big prime number chosen at random.
- The generator (called g): A prime number between two and five. Genero Web Services only uses two (2) for the generator.

If the private key (Priv) is a big number (not necessarily prime) chosen randomly, the public key (Pub) is calculated using P, g, and Priv as follows:

$Pub = g^{Priv} \bmod P$

Both devices need to use the same parameters for P and g. There are two ways to ensure this happens: Either P and g are chosen by a third party (such as a security authority) or one of the devices chooses them and sends them to the other peer with its public key.

Genero Web Services allows the Web service to generate the parameters itself, to load them from a string or from a PEM or DER file. The public key and the parameters can also be exchanged using an XML file.

This diagram shows the Diffie-Hellman algorithm steps between two devices, A and B, that need to communicate. Device A is in charge of generating the parameters. The shared secret is labeled K.

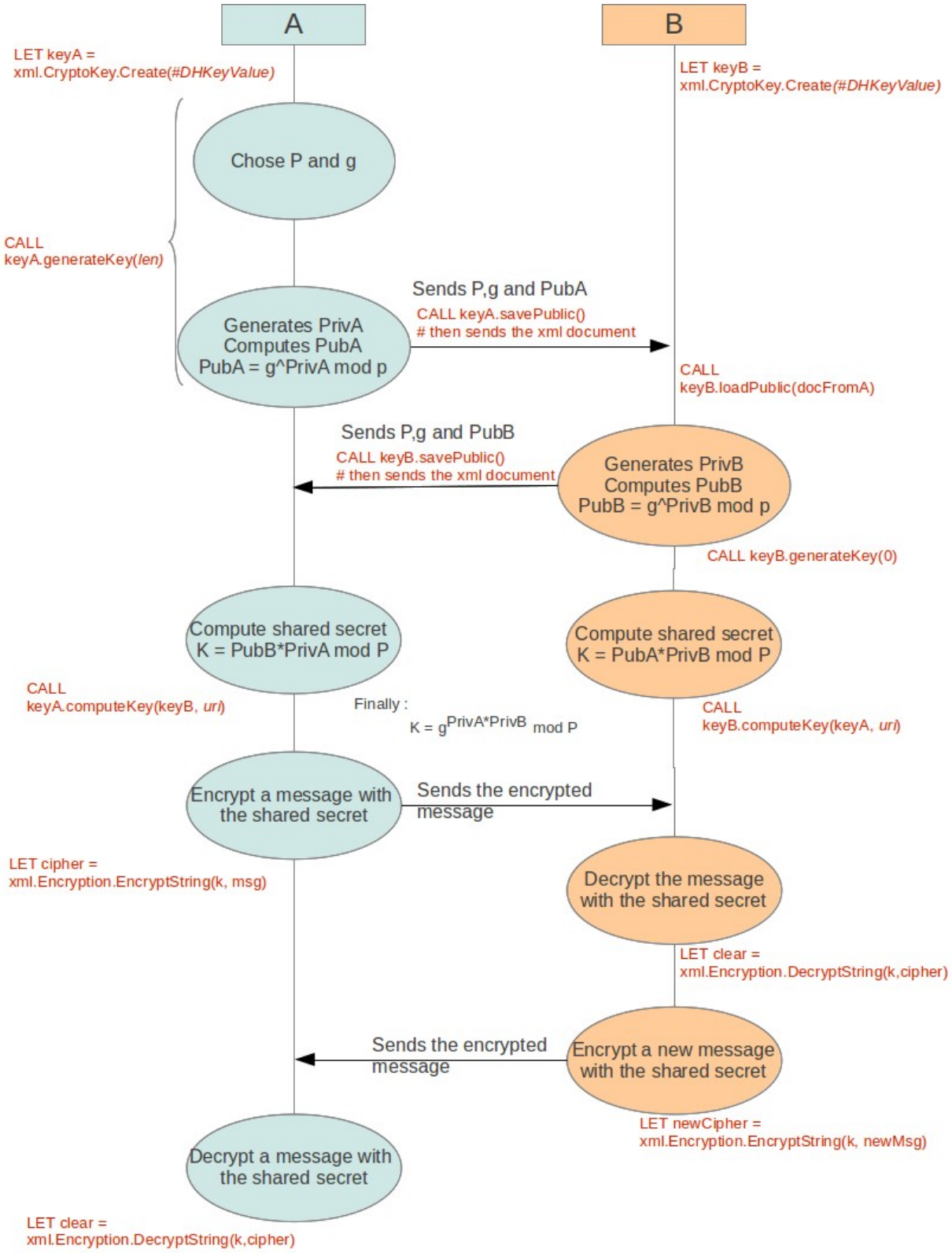


Figure 108: The Diffie-Hellman algorithm

For complete details about the mathematical basics underlying the Diffie-Hellman algorithm, see [RFC2631].

It is nearly impossible to get the private key from the public key, even knowing the values of parameters P and g. Therefore, a middle man will not be able to obtain the shared secret K. While devices A and B exchange their public key, and maybe the parameters as well, these values pass through different intermediate points. It is critical that A receives the correct public key from B, and that B receives the correct public key from A, in order to establish a common shared secret. A middle man could corrupt or replace one public key with his own. If that happens, A and B would be able to communicate because they won't compute the same shared secret. No secret data will be exchanged and readable to the middle man. To avoid this situation, one can use Digital Certificate that helps to deliver the public key and the parameters in an authenticated method.

Once the shared secret is established, the Diffie-Hellman public key, private key and parameters are no longer useful. The Diffie-Hellman key agreement algorithm is achieved.

With the library provided as part of Genero Web Services, the shared secret has been computed to fit given specifications such as HMAC, 3DES, AES128, AES192, AES256, KW-3DES, KW-AES128, KW-AES192, or KW-AES256. The shared secret is actually a symmetric key ready to be used in a signature (HMAC) or cipher algorithm. It allows devices A and B to finally communicate via an authenticated (HMAC) or encrypted method.

SOAP Web Services

Write a Web Services client or server using the SOAP protocol.

The Genero APIs for creating Web services can be found in the Library section of this manual. See The com package on page 2009 and The xml package on page 2103.

Writing a Web Services client application

Create, configure and deploy a Genero Web Services client using the SOAP protocol.

- Steps to write a WS client on page 2450
- Change WS client behavior at runtime on page 2453
- WS client stubs and handlers on page 2456
- Using logical names for service locations on page 2462
- Configure a WS client to connect via an HTTP Proxy on page 2463
- Authenticate the WS client to a server on page 2464
- Authenticate the WS client to a proxy on page 2465
- Configure a WS client to access an HTTPS server on page 2466

Steps to write a WS client

The Genero Web Services package (GWS) allows a BDL program to access Web services found on the Internet. GWS supports the WSDL1.1 specification of March 15, 2002. This example illustrates a client application that accesses the Add operation in the GWS Web Service **MyCalculator**.

See Writing a Web server application on page 2473 for information about the Service.

- Obtaining the WSDL information on page 2450
- Calling a web service on page 2452
- Setting a time period for the response on page 2453
- Handling GWS server errors on page 2453
- Compiling the client application on page 2453

Obtaining the WSDL information

To access a remote Web service, you must get the WSDL information from the service provider. Sample services can be found through UDDI registries or on other sites such as XMethods (http://www.xmethods.net).

You can use the **fglwsdl** tool provided by the Genero Web Services package to obtain the necessary WSDL information. The following example obtains the WSDL information for the GWS Service MyCalculator created by the Writing a Web server application on page 2473:

```
fglwsdl -o Example2Client http://localhost:8090/MyCalculator?WSDL
```

This generates two files:

- `Example2Client.inc` - the globals file containing the definitions of the input and output records, and the prototypes of the operations.
- `Example2Client.4gl` - a module containing the definitions of the functions that can be used in your GWS client application to perform the requested Web Service operation, and the code that manages the Web Service request.

  Note: The MyCalculator GWS Service must be running on the specified port in order to provide the WSDL information.

The following definitions were generated in the globals file, **Example2Client.inc**:

Input and Output records

```
DEFINE Add RECORD
  ATTRIBUTES( XMLName="Add",
              XMLNamespace="http://tempuri.org/webservices" )
  a INTEGER ATTRIBUTES( XMLName="a", XMLNamespace="" ),
  b INTEGER ATTRIBUTES( XMLName="b", XMLNamespace="" )
END RECORD

DEFINE AddResponse RECORD
  ATTRIBUTES( XMLName="AddResponse",
              XMLNamespace="http://tempuri.org/webservices" )
  r INTEGER ATTRIBUTES(XMLName="r",XMLNamespace="" )
END RECORD
```

Since BDL functions cannot have complex structures as parameters, the data types are defined as global or modular variables.

Function prototypes for the Operations

This globals file contains the prototype of two functions for the Add operation.

The **Add** function uses input and output parameters, and returns the status and result. This function can only be used if the input and output parameters are not complex structures such as arrays or records. Using this function, developers do not access the global records directly.

The **Add_g** function can be used with the global input and output records. Before calling this function, you must set the values in the variables of the global input record.

```
Operation: Add
#
# FUNCTION: Add_g()
# RETURNING: soapStatus
# INPUT: GLOBAL Add
# OUTPUT: GLOBAL AddResponse
#
# FUNCTION: Add(p_a, p_b)
# RETURNING: soapStatus ,p_r
```

See fglwsdl on page 1503 and WS client stubs and handlers on page 2456 for more details regarding the **fglwsdl** tool, its output and the generated functions.

Calling a web service

Step 1: Import the COM library of the GWS package

The methods associated with creating and publishing a Web Service are contained in the classes that make up the Genero Web Services Library (com). If you use any of these methods in your client application, you must import the library. Since this example application sets the timeout period that the client will wait for the Service to respond, include the following line at the top of the module:

```
IMPORT com
```

If your generated .inc file uses xml class data types, you need to add `IMPORT xml`.

Step 2: Specify the globals file

Use a GLOBALS statement to specify the generated globals file.

```
GLOBALS "Example2Client.inc"
```

Step 3: Write the MAIN program block

Provide values for the input and output messages of the operation, and call one of the generated functions. Since the input and output messages are simple integers, we can call the **Add** function.

```
MAIN
  DEFINE op1        INTEGER
  DEFINE op2        INTEGER
  DEFINE result     INTEGER
  DEFINE wsstatus   INTEGER

  LET op1 = 1
  LET op2 = 2
  CALL Add(op1, op2) RETURNING wsstatus, result
  IF wsstatus = 0 THEN
    DISPLAY "Result: ", result
  ELSE
    -- Use the global wsError record
    DISPLAY "Error: ", wsError.description
  END IF
END MAIN
```

Alternatively, we can use the global input and output records directly, calling the **Add_g** function:

```
MAIN
  DEFINE wsstatus INTEGER

  LET Add.a = 1
  LET Add.b = 2
  LET wsstatus = Add_g()
  IF wsstatus != 0 THEN
    -- Use the glocal wsError record
    DISPLAY "Error :", wsError.Description
  ELSE
    DISPLAY "Result: ", AddResponse.r
  END IF
END MAIN
```

These examples are very basic versions of the code. For complete examples, see the code samples provided with the package in **demo/WebServices**.

Setting a time period for the response

To protect against remote server failure or unavailability, set a timeout value that indicates how long you are willing to wait for the server to respond to your request.

Use the `SetOption()` method of the `WebServiceEngine` class to set the **readwritetimeout** option.

For example, to wait no more than 10 seconds:

```
CALL com.WebServiceEngine.SetOption( "readwritetimeout", 10 )
```

A timeout value of **-1** means "wait forever". This is the default value.

Handling GWS server errors

When a Genero Web Services Service operation returns a status that is non-zero, you can get a more detailed error description from the global record **wsError**.

This record is defined defined in the globals `.inc` file.

```
DEFINE wsError RECORD
  code STRING,          -- Short description of the error
  codeNS STRING,        -- The namespace of the error code
  description STRING,   -- Long description of the error
  action STRING         -- internal "SOAP action"
END RECORD
```

Compiling the client application

The library file `WSHelper.42m`, included in the `$FGLDIR/lib` directory of the Genero Web Services package, should be linked into every client or server program. Assuming the example client code shown above is in a module named `clientmain.4gl`, you can compile and link the client program:

```
fglcomp clientmain.4gl Example2Client.4gl
fgllink -o myclient.42r clientmain.42m Example2Client.42m WSHelper.42m
```

Change WS client behavior at runtime

Genero Web Services generates a global record called tGlobalEndpointType to change the client behavior at runtime without the need to modify any generated client stub. If WS-Addressing 1.0 is enabled, the global generated record is called tWSAGlobalEndpointType, and if needed you can also access the HTTP layer via the Request and Response record of the binding section.

- Global Endpoint type definition on page 2453
- WS-Addressing 1.0 Global Endpoint type definition on page 2454
- Change server location on page 2454
- Change the HTTP protocol version on page 2455
- Set an HTTP cookie on page 2455
- Set the connection timeout for a service on page 2455
- Set the read and write timeout for a service on page 2455
- Access HTTP request and response headers for a service on page 2455

Global Endpoint type definition

The following global type is used by any generated client stub to allow the programmer to change the client behavior at runtime.

```
TYPE tGlobalEndpointType RECORD # End point
      Address RECORD # Address
         Uri STRING # URI
      END RECORD,
      Binding RECORD # Binding
         Version STRING, # HTTP Version (1.0 or 1.1)
```

```
          Cookie STRING, # Cookie to be set
          ConnectionTimeout INTEGER,# Connection timeout
          ReadWriteTimeout INTEGER # Read write timeout
      END RECORD
END RECORD
```

Description of variables:

- `Address.Uri`: Represents the location of the server.

  Important: It replaces the global variable of type STRING generated prior to version 2.40, therefore it is mandatory to regenerate the client stub and to modify the location assignation in your application.
- `Binding.Version`: Represents the HTTP version to use for communication (only 1.0 or 1.1 allowed, default is 1.1).
- `Binding.Cookie`: Represents the HTTP cookie to use for communication (or NULL if there is no cookie to send).
- `Binding.ConnectionTimeout` : Represents the maximum time in seconds to wait for the establishment of the connection to the server.
- `Binding.ReadWriteTimeout`: Represents the maximum time in seconds to wait for a connection read or write operation before breaking the connection.

WS-Addressing 1.0 Global Endpoint type definition

The following global type is used by any generated client stub where support of WS-Addressing 1.0 is enabled. It allows the programmer to change the client behavior at runtime, and to send additional WS-Addressing 1.0 reference parameters to a server.

If this global type is used in your main application, you must add the `IMPORT xml` instruction.

```
TYPE tWSAGlobalEndpointType RECORD # End point
  Address RECORD # Address
    Uri STRING, # URI
    Parameters DYNAMIC ARRAY OF xml.DomDocument
      ATTRIBUTES(XMLNamespace="##any",XMLAny) # End point WSA reference
 parameters
  END RECORD,
  Binding RECORD # Binding
    Version STRING, # HTTP Version (1.0 or 1.1)
    Cookie STRING, # Cookie to be set
    ConnectionTimeout INTEGER,# Connection timeout
    ReadWriteTimeout INTEGER # Read write timeout
  END RECORD
END RECORD
```

Description of variables:

- `Address.Parameters`: Represents the WS-Addressing 1.0 reference parameter to send to a WS-Addressing 1.0 compliant server.

Change server location

To change the server location at runtime, set the record `Uri` member with a valid URL of another service. All services must respect the same WSDL contract. If you let the variable unset, the client will connect to the server URL defined in the WSDL at code generation time.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Address.Uri =
  http://zeus:1111/mydomain/Calculator
```

You can assign this variable with a URL set in the FGLPROFILE (see Logical Service location).

If you are migrating from a version prior to 2.40, see the migration note.

Change the HTTP protocol version

To communicate with a service that speaks only a given version of HTTP, set the record `Version` member with the desired value. If you let the variable unset, the client will communicate in HTTP 1.1.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Binding.Version = "1.0"
```

If you do not want the request to be split into chunks, set the HTTP protocol version to 1.0.

> **Important:** On GMI mobile devices, the HTTP protocol version definition is ignored, it will always be version 1.1.

Set an HTTP cookie

To send an HTTP cookie to the service, set the record `Cookie` member with the cookie value. If you let the variable unset, the client won't send any cookie.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Binding.Cookie = "MyCookie=AValue"
```

Unset that variable if you don't need the cookie to be sent anymore.

Set the connection timeout for a service

To change the default timeout value for the establishment of the connection to the service, set the record `ConnectionTimeout` member with the timeout value in seconds.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Binding.ConnectionTimeout = 15
```

> **Important:** On GMI mobile devices, the max of `ConnectionTimeout` and `ReadWriteTimeout` will be used.

Set the read and write timeout for a service

To change the default time of reading and writing to or from a service, set the record `ReadWriteTimeout` member with the time out value in seconds.

Example:

```
LET Calculator_CalculatorPortTypeEndpoint.Binding.ReadWriteTimeout = 5
```

> **Important:** On GMI mobile devices, the max of `ConnectionTimeout` and `ReadWriteTimeout` will be used.

Access HTTP request and response headers for a service

To access HTTP headers exchanged between the Genero client and a web service, you must use following records in the binding section :

- one record called `Request` in order to customize HTTP headers to be sent to a web service
- one record called `Response` in order to retrieve all HTTP headers returned by a web service

```
TYPE tGlobalEndpointWithHttpLayerType RECORD # End point
  Address RECORD # Address
    Uri STRING # URI
  END RECORD,
  Binding RECORD # Binding
    Version STRING, # HTTP Version (1.0 or 1.1)
```

```
    Cookie STRING, # Cookie to be set
    Request RECORD
      Headers DYNAMIC ARRAY OF RECORD # HTTP Headers
        Name   STRING,
        Value  STRING
    END RECORD,
    Response RECORD
      Headers DYNAMIC ARRAY OF RECORD # HTTP Headers
        Name   STRING,
        Value  STRING
    END RECORD,
    ConnectionTimeout INTEGER,# Connection timeout
    ReadWriteTimeout INTEGER # Read write timeout
    CompressRequest  STRING # HTTP compression mode (gzip or deflate)
  END RECORD
END RECORD
```

Description of additional Request and Response variables:

- `Binding.Request.Headers`: Represents the additional HTTP headers to be sent to the web service. (Notice that client stub headers will replace user ones if the same name).
- `Binding.Response.Headers`: Represents the HTTP headers returned by a web service.

WS client stubs and handlers

To access a remote Web Service, you first must get the WSDL information from the service provider. Sample services can be found through UDDI registries (http://www.uddi.org), or on other sites such as XMethods (http://www.xmethods.net).

- Generating stub files for a GWS Client on page 2456
- Handling GWS server errors on page 2453
- Global Endpoint user-defined type definition on page 2457
- The generated functions on page 2457
- The generated callback handlers on page 2458
- Example output on page 2459
- Using the generated functions on page 2461

Generating stub files for a GWS Client

Use the `fglwsdl` tool to generate the BDL stub from a WSDL URL or file.

The next example requests the Calculator Web Service information from the specified URL, and the output files will have the base name ws_calculator:

```
fglwsdl -o ws_calculator http://localhost:8090/Calculator?WSDL
```

For a client application, `fglwsdl` generates two output files, which should not be modified:

- *filename*.inc - the globals file, containing declarations of global variables that can be used as input or output for functions accessing Web Service operations, and the global wsError record. In our example, the file is `ws_calculator.inc`.

  This file must be listed in a GLOBALS statement at the top of any .4gl modules that you write for your GWS Client application.
- *filename*.4gl - containing the definitions of the functions that can be used in your GWS client application to perform the requested Web Service operation, and the code that manages the Web Service request. In our example, the file is `ws_calculator.4gl`.

  This file must be compiled and linked into your GWS Client application.

Handling GWS server errors

When a Genero Web Services Service operation returns a status that is non-zero, you can get a more detailed error description from the global record **wsError**.

This record is defined defined in the globals `.inc` file.

```
DEFINE wsError RECORD
  code STRING,          -- Short description of the error
  codeNS STRING,        -- The namespace of the error code
  description STRING,   -- Long description of the error
  action STRING         -- internal "SOAP action"
END RECORD
```

Global Endpoint user-defined type definition

The fglwsdl tool generates the globals (inc) file to use for a Web services client. Part of this globals file is a global endpoint user-defined type definition.

```
#
# Global Endpoint user-defined type definition
#
TYPE tGlobalEndpointType RECORD   # End point
        Address RECORD # Address
           Uri STRING                # URI
        END RECORD,
        Binding RECORD # Binding
          Version STRING,             # HTTP Version (1.0 or 1.1)
          Cookie STRING,              # Cookie to be set
          ConnectionTimeout INTEGER,  # Connection timeout
          ReadWriteTimeout INTEGER,   # Read write timeout
          CompressRequest STRING      # HTTP request compression mode (gzip
 or deflate)
       END RECORD
     END RECORD

#
# Location of the SOAP endpoint.
# You can reassign this value at run-time.
#

DEFINE EchoDocStyle_EchoDocStylePortTypeEndpoint tGlobalEndpointType
```

The `CompressRequest` entry is of type string. It is NULL by default, meaning that no request is compressed. To compress a request, set this variable to **gzip** or **deflate**. The server must support compression, otherwise the request will be rejected.

The generated functions

Genero Web Services (GWS) client functions have the following requirements:

- The function cannot have input parameters.
- The function cannot have return values.
- The function's input message must be defined as a global or module RECORD.
- The function's output message must be defined as a global or module RECORD.

As a result, two types of GWS functions are generated for the Web Service operation that you requested:

- One function type uses global records for the input and output. The names of these functions end in "_g". Before calling the function in your GWS Client application, you must set the values in the global input record. After the function call, the status of the request is returned from the server, and the output message is stored in the global output record. In addition to performing the desired operation, this function handles the communication for the SOAP request and response, and sets the values in the wsError record as needed.

- The other function type serves as a "wrapper" for the "_g" function. It passes the values of input parameters to the "_g" function, and returns the output values and status received from the "_g" function. Your client application does not need to directly access the global records. This function can only be used if the parameters are simple variables (no records or arrays).

The generated **.inc** globals file contains comments that list the prototypes of the functions for the GWS operation, and the definitions of the global INPUT and OUTPUT records.

The generated callback handlers

More and more Web Services provide support of the different **WS-*** specifications. To enable a better interoperability with such services, the fglwsdl tool allows the programmer to modify the SOAP request before it is sent, and to perform additional verifications of the SOAP response before it is returned from the BDL function.

When option **-domHandler** is used, the fglwsdl tool performs the following two operations at once:

- It generates the client stub based entirely on the DOM API to ease the manipulation of the XML requests and responses.
- It generates additional calls for each operation of a service to execute one of the three callback handlers the programmer has to implement.

Handler definition

There are three kind of callbacks you **must** implement for each service generated with the -domHandler option.

- The request handler that allows the modification of the entire SOAP request before it is send over the net.

  It must be named ***ServiceName*_HandleRequest**, where *ServiceName* is the name of the service according to the different prefix options used during generation.

  It must return **TRUE** if you want the caller function to continue normally or **FALSE** to return from the caller function with a SOAP error you can define via the **wsError** record.

```
FUNCTION ServiceName_HandleRequest(operation,doc,header,body)
  DEFINE operation STRING          -- Operation name of the
                                   -- request to be modified.
  DEFINE doc       xml.DomDocument -- Entire XML document of the request
  DEFINE header    xml.DomNode     -- XML node of the SOAP header
                                   -- of the request
  DEFINE body      xml.DomNode     -- XML node of the SOAP body of the
                                   -- request
  CASE operation
    WHEN "Add"
      ... -- Use the DOM APIs to modify the request of the Add operation
    WHEN "Sub"
      ... -- Use the DOM APIs to modify the request of the Sub operation
    OTHERWISE
      DISPLAY "No modification for operation :",operation
  END CASE
  RETURN TRUE -- Continue normally in Add_g() or Sub_g()
END FUNCTION
```

- The response handler that allows the validation of the entire SOAP response before it is de-serialized into the corresponding record.

  It must be named ***ServiceName*_HandleResponse**, where *ServiceName* is the name of the service according to the different prefix options used during generation.

  It must return **TRUE** if you want the caller function to continue normally or **FALSE** to return from the caller function with a SOAP error you can define via the **wsError** record.

```
FUNCTION ServiceName_HandleResponse(operation,doc,header,body)
```

```
  DEFINE operation STRING         -- Operation name of the
                                  -- response to be checked.
  DEFINE doc      xml.DomDocument -- Entire XML document of the response
  DEFINE header   xml.DomNode     -- XML node of the SOAP header of
                                  -- the response
  DEFINE body     xml.DomNode     -- XML node of the SOAP body of the
                                  -- response
  CASE operation
    WHEN "Add"
      ... -- Use the DOM APIs to check the response of the Add operation
    WHEN "Sub"
      ... -- Use the DOM APIs to check the response of the Sub operation
    OTHERWISE
      DISPLAY "No verification for operation :",operation
  END CASE
  RETURN TRUE -- Continue normally in Add_g() or Sub_g()
END FUNCTION
```

- The fault response handler that allows the verification of the entire SOAP fault response before it is de-serialized into the wsError record.

  It must be named ***ServiceName*_HandleResponseFault**, where *ServiceName* is the name of the service according to the different prefix options used during generation. It must return **TRUE** if you want the caller function to continue normally or **FALSE** to return from the caller function with a SOAP error you can define via the **wsError** record.

```
FUNCTION ServiceName_HandleResponseFault(operation,doc,header,body)
  DEFINE operation STRING          -- Operation name of the fault
                                   -- response to be checked.
  DEFINE doc       xml.DomDocument -- Entire XML document of the fault
 response
  DEFINE header    xml.DomNode     -- XML node of the SOAP header of the
                                   -- fault response
  DEFINE body      xml.DomNode     -- XML node of the SOAP body of the
 fault
                                   -- response
  CASE operation
    WHEN "Add"
      ... -- Use the DOM APIs to verify the SOAP fault response
          -- of the Add operation
    WHEN "Sub"
      ... -- Use the DOM APIs to verify the SOAP fault response
          -- of the Sub operation
    OTHERWISE
      DISPLAY "No verification for operation :",operation
  END CASE
  RETURN TRUE -- Continue normally in Add_g() or Sub_g()
END FUNCTION
```

Example output

The example Web Service for which the WSDL information was requested, Calculator, has an Add operation that returns the sum of two integers.

The generated file **ws_calculator.inc** lists the prototype for the **Add** and **Add_g** functions, the asynchronous **AddRequest_g** and **AddResponse_g** functions, as well as the definitions of the global variables **Add** and **AddResponse**:

```
# Operation: Add## FUNCTION: Add_g()  -- Function that uses the global input
                                     -- and output records
#   RETURNING: soapStatus            -- An integer where 0 represents
 success
#   INPUT: GLOBAL Add
```

```
#   OUTPUT: GLOBAL AddResponse
#
# FUNCTION: Add(p_a, p_b)             -- Function with input parameters
 that
#  RETURNING: soapStatus ,p_r         -- correspond to the a and b
 variables
                                      -- of the global INPUT record
                                      -- Return values are the status
 integer
                                      --  and the value in the r variable
 of
                                      -- the global OUTPUT record
#
# FUNCTION: AddRequest_g()            -- Asynchronous function that uses
 the
                                      -- global input record
#  RETURNING: soapStatus               -- An integer where 0 represents
#   INPUT: GLOBAL Add                  -- success, -1 error and -2 means
 that
                                      -- a previous request was sent
                                      -- and that a response is in progress.
#
# FUNCTION: AddResponse_g()           -- Asynchronous function that uses
                                      -- the global output record
#  RETURNING: soapStatus               -- An integer where 0 represents
#   OUTPUT: GLOBAL AddResponse         -- success, -1 error and -2 means that
                                      -- the response was not
                                      -- yet received, and that a new call
                                      -- should be done later.

#VARIABLE : Add   -- defines the global INPUT record
DEFINE Add RECORD ATTRIBUTES(XMLName="Add",
                             XMLNamespace="http://tempuri.org/")
       a INTEGER ATTRIBUTES(XMLName="a",XMLNamespace=""),
       b INTEGER ATTRIBUTES(XMLName="b",XMLNamespace="")
    END RECORD

# VARIABLE : AddResponse   -- defines the global OUTPUT record
DEFINE AddResponse RECORD ATTRIBUTES(XMLName="AddResponse",
                          XMLNamespace="http://tempuri.org/")
       r INTEGER ATTRIBUTES(XMLName="r",XMLNamespace="")
    END RECORD
```

Multipart in the client stub

You can generate a client stub for a Web service that has multiple parts.

If the WSDL for a Web service indicates that the Web service uses multiple parts, the client stub generated will support multiple parts.

For the request

There are as many `com.HTTPPart` input parameters as parts defined for the input request, plus one `AnyInputParts DYNAMIC ARRAY OF com.HTTPPart` parameter, to manage the optional parts a user can add to the request.

For example:

```
FUNCTION xxx_g(InputHttpPart_1, ..., InputHttpPart_n, AnyInputParts)
  DEFINE InputHttpPart_1 com.HTTPPart
  ...
  DEFINE InputHttpPart_n com.HTTPPart
  DEFINE AnyInputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
```

```
  RETURN wsstatus
END FUNCTION
```

For the response

There are as many `com.HTTPPart` variables are described in the WSDL, plus one `AnyOutputParts DYNAMIC ARRAY OF com.HTTPPart` to handle the optional parts that may be returned by a service.

For example:

```
FUNCTION xxx_g()
  DEFINE wsstatus INTEGER
  DEFINE OutputHttpPart_1 com.HTTPPart
  DEFINE AnyOutputParts DYNAMIC ARRAY OF com.HTTPPart
  ...
  RETURN wsstatus, OutputHttpPart_1, AnyOutputParts
END FUNCTION
```

Using the generated functions

The information obtained from the **ws_calculator.inc** file allows you to write code in your own .4gl module as part of the Client application, using the Web Service operation Add.

Using parameters and return values

Since the input variables for our example are simple integers, you can call the **Add** function in your Client application, defining variables for the parameters and return values.

```
FUNCTION myWScall()
  DEFINE op1 INTEGER
  DEFINE op2 INTEGER
  DEFINE result INTEGER
  DEFINE wsstatus  INTEGER
  ...
  LET op1 = 6
  LET op2 = 8
  CALL Add(op1, op2)
    RETURNING wsstatus, result ...
  DISPLAY result
```

Using global records

You could choose to call the **Add_g** function instead, using the global records **Add** and **AddResponse** directly. If the input variables are complex structures like records or arrays, you are required to use this function.

```
FUNCTION myWScall()
  DEFINE wsstatus INTEGER
  ...
  LET Add.a = 6
  LET Add.b = 8
  LET wsstatus = Add_g()
  ...
  DISPLAY AddResponse.r
```

In this case, the status is returned by the function, which has also put the result in the **AddResponse** global record.

See Tutorial: Writing a Client Application for more information. The **demo/WebServices** subdirectory of your Genero installation directory contains complete examples of Client Applications.

Using asynchronous calls

If you don't want your application to be blocked when waiting for the response to a request, you should first call **AddRequest_g**; this will send the request using the global **Add** record to the server. It returns a status of 0 (zero) if everything goes well, -1 in case of error, or -2 if you tried to resend a new request before the previous response was retrieved.

```
FUNCTION sendMyWScall()
  DEFINE wsstatus INTEGER
  ...
  LET Add.a = 6
  LET Add.b = 8
  LET wsstatus = AddRequest_g()
  IF wstatus <> 0 THEN
    DISPLAY "ERROR :", wsError.code
  END IF
  ...
```

You can then call **AddResponse_g** to retrieve the response in the **AddResponse** global record of the previous request. If returned status is 0 (zero) the response was successfully received, -1 means that there was an error, and -2 means that the response was not yet received and that the function should be called later.

```
FUNCTION retrieveMyWScall()
  DEFINE wsstatus INTEGER
  ...
  LET wsstatus = AddResponse_g()
  CASE wstatus
    WHEN -2
      DISPLAY "No response available, try later"
    WHEN 0
      DISPLAY "Response is :",AddResponse.r
    OTHERWISE
      DISPLAY "ERROR :", wsError.code
  END CASE
  ...
```

You can mix the asynchronous call with the synchronous one as they are using two different requests. In other words, you can perform an asynchronous request with **AddRequest_g**, then a synchronous call with **Add_g**, and then retrieve the response of the previous asynchronous request with **AddResponse_g**.

> **Important:** In development mode, a single BDL Web Service server can only handle one request at a time, and several asynchronous requests in a row without retrieving the corresponding response will lead to a deadlock. To support several asynchronous requests in a row, it is recommended that you are in deployment mode with a GAS as the front end.

Using logical names for service locations

Genero Web Services, starting with version 2.00, provides a repository for Web Service locations using FGLPROFILE. To achieve maximum flexibility, you can map a logical reference used by your Web Services Client application to an actual URL. This is subject to the network configuration and access rights management of the deployment site.

> **Important:** On GMI mobile devices, FGLPROFILE settings for logical names are not supported.

- FGLPROFILE entry on page 2463
- Logical reference in the client application on page 2463
- Logical reference in the URL on page 2463

FGLPROFILE entry

The following entry in the FGLPROFILE file maps the logical reference "myservice" to an actual URL:

```
ws.myservice.url = " http://www.MyServer.com/cgi-bin/fglccgi.exe/ws/r/
MyWebService"
```

Logical reference in the client application

When you generate a Client stub from WSDL information using the tool **fglwsdl**, a global variable for the URL of the Web Service is contained in the **.inc** file.

For example:

```
# Location of the SOAP server.
# You can reassign this value at run-time.
#
DEFINE Calculator_CalculatorPortTypeEndpoint tGlobalEndpointType
```

You can assign a logical name to this global variable in your Web Services Client application:

```
LET Calculator_CalculatorPortTypeEndpoint.Address.Uri = "alias://myservice"
```

When the Client application accesses the Service, the actual location will be supplied by the entry in FGLPROFILE on the Client machine. This allows you to provide the same compiled .42r application to different customers. The entries in FGLPROFILE on each customer's machine would customize the Web Service location for that customer.

If you are migrating from a version prior to 2.40, see migration note.

Logical reference in the URL

When you deploy a Genero Web Service with a GAS behind a Web Server, the service can be accessed by two different URLs. You can use a logical name in the URL, mapping the actual location of the Web Service in FGLPROFILE, depending on the location of the client machine.

For example:

- For internal Clients: **http://zeus:6394/ws/r/myservice**
- For Clients using the Web: **http://www.myServer.com/...**

These two URLS could be mapped in the FGLPROFILE file on the Client machine, each specifying the location of the Service.

Configure a WS client to connect via an HTTP Proxy

Configuration steps to connect via a HTTP proxy.

> **Important:** On GMI mobile devices, FGLPROFILE settings are ignored: The device configuration for proxy will always be used.

1. Add the location of the proxy to `fglprofile` with the `proxy.http.location` entry.

   Add the entry `proxy.http.location` to your `fglprofile`. For the value, provide the IP address of the HTTP proxy and the port number where the HTTP proxy is listening, separated by a colon. For example, to have a client connect via a HTTP proxy located at the IP address "10.0.0.170" and listening on port number "8080", add this entry to your `fglprofile`:

   ```
   proxy.http.location = "10.0.0.170:8080"
   ```

   > **Note:** To configure the client to connect via an HTTPS proxy, replace *http* with *https*.

2. Define the list of host names the client will **not** have to connect to via a proxy with the `proxy.http.list` entry.

Add the entry `proxy.http.list` to your `fglprofile`. For the value, provide a semi-colon separated list of clients. For example, to exclude all hosts beginning with "www.mycompany.com" or "www.google." from connecting via a HTTP proxy, add this entry to your `fglprofile`:

```
proxy.http.list = "www.mycompany.com;www.google."
```

Configure a WS client to use IPv6
Configuration steps to customize IPv6 for a WS client.

A Web Services client program can access to a WS server using IPv6.

URLs that map to IPv6 addresses will be automatically handled by the Web Services library. It is also possible to specify an IPv6 address directly as URL in your BDL code by enclosing the address in `[]` square brackets, for example:

```
LET myURL = "http://[fe80::20c:29ff:fe05:9ca3]:80/index.html"
```

By default, the WS library will automatically use IPv6 addresses if available, and fallback to IPv4 otherwise. To overcome the default behavior, you can specify explicitely the IP version.

Indeed, the platform where WS client programs execute must support IPv6.

1. If needed, force the IP version with the `ip.global.version` entry in `fglprofile`, by specifying "4" for IPv4 or "6" for IPv6.

   For example, to force IPv4 (when IPv6 is available):

   ```
   ip.global.version = "4"
   ```

2. When using IPv6 for link-local addresses, if several network interfaces exist on the machine, you can explicitely specify what interface must be used with the `ip.global.v6.interface.name` or `ip.global.v6.interface.id` entry in `fglprofile`.

   In order to specify the IPv6 network interface by name, use:

   ```
   ip.global.v6.interface.name = "eth0"
   ```

   > **Important:** The `ip.global.v6.interface.name` entry is not supported on Microsoft™ Windows™ platforms.

   In order to specify the IPv6 network interface by id, use:

   ```
   ip.global.v6.interface.id = "2"
   ```

Authenticate the WS client to a server
Configuration steps to authenticate the client to a server (HTTP authentication).

> **Important:** On GMI mobile devices, FGLPROFILE settings are ignored: Use the com.HTTPRequest.setAuthentication on page 2063 API instead.

1. Add HTTP authenticate entries to `fglprofile`.

   To connect to a server with HTTP Authentication, define the client login and password with the same values as registered on the server side. These entries must be defined with an unique identifier (`httpauth` in this example) to define a HTTP Authentication with "`mylogin`" as login and "`mypassword`" as password:

   ```
   authenticate.httpauth.login = "mylogin"
   authenticate.httpauth.password = "mypassword"
   ```

   See [RFC2617] for more details.

2. Encrypt the password.

   Due to security leaks, it is recommended that you NOT have a password in clear text. The Genero Web Services package provides the tool `fglpass`, which encrypts a password with a certificate that is readable only with the associated private key. To encrypt the HTTP authentication password:

a) Encrypt the clear text password with `fglpass` using the client certificate.

```
$ fglpass -e -c MyClient.crt
Enter password :mypassword
```

Note: **fglpass** outputs the encrypted password on the console but can be redirected to a file.

b) Modify the HTTP authentication password entry by specifying the security configuration to use to decrypt it (id1 in our case)

```
authenticate.id2.password.id1="HWTFu8QE2t3e5D4joy7js8mB95oOGTzLmcAor9j5DS
+C
loiliGCwZvZ9eWpfmIWSON9IwoiJheYxfnu20uaGGmmiUGiHxT6341ePXNSicu32NtlVp9t6RcS0
wN/p9a6D4XtiD9iHW7iQvXhqC9uamd3gI9Q3GhHwXOMMlY//c8Y="
```

Note: Hard returns have been added to the code sample above, for the purpose of printing and viewing within this document. The value for authenticate.id2.password.id1 is a single string with no spaces.

Note: The size of the encrypted password depends on the size of the public key, and can change according to the certificate used to encrypt it.

3. Configure the client to authenticate to a server.

As a client is able to connect to different servers that do not know the client with the same login and password, it is necessary to specify the login and password that correspond to each server. To authenticate the client known as "myclient" and with the password **passphrase** by the server **myserver**, add the following entry:

```
ws.myserver.authenticate = "httpauth"
```

Authenticate the WS client to a proxy

Configuration steps to authenticate the client to a proxy (proxy authentication).

Important: On GMI mobile devices, FGLPROFILE settings are ignored: The device configuration for proxy will always be used.

1. Add an HTTP authenticate entry to `fglprofile`.

To connect via a proxy with HTTP Proxy Authentication, it is necessary to define the client login and password as registered on the HTTP proxy.

The following two entries must be defined with an unique identifier (**proxyauth** for our example) to define a HTTP Proxy Authentication with **myapplication** as login and **mypassword** as password:

```
authenticate.proxyauth.login = "myapplication"
authenticate.proxyauth.password = "mypassword"
```

See [RFC2617] for more details.

2. For proxy authentication, an entry must be made to the HTTP proxy configuration in order to authenticate a client.

To authenticate a client known as **myapplication** and with **mypassword** as password by the HTTP Proxy, add the following entry to the HTTP proxy configuration:

```
proxy.http.authenticate = "proxyauth"
```

Note: To authenticate the client to a HTTPS proxy, replace *http* with *https*.

Configure a WS client to access an HTTPS server

Configuration steps to access a server in HTTPS.

To configure access to an HTTPS server, you will need to configure for the client certificate, configure for the certificate authority list, and the add additional entries for the server to the `fglprofile`.

> **Important:** On GMI mobile devices, FGLPROFILE settings are ignored: The device KeyChain must hold the server certificate authority.

1. Configure for the client certificate. See Configure for the client certificate on page 2466.
2. Configure for the certificate authority list. See Configure for the certificate authority list on page 2467.
3. Add configuration entries for the server to `fglprofile`.

   The Genero Web Services client needs a set of configuration entries that specify the security configuration and the HTTP authentication when accessing an HTTPS server. The following entries must be defined with an unique identifier (such as **myserver**) :

   ```
   ws.myserver.url =
   "https://www.MyServer.com/cgi-bin/fglccgi.exe/ws/r/MyWebService"
   ws.myserver.security = "id1"
   ```

   (line breaks added for document readability)

   - The unique identifier **myserver** can be used in the BDL client code in place of the actual URL.
   - The security entry value (`id1` in this example) must match the unique identifier defined by the client security entry created in 3 on page 2467.

Configure for the client certificate

You generate a client certificate and configure your application to use the client certificate generated. For production systems, you add the configuration details to `fglprofile`.

During development, if you do not have the certificate information in your `fglprofile`, Genero creates a certificate for you. When you move into production, however, the server provides a certificate for you, and you need to add the certificate information to the `fglprofile`.

1. Create the root certificate authority.
   a) Create the root certificate authority serial file.
      ```
      $ echo 01 > MyCompanyCA.srl
      ```
   b) Create the Root Authority's Certificate Signing Request and private key.
      ```
      $ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
      ```
   c) Create the Root Certificate Authority for a period of validity of 2 years.

      (line breaks added for document readability)

      ```
      $ openssl x509 -trustout -in MyCompanyCA.csr
      -out MyCompanyCA.crt -req -signkey MyCompanyCA.pem
      -days 730
      ```

      (line breaks added for document readability)

      > **Note:** The private key file (`MyCompanyCA.pem`) of a Root Certificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.
2. Create the client's X.509 certificate and private key.
   a) Create the client serial file.
      ```
      $ echo 01 > MyClient.srl
      ```
   b) Create the client's Certificate Signing Request and private key.
      ```
      $ openssl req -new -out MyClient.csr
      ```

Note: By default, `openssl` outputs the private key in the `privkey.pem` file.

c) Remove the password from the RSA private key.

```
$ openssl rsa -in privkey.pem -out MyClient.pem
```

Note: The key is also renamed in `MyClient.pem`.

d) Create the client's Certificate trusted by the Root Certificate Authority (self-signed X.509 certificate valid for a period of 1 year).

(line breaks added for document readability)

```
$ openssl x509 -in MyClient.csr -out MyClient.crt -req
-signkey MyClient.pem -CA MyCompanyCA.crt
-CAkey MyCompanyCA.pem -days 365
```

Note: Most servers do not check the identity of the clients. For these servers, the client's certificate does not necessary need to be trusted; it is only used for data encryption purpose. If, however, the server performs client identification, you must trust a Certificate Authority in which it has total confidence concerning the validity of the client's certificates.

Note: The purpose of the client's Certificate is to identify the client to any server; therefore the subject of the certificate must correspond to the client's identity as it is known by the servers.

Note: To import the certificate in a keystore you can create a pkcs12 certificate.

3. Add the client's security configuration to `fglprofile`.

The client security entry defines the certificate and the associated private key used by the Genero Web Services client during communication with an HTTPS communication. The security entry must be defined with an unique identifier (**id1**in this example).

```
security.id1.certificate = "MyClient.crt"
security.id1.privatekey = "MyClient.pem"
```

Note: If the private key is protected with a password, you must remove it or create a script that returns the password on demand.

A client certificate is created and your application is configured to use it. The client certificate is not self-signed but issued by a company, created with a root certificate.

Configure for the certificate authority list

When a client accesses a server with a certificate, the server sends back its certificate. The client needs to check to see if that certificate is trusted. This is done using a certificate authority list.

1. Create the client's certificate authority list.

a) Save the certificate of the HTTPS server to disk.

Type the server's URL in your Internet browser. When prompted, save the certificate to disk.

b) Create the client's Certificate Authority List from the certificate that you saved to disk.

```
$ openssl x509 -in ServerCertificate.crt -text >> ClientCAList.pem
```

Note: All trusted certificate authorities are listed. All other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the client.

2. Set the global certificate authority list in `fglprofile`.

The global certificate authority list entry defines the file containing the certificate authority list used by the Genero Web Services client to validate all certificates coming from the different servers it will connect to.

```
security.global.ca = "ClientCAList.pem"
```

If `security.global.ca` is not defined, Genero Web Services will look to see whether the operating system has a keystore, otherwise `security.global.ca.lookuppath` will be used.

The client application is configured to use the appropriate certificate authority list to validate a server's certificate.

Writing a Web Services server application

These topics cover creating a Genero Web Services server using the SOAP protocol.

- Writing a Web services server function on page 2468
- WS server stubs and handlers on page 2470
- Writing a Web server application on page 2473
- Get HTTP headers information at WS server side on page 2481
- Choosing a web services style on page 2483
- Web services server program deployment on page 2506
- Configuring the apache web server for HTTPS on page 2507

Writing a Web services server function

Writing a Web service with Genero is quite simple. You create a standard Genero function and publish it as a Web function (Web services operation) using methods from the classes in the COM library. There are restrictions on the function - input and output parameters are not allowed. By using global or module variables, however, to work around this exception.

See also Tutorial: Writing a GWS Server Application

The steps for writing a Web Services function:

1. Define the input parameters on page 2468
2. Define the output parameters on page 2469
3. Write the BDL function on page 2469
4. Create and publish the Web services operation on page 2469

Define the input parameters

As stated in the introduction, input parameters in Genero Web Service operations are not allowed. However, each Web Function can have one global variable or module variable that defines the input message of the function. This variable must be a record in which each field represents one of the input parameters of the Web Function.

The name of each field corresponds to the name used in the SOAP request. These fields are filled with the contents of the SOAP request by the Web Services engine just before executing the corresponding BDL function.

Example

```
DEFINE add_in RECORD
  a INTEGER,
  b INTEGER
END RECORD
```

Note: Genero version 2.0 allows you to add optional attributes to the definition of data types. You can use attributes to map the BDL data types in a Genero application to their corresponding XML data types. See Attributes to Customize XML Mapping for additional information.

Define the output parameters

Output parameters in Genero Web Functions are not allowed, but each Web Function can have one global variable or module variable that defines the output message of the function. This message must be a record where each field represents one of the output parameters of the Web Function.

The name of each field corresponds to the name used in the SOAP request. These fields are retrieved from the Web Services engine immediately after executing the BDL function, and sent back to the client.

Example

```
DEFINE add_out RECORD
  r INTEGER
END RECORD
```

Note: GWS 2.0 allows you to add optional attributes to the definition of data types. You can use attributes to map the BDL data types in a Genero application to their corresponding XML data types. See Attributes to Customize XML Mapping for additional information.

Write the BDL function

A Web Function is a normal BDL function that uses the input and output records that you have defined.

Example

```
FUNCTION add()
   LET add_out.r = add_in.a + add_in.b
END FUNCTION
```

Create and publish the Web services operation

Methods are available in the Genero Web Services library (**com**) to:

- Define the Web Service, by creating a WebService object
- Define the Web Services operation for your function, by creating a WebOperation object
- Publish the operation - associate it with the Web Service object that you defined.

The **com** library must be imported into each module of a Web Services Server application.

The following abbreviated example is from the Web Services Server tutorial:

```
IMPORT com
...
FUNCTION createservice()
    DEFINE serv  com.WebService    # A WebService
    DEFINE op    com.WebOperation  # Operation of a WebService

   --Create WebService object
   LET serv = com.WebService.CreateWebService("MyCalculator",
                       "http://tempuri.org/webservices")

   --Create WebOperation object
   LET op = com.WebOperation.CreateRPCStyle("add", "Add", add_in, add_out)

   --Publish the operation, associating it with the WebService object
   CALL serv.publishOperation(op,NULL)
...
END FUNCTION
```

See the Web Services Server tutorial and Choosing a Web Service Style for complete examples and explanations.

WS server stubs and handlers

To access a remote Web Service, you first must get the WSDL information from the service provider. Sample services can be found through UDDI registries (http://www.uddi.org), or on other sites such as XMethods (http://www.xmethods.net).

Generating files for a GWS server

You can write a Genero Web Services Server application for a Web Service that you have created; see Tutorial: Writing a Server Application.

If you want to make sure your Web Service is compatible with that of a third-party (an accounting application vendor, for example), you can use the **fglwsdl** tool to obtain the WSDL information that complies with that vendor's standards, and to generate corresponding files that can be used in your GWS Server application.

This example requests the Calculator Web Service information from the specified URL, and the output files will have the base name "ws_calculator".

```
fglwsdl -s -o ws_calculator http://localhost:8090/Calculator?WSDL
```

For a server application, fglwsdl generates two files, which should not be modified:

- *filename*`.inc` - the globals file, containing declarations of global variables that can be used as input or output to functions accessing the Web Service operations. In our example, the file is **ws_calculatorService.inc**.

  This file must be listed in a GLOBALS statement at the top of any .4gl modules that you write for your GWS Server application.
- *filename*`.4gl` - containing a function that creates the service described in the WSDL, publishes the operations of the service, and registers the service. In our example, the file is **ws_calculatorService.4gl**.

  This file must be compiled and linked into your GWS Server application.

Server handlers

The COM library enables to intercept high-level web services operation on server side. You can now define three BDL functions via the following methods of the web service class. They will be executed at different steps of a web service request processing in order to modify the SOAP request, response or the generated WSDL document before or after the SOAP engine has processed it. This helps handle **WS-*** specifications not supported in the web service API.

- Method **registerWSDLHandler()**
- Method **registerInputRequestHandler()**
- Method **registerOutputRequestHandler()**

All three kinds of BDL callback functions must conform to this prototype:

```
FUNCTION CallbackHandler( doc xml.DomDocument )
   RETURNING xml.DomDocument
```

Example 1: Modify the generation of a WSDL

Register your handler with:

```
CALL serv.registerWsdlHandler("WSDLHandler")
```

where serv is of class *com.WebService* and *WSDLHandler* is the following function:

```
FUNCTION WSDLHandler(wsdl)
  DEFINE wsdl Xml.DomDocument
  DEFINE node Xml.DomNode
  DEFINE list Xml.DomNodeList
  DEFINE ind  INTEGER
  DEFINE name STRING
  # Add a comment
  LET node = wsdl.createComment(
             "First modified WSDL via a BDL callback function")
  CALL wsdl.prependDocumentNode(node)
  # Rename input and output parameter in UPPERCASE
  LET list = wsdl.selectByXPath(
             "//wsdl:definitions/wsdl:types/xsd:schema/
xsd:complexType/xsd:sequence/xsd:element/xsd:complexType/
xsd:sequence/xsd:element",NULL)
-- first input parameter for selectByXPath above
-- one string, no spaces!
  FOR ind=1 TO list.getCount()
    LET node = list.getItem(ind)
    LET name = node.getAttribute("name")
    LET name = name.toUpperCase()
    CALL node.setAttribute("name",name)
  END FOR
  RETURN wsdl
END FUNCTION
```

If NULL is returned from the callback function, an HTTP error will be sent and the ProcessServices() returns error code -20.

Example 2: Change the SOAP incoming request

Register your handler with:

```
CALL serv.registerInputRequestHandler("InputRequestHandler")
```

where serv is of class *com.WebService* and *InputRequestHandler* is this function:

```
FUNCTION InputRequestHandler(in)
  DEFINE in Xml.DomDocument
  DEFINE ind  INTEGER
  DEFINE node Xml.DomNode
  DEFINE copy Xml.DomNode
  DEFINE tmp  Xml.DomNode
  DEFINE parent Xml.DomNode
  DEFINE name STRING
  DEFINE list Xml.DomNodeList
  # Change input parameter below myrecord in lower case
  # to follow high-level web service
  LET list = in.SelectByXPath(
    "//SOAP:Envelope/SOAP:Body/fjs:EchoDOCRecordRequest/fjs:myrecord/*",
    "SOAP","http://schemas.xmlsoap.org/soap/envelope/",
    "fjs","http://www.mycompany.com/webservices")
  FOR ind = 1 TO list.getCount()
    LET node = list.getItem(ind)
    LET parent = node.getParentNode()
    LET name = node.getLocalName()
    LET copy = in.createElementNS(node.getPrefix(),
      name.toLowerCase(),node.getNamespaceURI())
    LET tmp = node.getFirstChild()
    LET tmp = tmp.clone(true)
```

```
    CALL copy.appendChild(tmp)
    CALL parent.replaceChild(copy,node)
  END FOR
  RETURN in
END FUNCTION
```

If NULL is return from the callback function, a SOAP fault will be sent (but can be changed from the output handler) and the ProcessServices() returns error code -18.

Example 3: Modify the SOAP outgoing request

Register your handler with:

```
CALL serv.registerOutputRequestHandler("OutputRequestHandler")
```

where serv is of class *com.WebService* and *OutputRequestHandler* is this function:

```
FUNCTION OutputRequestHandler(out)
  DEFINE out Xml.DomDocument
  DEFINE ind  INTEGER
  DEFINE node Xml.DomNode
  DEFINE copy Xml.DomNode
  DEFINE tmp  Xml.DomNode
  DEFINE parent Xml.DomNode
  DEFINE name STRING
  DEFINE list Xml.DomNodeList
  # Change output parameter below myrecord in uppercase
  # before sending back to the client
  LET list = out.SelectByXPath(
    "//SOAP:Envelope/SOAP:Body/fjs:EchoDOCRecordResponse/fjs:myrecord/*",
    "SOAP","http://schemas.xmlsoap.org/soap/envelope/",
    "fjs","http://www.mycompany.com/webservices")
  FOR ind = 1 TO list.getCount()
    LET node = list.getItem(ind)
    LET parent = node.getParentNode()
    LET name = node.getLocalName()
    LET copy = out.createElementNS(node.getPrefix(),name.toUpperCase(),
      node.getNamespaceURI())
    LET tmp = node.getFirstChild()
    LET tmp = tmp.clone(true)
    CALL copy.appendChild(tmp)
    CALL parent.replaceChild(copy,node)
  END FOR
  RETURN out
END FUNCTION
```

If NULL is return from the callback function, a SOAP fault will be sent and the ProcessServices() returns error code -19.

Example output

In the generated file **ws_calculatorService.inc**, the definitions of the variables for the input and output record are the same as those generated for the Web Service Client application:

```
#VARIABLE : Add -- defines the global INPUT record
DEFINE Add RECORD ATTRIBUTES(XMLName="Add",
                             XMLNamespace="http://tempuri.org/")
  a INTEGER ATTRIBUTES(XMLName="a",XMLNamespace=""),
  b INTEGER ATTRIBUTES(XMLName="b",XMLNamespace="")
END RECORD

# VARIABLE : AddResponse -- defines the global OUTPUT record
```

```
DEFINE AddResponse RECORD ATTRIBUTES(XMLName="AddResponse",
                                     XMLNamespace="http://tempuri.org/")
  r INTEGER ATTRIBUTES(XMLName="r",XMLNamespace="")
END RECORD
```

The generated file **ws_calculatorService.4gl** contains a single function that creates the Calculator service, creates and publishes the service operations, and registers the Calculator service:

```
FUNCTION Createws_calculatorService()
  DEFINE service com.WebService
  DEFINE operation com.WebOperation
  ...  # Create Web Service
  LET service = com.WebService.CreateWebService("Calculator",
    "http://tempuri.org/")
  # Publish Operation : Add
  LET operation = com.WebOperation.CreateRPCStyle("Add","Add",
    Add,AddResponse)
  CALL service.publishOperation(operation,"")  ...
  # Register Service
  CALL com.WebServiceEngine.RegisterService(service)
  RETURN 0
  ...
END FUNCTION
```

Writing your functions

The ws_calculator.inc file provides you with the global input and output records and function names that allow you to write your own code implementing the Add operation. Your new code should not be written in the generated modules. For example, do not add your own version of the Add function to the generated ws_calculator.4gl module; it can be included in your module containing the MAIN program block, or in a separate module to be included as part of the Web server application. The function must use the generated definitions for the global input and output records.

In your version of the operation, this function adds 100 to the sum of the variables in the input record:

```
FUNCTION Add()
  LET AddResponse.r = (Add.a + Add.b) + 100
END FUNCTION
```

See Tutorial: Writing a Server application for more information. The **demo/WebServices** subdirectory of your Genero installation directory contains complete examples of Server Applications.

Writing a Web server application

This tutorial guides you through the steps to create a Server application for a Genero Web Service that can be accessed over the web by Client applications. A complete example is provided at `$FGLDIR/demo/WebServices`.

You can write your Server application based on input/output records that you have defined. Or, you can use the **fglwsdl** tool to include third-party WSDL information in your Server application.

Including the web services library

The methods associated with creating and publishing a Web Service are contained in the classes that make up the Genero Web Services Library (com). Include this line at the top of each module of your GWS Server application to import the library:

```
IMPORT com
```

Example 1: Writing the entire server application

You can define a Web Service in your application and write definitions for the input and output records that will be used by the Service. This example illustrates a Service that has one operation, **Add**, to provide the sum of two numbers.

- Step 1: Define input and output records on page 2474
- Step 2: Write a BDL function for each service operation on page 2474
- Step 3: Create the service and operations on page 2474
- Step 4: Register the service on page 2476
- Step 5: Start the GWS server and process requests on page 2476

Step 1: Define input and output records

Based on the desired functionality of the operations that you plan for the Service, define the input and output records for each operation. BDL functions that are written to implement a Web Service operation cannot have input parameters or return values. Instead, each function's input and output message must be defined as a global or module RECORD.

The Input message

The fields of the global or module record represent each of the input parameters of the Web Function. The name of each field in the record corresponds to the name used in the SOAP request. These fields are filled with the contents of the SOAP request by the Web Services engine just before executing the corresponding BDL function.

The Output message

The fields of the global or module record represent each of the output parameters of the Web Function. The name of each field in the record corresponds to the name used in the SOAP request. These fields are retrieved from the Web Services engine immediately after executing the BDL function, and sent back to the client.

Your Genero Web Services service has one planned operation that adds two integers and returns the result. The input and output records are defined as follows:

```
GLOBALS
  DEFINE
    add_in RECORD   # input record
      a INTEGER,
      b INTEGER
    END RECORD,
    add_out RECORD  # output record
      r INTEGER
    END RECORD
END GLOBALS
```

Step 2: Write a BDL function for each service operation

You will need to write a function to implement each operation, using the input and output global records.

To implement your **Add** operation:

```
#User Public Functions
FUNCTION add()
  LET add_out.r = add_in.a + add_in.b
END FUNCTION
```

Step 3: Create the service and operations

The Genero Web Services library (**com**) provides classes and methods that allow you to use Genero BDL to configure a Web Service and its operations.

- **WebService** - this is a container for web operations.
- **WebOperation** - describes the operation.

Define variables for the WebService and WebOperation objects

```
FUNCTION createservice()
  DEFINE serv  com.WebService    # A WebService
  DEFINE op    com.WebOperation  # Operation of a WebService
```

Choose a Namespace

XML uses namespaces to group the element and attribute definitions, and to avoid conflicting names. In practice, a namespace must be a unique identifier (URI: Uniform Resource Identifier). If you do not know the unique identifier to use, your company's Web site domain name is guaranteed to be unique (such as "www.mycompany.com"); then, append any string.

Examples of valid namespaces for the fictional My Company company:

- **"http://www.mycompany.com/MyServices"**
- **"http://www.mycompany.com/any_string"**

Another option (for testing only) is to use the temporary namespace "http://tempuri.org/".

Create the WebService object

Call the constructor method of the WebService class. The parameters are:

1. Service name
2. Valid namespace

This example uses the temporary namespace and creates a Service named "MyCalculator".

```
LET serv =
  com.WebService.CreateWebService("MyCalculator", "http://tempuri.org/
webservices")
```

Create the WebOperation object

A WebService object can have multiple operations. The operations can be created in RPC style or Document style by calling the corresponding constructor method of the WebOperation class. The parameters are:

1. the name of the BDL function that is executed to process the XML operation
2. the name you wish to assign to the XML operation
3. the input record defining the input parameters of the operation (or NULL if there is none)
4. the output record defining the output parameters of the operation (or NULL if there is none)

To create the operation for the previously defined **add** function in RPC style:

```
LET op = com.WebOperation.CreateRPCStyle("add", "Add", add_in, add_out)
```

To create the operation for the previously defined **add** function in Document style:

```
LET op = com.WebOperation.CreateDOCStyle("add", "Add", add_in, add_out)
```

Mixing RPC style and Document style operations in the same service is not recommended, as it is not WS-I compliant. See Web Services Styles for additional information about styles.

The rest of the code in your application is the same, regardless of the Web Services style that you have chosen.

Publish the operation

Once an operation is defined, it must be associated with its corresponding WebService (the operation must be published). The publishOperation method of the WebService object has the following parameters:

- the WebOperation to be published
- a string to identify the operation if several operations have the same name; if this is NULL, the default value is an empty string

For example, to publish the **Add** operation of the **Calculator** service, which was defined as **op**:

```
CALL serv.publishOperation(op,NULL)
```

Step 4: Register the service

Once the Service and operations are defined and the operations are published, the WebService and WebOperation objects have completed their work. Registering a service puts the Genero DVM in charge of the execution of all the operations of that service - dispatching the incoming message to the right service, returning the correct output, and so on. The same service may be registered at different locations on the Web.

The WebServiceEngine is a global built-in object that manages the Server part of the Genero DVM. Use the **RegisterService** class method of the WebServiceEngine class. The parameter is:

1. The name of the WebService object

To register the Calculator service example created in Step 3: Create the service and operations on page 2474:

```
CALL com.WebServiceEngine.RegisterService(serv)
END FUNCTION
```

> **Note:** If you wanted to create a single GWS Server DVM containing multiple Web Services, you could define additional input and output records and repeat steps 2 through 6 for each Web Service. In Step 5: Start the GWS server and process requests on page 2476, a GWS Server DVM is started, containing as many Web Services as you have defined. See Web services server program deployment on page 2506 for additional discussion of GWS Services and GWS Servers.

Step 5: Start the GWS server and process requests
Once you have registered the Web Service(s), you are ready to start the Genero Web Services (GWS) Server and process the incoming SOAP requests.

The GWS Server is located on the same physical machine where the application is being executed (In other words, where `fglrun` executes).

This is the `MAIN` program block of your application.

Define a variable for status

Define a variable to hold the returned status of the request:

```
MAIN
  DEFINE ret INTEGER
```

Call the function that you created, which defined and registered the service and its operations:

```
  CALL createservice()
```

Start the GWS Server

Use the `Start` class method of the `WebServiceEngine` class to start the server.

```
  CALL com.WebServiceEngine.Start()
```

Process the requests

This example uses the `ProcessServices` method of the `WebServiceEngine` class to process each incoming request. It returns an integer representing the status. The parameter specifies the timeout period (in seconds) for which the method should wait to process a service. The value -1 specifies an infinite waiting time.

```
  WHILE TRUE
    # Process each incoming requests (infinite loop)
    LET ret = com.WebServiceEngine.ProcessServices(-1)
    CASE ret
      WHEN 0
        DISPLAY "Request processed."
      WHEN -1
        DISPLAY "Timeout reached."
      WHEN -2
        DISPLAY "Disconnected from application server."
        EXIT PROGRAM
      WHEN -3
        DISPLAY "Client Connection lost."
      WHEN -4
        DISPLAY "Server interrupted with Ctrl-C."
      WHEN -10
        DISPLAY "Internal server error."
        EXIT PROGRAM
      WHEN -15
        DISPLAY "Server was not started."
        EXIT PROGRAM
      OTHERWISE
        DISPLAY "ERROR: ", STATUS, SQLCA.SQLERRM
    END CASE
    IF int_flag<>0 THEN
      LET int_flag=0
      EXIT WHILE
    END IF
  END WHILE

  DISPLAY "Server stopped"

END MAIN
```

> **Note:** For testing purposes only, the GWS Server can be started in standalone mode. In a production environment, the Genero Application Server (GAS) is required to manage your application. For deployment, the GWS Server application must be added to the GAS configuration. See *Adding Applications* in the *Genero Application Server User Guide*.

Example 2: Writing a server using third-party WSDL (the fglwsdl tool)

To write a Web Service that is compatible with the specification of the input and output records defined by a third-party (for example, a vendor of manufacturing software, or a WSDL specialist in your company) you can use the `fglwsdl` tool to obtain the WSDL information and generate a part of the Server application. See fglwsdl on page 1503 for a complete description of the tool and its use.

- Step 1: Get the WSDL description and generate files on page 2478
- Step 2: Write a BDL function for your service operation on page 2479
- Step 3: Create service, start server and process requests on page 2479

Step 1: Get the WSDL description and generate files

This tutorial uses fglwsdl and the Calculator Service defined in Example 1: Writing the entire server application on page 2474 to obtain the WSDL information and generate two corresponding BDL files:

- the globals file, containing declarations of global variables that can be used as input or output to functions accessing the Web Service operations.
- a .4gl file containing a function that creates the service described in the WSDL, publishes the operations of the service, and registers the service.

```
fglwsdl -s -o example1 http://localhost:8090/MyCalculator?WSDL
```

> **Note:** the MyCalculator Genero Web Services Service created in Example 1: Writing the entire server application on page 2474 must be running in order to obtain the WSDL information.

The generated globals file

The globals file **example1Service.inc** provides the definition of the global input and output records as described in the Step 1: Define input and output records on page 2474 of the Example 1: Writing the entire server application on page 2474 GWS Server program. The names of the input and output records have been assigned by **fglwsdl**, in accordance with the Style of the Web Service MyCalculator (created as RPCStyle in the Example1 program). Do not modify this file.

Input and output records:

```
# VARIABLE : Add
DEFINE Add RECORD
           ATTRIBUTES( XMLName="Add",
                       XMLNamespace="http://tempuri.org/webservices" )
  a INTEGER ATTRIBUTES(XMLName="a",XMLNamespace=""),
  b INTEGER ATTRIBUTES(XMLName="b",XMLNamespace="")
END RECORD

# VARIABLE : AddResponse
DEFINE AddResponse RECORD
                   ATTRIBUTES(XMLName="AddResponse",
                              XMLNamespace="http://tempuri.org/webservices" )
  r INTEGER ATTRIBUTES(XMLName="r",XMLNamespace="")
END RECORD
```

The generated .4gl file

The **example1Service.4gl**file contains a single function that creates the service, publishes the operation, and registers the Service. The Web Service Style that is created is determined by the style specified in the WSDL information. The functions in this file accomplish the same tasks as Step 3: Create the service and operations on page 2474 and Step 4: Register the service on page 2476 of Example 1. Do not modify this file.

```
# example1Service.4gl
# Generated file containing the function Createexample1Service

IMPORT com
GLOBALS "example1Service.inc"

# FUNCTION Createexample1Service
#   RETURNING soapstatus
FUNCTION Createexample1Service()
DEFINE service      com.WebService
DEFINE operation    com.WebOperation
  # Set ERROR handler
WHENEVER ANY ERROR GOTO error
```

```
  # Create Web Service
LET service = com.WebService.CreateWebService(
                             "MyCalculator",
                             "http://tempuri.org/webservices")
  # Operation: Add
  # Publish Operation : Add
LET operation = com.WebOperation.CreateRPCStyle(
                                "Add",
                                "Add",
                                Add,
                                AddResponse)
CALL service.publishOperation(operation,"")
  # Register Service
CALL com.WebServiceEngine.RegisterService(service)
RETURN 0
  # ERROR handler
LABEL error:
RETURN STATUS
  # Unset ERROR handler
WHENEVER ANY ERROR STOP
END FUNCTION
```

Step 2: Write a BDL function for your service operation

Using the information from these generated files, the Add operation from Example 1: Writing the entire server application on page 2474 is rewritten to have different functionality but to still be compatible with the WSDL description of the operation. This step accomplishes the same thing as Step 2: Write a BDL function for each service operation on page 2474 in Example 1. In this version of the add operation, the sum of the two numbers in the input record is increased by 100.

```
# my_function.4gl                           -- file containing the function
                                            -- definition
IMPORT com                                  -- import the Web Services library

GLOBALS "example1Service.inc"               -- use the generated globals file
#User Public Functions
FUNCTION add()                              -- new version of the add function
LET AddResponse.r = (Add.a + Add.b)+ 100    -- the global input and output
                                            -- records are used
END FUNCTION
```

Step 3: Create service, start server and process requests

Create your own Main module that calls the function from the generated .4gl file to create the service, and then starts the Genero Web Services Server and manages requests as in Step 5: Start the GWS server and process requests on page 2476 of Example 1: Writing the entire server application on page 2474.

```
# example2main.4gl file -- contains the MAIN program block

IMPORT com

GLOBALS "example1Service.inc"

MAIN
   DEFINE create_status INTEGER

   DEFER INTERRUPT

   CALL Createexample1Service()  -- call the function generated
                                 -- in example1Service.4gl
      RETURNING create_status
   IF create_status <> 0 THEN
      DISPLAY "error"
```

```
   ELSE
      # Start the server and manage requests
      CALL ManageService()
   END IF

END MAIN

FUNCTION ManageService()
   DEFINE ret INTEGER
   CALL com.WebServiceEngine.start()
   WHILE TRUE
   # continue as in Step 5 of Example 1
   ...
END FUNCTION
```

Compiling GWS server applications

The library file `WSHelper.42m`, included in the $FGLDIR/lib directory of the Genero Web Services package, should be linked into every GWS Server application.

If your application uses the `fglwsdl` tool to generate information, link the .4gl generated file into the application.

Examples

Compiling the Example 1: Writing the entire server application on page 2474 program:

```
fglcomp example1.4gl
fgllink -o example1.42r example1.42m WSHelper.42m
```

Compiling the Example 2: Writing a server using third-party WSDL (the fglwsdl tool) on page 2477 program:

```
fglcomp example2main.4gl my_function.4gl example1Service.4gl
fgllink - o example2.42r example2main.42m my function.42m
   example1Service.42m WSHelper.42m
```

Testing the GWS service in stand-alone mode

For testing and development purposes only, the Genero Web Services Server application can be executed directly, without using the Genero Application Server (GAS).

1. Use the Genero `fglrun` command to execute the GWS Server application, which must reside on the same machine:

   ```
   fglrun <gws application>
   ```

   This will start the GWS Server on the port specified by the FGLAPPSERVER environment variable. If this environment variable is not set for the user, port number 80 is used. For example, if FGLAPPSERVER is set to 8090, the server will be started on that port.

   > **Note:** The user must not set the FGLAPPSERVER variable in production environments, since the port number is selected by the Genero Application Server.

2. Obtain the WSDL information for your Service and write a test Client application. If the GWS Server in step 1 was started on your local machine, for example, the command to get the WSDL information would be:

   ```
   fglwsdl -o <test-client> http://localhost:8090/<service-name>?WSDL
   ```

Configuring the Genero application server for the GWS Application

The final step is to configure the Genero Application Server (GAS) to handle the GWS application. In a production environment, Genero Web Services becomes a part of a global application architecture handled by the application server of the **GAS** package. See Web services server program deployment on page 2506, as well as *Adding Applications* in the GAS manual.

Making the GWS service available

Once you compile and deploy your Genero Web Services Server application (see Web services server program deployment on page 2506), it can be used by others to obtain the WSDL information and write a client application that accesses your Genero Web Service. See Steps to write a WS client on page 2450.

Your company can provide the location of the GWS Server to potential users of your Web Service in various ways. For example:

- Provide the location on a company web site
- Register the Web Service with UDDI (Universal Description, Discovery, and Integration) - the XML-based registry providing Internet listings for companies worldwide
- Communicate directly with your potential users

Get HTTP headers information at WS server side

In high level web services, we now give access to the HTTP headers request and response.

The web service can get information from the request headers and reply with custom headers and status.

1. Declare variables to receive or send HTTP headers.
2. Register these variables to the web service server.

Declare variables to receive or send HTTP headers

The variable for the request headers:

```
DEFINE http_in RECORD
           verb STRING,
           url STRING,
           headers DYNAMIC ARRAY OF RECORD
                     name STRING,
                     value STRING
                   END RECORD
           END RECORD
```

After the web service operation has been processed, the variable is set to NULL.

The variable for the response headers:

```
DEFINE http_out RECORD
           code INTEGER,
           desc STRING,
           headers DYNAMIC ARRAY OF RECORD
                     name STRING,
                     value STRING
                   END RECORD
         END RECORD
```

After the web service operation has been processed, the variable is set to NULL.

> **Note:** While the variables must follow the structure shown, the variable name can be any name you choose.

The web service engine headers have precedence. For example, if you set the "Content-Length" value, the one that is taken into account is the one defined by the Genero Web Services engine.

Register the variables to the server

This code example uses two methods, which use the defined variables:

- `com.WebService.registerInputHttpVariable(`*`http_in`*`)` where *http_in* is the RECORD variable for the request headers.
- `com.WebService.registerOutputHttpVariable(`*`http_out`*`)` where *http_out* is the RECORD variable for the response headers

Example

```
FUNCTION CreateService()

  DEFINE serv com.WebService # WebService
  DEFINE op com.WebOperation # Operation of a WebService

  TRY
    #
    # Create a Web Service
    #
    LET serv = com.WebService.CreateWebService("EchoHttpHeadersService",
                                                Namespace)
    #
    # Create Document Style Operations
    #
    # EchoDOCRecord
    LET op = com.WebOperation.CreateDOCStyle("echoDocRecord",
                                             "EchoDOCRecord",
                                             echoRecordDoc_in,
                                             echoRecordDoc_out)
    CALL serv.publishOperation(op,NULL)

    # Register HTTP input
    CALL serv.registerInputHttpVariable(http_in)

    # Register HTTP output
    CALL serv.registerOutputHttpVariable(http_out)

    #
    # Register service
    #
    CALL com.WebServiceEngine.RegisterService(serv)
    DISPLAY "EchoHttpHeadersService Service registered"
    CATCH
       DISPLAY "Unable to create 'EchoHttpHeadersService' Web Service : ",
               STATUS||" ("||SQLCA.SQLERRM||")"
       EXIT PROGRAM (-1)
    END TRY

END FUNCTION

FUNCTION echoDocRecord()
  DEFINE ind INTEGER
  DEFINE ok BOOLEAN

  # Check incoming VERB
  IF http_in.verb != "POST" THEN
    LET http_out.code = 400
    LET http_out.desc = "Bad request: method should be POST"
    RETURN
  END IF

  # Check incoming query string
  IF http_in.url.getIndexOF("?MyQuery=OK",1)<=0 THEN
```

```
    LET http_out.code = 400
    LET http_out.desc = "Bad request: URL should have MyQuery=OK"
    RETURN
  END IF

  # Check incoming header called MyPersonal
  LET ok = FALSE
  FOR ind = 1 TO http_in.headers.getLength()
    DISPLAY ind||"# ",http_in.headers[ind].name,
            "=",http_in.headers[ind].value
    IF http_in.headers[ind].name == "MyPersonal" THEN
      IF http_in.headers[ind].value == "Header" THEN
        LET ok = TRUE
      END IF
    END IF
  END FOR
  IF NOT ok THEN
    LET http_out.code = 400
    LET http_out.desc =
        "Bad request: expected additional header called MyPersonal"
    RETURN
  END IF

  # assign the output record
  LET echoRecordDoc_out.MyRecord.MyInt =
 echoRecordDoc_in.MyRecord.MyInt
  LET echoRecordDoc_out.MyRecord.MyFloat =
 echoRecordDoc_in.MyRecord.MyFloat

  # Add MyPersonalHeader=MyPersonalValue http headers
  LET http_out.headers[1].name = "MyPersonalHeader"
  LET http_out.headers[1].value = "MyPersonalValue"

END FUNCTION
```

Choosing a web services style

Genero Web Services 2.0 allows you to create Web Services operations in the following styles:

Table 555: Web Services Styles

Web Services Style	Description
RPC Style Service (RPC/Literal)	Generally used to execute a function, such as a service that returns a stock option.
Document Style Service (Doc/Literal)	Generally used for more sophisticated operations that exchange complex data structures, such as a service that sends an invoice to an application, or exchanges a Word document; this is the MS.Net default. Both RPC/Literal and Doc/Literal Styles are WS-I compliant (Web Services Interoperability organization).
RPC Style Service (RPC/Encoded)	Provided only for backwards compatibility with older versions of web services already published. **Important:** This style is deprecated by the WS-I organization, and is not recommended, as most Web Service

Web Services Style	Description
	implementations won't support it in the future.

The style of service to be created is specified in the Genero application for the Web Service, using the following methods of the WebOperation class from from the Web Services COM Library (**com**). The parameters are the same for both methods:

1. The name of the BDL function that is executed to process the Web Service operation
2. The name you wish to assign to the Web Service operation
3. The input record defining the input parameters of the operation (or NULL if there is none)
4. The output record defining the output parameters of the operation (or NULL if there is none)

```
LET op = com.WebOperation.CreateRPCStyle("add","Add",
           add_in,add_out)
LET op = com.WebOperation.CreateDOCStyle("checkInvoice",
           "CheckInvoice",invoice_in,invoice_out)
```

Calling the appropriate function for the desired style is the only difference in your Genero code that creates the service. The remainder of the code that describes the service is the same, regardless of whether you want to create an RPC or Document style of service.

> **Important:** Do not use the setInputEncoded() and setOutputEncoded() methods of the WebService class from the Web Services COM Library (**com**), as they apply only to RPC/Encoded Style, which is not recommended.
>
> **Note:** If you add headers to your RPC Style service, choose the Literal serialization mechanism by setting the **encoded** parameter of the createHeader() method to FALSE:
>
> ```
> CALL serv.createHeader(var,FALSE)
> ```
>
> **Note:** GWS release 2.0 allows you to create RPC Style and Document Style operations in the same Web Service. However, we do not recommend this, as it is not WS-I compliant.

How To's

These topics provide you with the information needed to perform specific tasks related to Genero Web Services using the SOAP protocol.

- How to call Java APIs from Genero in a SOA environment on page 2484
- How to call .NET APIs from Genero in a SOA environment on page 2490
- Compute a hash value from a BDL string on page 2497
- Fix Genero 2.10 to 2.11 WSDL generation issue on page 2499
- How to handle WS security on page 2501

How to call Java™ APIs from Genero in a SOA environment

Overview

This tutorial explains how to call a Java™ library from Genero in a SOA environment, using Genero and Java™ Web services. This can easily be done using the Java™ JAX-WS framework on a server, and a Genero application for the client part. Notice that there is no strong linkage between Genero and a java JVM.

For this tutorial we will use a Java™ barcode creation library to build a picture from a code.

> **Note:** Accessing a .NET library could be done in the same manner.

Recommendation

The usage of Genero Web Services to call a Java™ service is recommended in a **SOA** environment. It enables several Genero applications to connect to a centralized Java™ service without the need to start a new JVM for each running Genero application. It also provides more flexibility because there is no strong linkage between Genero and the Java™ virtual machine. You can for instance upgrade the Java™ service without changing anything in your Genero code.

However, due to the XML serialization process and the HTTP transport protocol in Web Services, there can be some performance issues. So if your main concern is performance, it is recommended to use the Genero Java™ bridge.

Prerequisites

- A JRE 1.5 or above
- The Java™ barcode library (available here)
  - You must add these JARs to the Java™ CLASSPATH: **barcode.jar** and **BarcodeReader.jar**
  - The trial version has some functions partially implemented.
- Download the JAX-WS framework from the Sun metro project here; add this JAR to the java CLASSPATH: **webservices-tools.jar**

Using the barcode library

The **barcode library** is composed of two libraries:

- A library for building a barcode image from a numeric code
- A library for reading a barcode image to return the numeric code

This section depends on the library you want to use in Genero.

In our tutorial, we create two functions called **buildImage** and **readImage**.

This is the Java™ implementation:

```
buildImage( type : String, code : String) : byte[ ]
```

```
try {
  Barcode builder=new Barcode();
  builder.setType(GetBarcodeBuilderType(type));
  builder.setData(data);
  builder.setAddCheckSum(true);
  ByteArrayOutputStream out=new ByteArrayOutputStream();
  if (builder.createBarcodeImage(out)) {
   byte[] ret = out.toByteArray();
   return ret;
 } else {
   return null;
 }
} catch (Exception e) {
  return null;
}
```

```
readImage( type : String, img : byte[ ]) : String
```

```
try {
  File f=new File("tmp.jpg");
  FileOutputStream stream=new FileOutputStream(f);
  stream.write(img);
  stream.close();  String[] datas =
    BarcodeReader.read(f, GetBarcodeReaderType(type));
  if (datas==null) {
    return null;
  } else {
```

```
    String ret = datas[0];
    return ret;
  }
} catch (Exception e) {
  return null;
}
```

The following two functions convert the type of a code bar to the type expected by the library:

```
private int GetBarcodeBuilderType(String str) {
  if (str.equals("CODABAR")) {
    return Barcode.CODABAR;
  } else if (str.equals("CODE11")) {
    return Barcode.CODE11;
  } else if (str.equals("CODE128")) {
    return Barcode.CODE128;
  } else if (str.equals("CODE128A")) {
    return Barcode.CODE128A;
  } else if (str.equals("CODE128B")) {
    return Barcode.CODE128B;
  } else if (str.equals("CODE128C")) {
    return Barcode.CODE128C;
  } else if (str.equals("CODE2OF5")) {
    return Barcode.CODE2OF5;
  } else if (str.equals("CODE39")) {
    return Barcode.CODE39;
  } else if (str.equals("CODE39EX")) {
    return Barcode.CODE39EX;
  } else if (str.equals("CODE93")) {
    return Barcode.CODE93;
  } else if (str.equals("CODE93EX")) {
    return Barcode.CODE93EX;
  } else if (str.equals("EAN13")) {
    return Barcode.EAN13;
  } else if (str.equals("EAN13_2")) {
    return Barcode.EAN13_2;
  } else if (str.equals("EAN13_5")) {
    return Barcode.EAN13_5;
  } else if (str.equals("EAN8")) {
    return Barcode.EAN8;
  } else if (str.equals("EAN8_2")) {
    return Barcode.EAN8_2;
  } else if (str.equals("EAN8_5")) {
    return Barcode.EAN8_5;
  } else if (str.equals("INTERLEAVED25")) {
    return Barcode.INTERLEAVED25;
  } else if (str.equals("ITF14")) {
    return Barcode.ITF14;
  } else if (str.equals("ONECODE")) {
    return Barcode.ONECODE;
  } else if (str.equals("PLANET")) {
    return Barcode.PLANET;
  } else if (str.equals("POSTNET")) {
    return Barcode.POSTNET;
  } else if (str.equals("RM4SCC")) {
    return Barcode.RM4SCC;
  } else if (str.equals("UPCA")) {
    return Barcode.UPCA;
  } else if (str.equals("UPCE")) {
    return Barcode.UPCE;
  } else {
    return -1;
  }
```

```
}

private int GetBarcodeReaderType(String str) {
  if (str.equals("CODABAR")) {
    return BarcodeReader.CODABAR;
  } else if (str.equals("CODE11")) {
    return BarcodeReader.CODE11;
  } else if (str.equals("CODE128")) {
    return BarcodeReader.CODE128;
  } else if (str.equals("CODE39")) {
    return BarcodeReader.CODE39;
  } else if (str.equals("CODE39EX")) {
    return BarcodeReader.CODE39EX;
  } else if (str.equals("CODE93")) {
    return BarcodeReader.CODE93;
  } else if (str.equals("DATAMATRIX")) {
    return BarcodeReader.DATAMATRIX;
  } else if (str.equals("EAN13")) {
    return BarcodeReader.EAN13;
  } else if (str.equals("EAN8")) {
    return BarcodeReader.EAN8;
  } else if (str.equals("INTERLEAVED25")) {
    return BarcodeReader.INTERLEAVED25;
  } else if (str.equals("ITF14")) {
    return BarcodeReader.ITF14;
  } else if (str.equals("ONECODE")) {
    return BarcodeReader.ONECODE;
  } else if (str.equals("PLANET")) {
    return BarcodeReader.PLANET;
  } else if (str.equals("POSTNET")) {
    return BarcodeReader.POSTNET;
  } else if (str.equals("QRCODE")) {
    return BarcodeReader.QRCODE;
  } else if (str.equals("RM4SCC")) {
    return BarcodeReader.RM4SCC;
  } else if (str.equals("RSS14")) {
    return BarcodeReader.RSS14;
  } else if (str.equals("RSSLIMITED")) {
    return BarcodeReader.RSSLIMITED;
  } else if (str.equals("UPCA")) {
    return BarcodeReader.UPCA;
  } else if (str.equals("UPCE")) {
    return BarcodeReader.UPCE;
  } else {
    return -1;
  }
}
```

Calling Java™ from Genero

The integration of one or several Java™ libraries with multiple methods in a Genero application can be performed, as described in the following topics.

Step 1: Write a new java class

Instead of writing the functions in 4GL, you simply need to write them in a Java™ class with the methods you want to use in 4GL. In our example, the two functions are **buildImage** and **readImage**. And of course, don't forget to import the necessary Java™ import instructions.

```
import com.barcodelib.barcodereader.BarcodeReader;
import com.barcodelib.barcode.Barcode;
import java.io.*;
```

```
import javax.jws.*;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.Endpoint;

public class BarcodeService {
  public byte [] buildImage(String type, String data)
  {
  /*BUILDIMAGE IMPLEMENTATION CODE DESCRIBED ABOVE*/
  }
  public String readImage(String type,byte[] img)
  { { { {
  /*READIMAGE IMPLEMENTATION CODE DESCRIBED ABOVE*/
  } }
}
```

Notice that if you want the service to run standalone, you must also add following the main method to tell the system the port number on which the service will run:

```
public static void main(String[] args)
{
  String endpointUri = "http://localhost:9090/";
  Endpoint.publish(endpointUri, new BarcodeService ());
  System.out.println("BarcodeService started at " + endpointUri);
}
```

Step 2: Transform the Java™ class in a web service

To transform the previous java class in a Web Service, simply add a WebService annotation:

```
@WebService(targetNamespace = http://www.mycompany.com/barcode ",
  name="Barcode",
  serviceName="BarcodeService")
public class BarcodeService{
...
}
```

This defines all public and non static methods of the class as operations of the **BarcodeService** Web Service.

Step 3: Start the service

Compile the previously created java class, and run it.

Commands to compile and execute the service in standalone mode:

```
$ javac BarcodeService.java
$ java BarcodeService
```

Once the service is started, it is ready to accept requests and you can also retrieve its WSDL at following URL:

http://localhost/9090/BarcodeService?WSDL

> **Note:** If you want the service to be started on a web server, you must deploy it first using Eclipse or the Web Server deployment tools.

Step 4: Generate BDL stub to access the Java™ library

Use the **fglwsdl** tool to generate the client stub to access the BarcodeService:

```
$ fglwsdl http://localhost:9090/BarcodeService?WSDL
```

This will create two .4gl files that must be compiled and linked into your BDL application in order to call the Java™ barcode library functions. These files contain the BDL interface to access the Java™ library where you will find the two functions, **readImage** and **buildImage**, defined in BDL.

Step 5: Modify your BDL application

The last step is to modify the existing application where you want to use the Java™ library, by calling the BDL functions generated in the stub. Then compile your application and the previously generated stubs, and link everything together.

Your application is now ready to use the different features of your Java™ library.

Example program

This program calls the **buildImage** function of the Barcode Java™ library.

```
GLOBALS "BarcodeService_BarcodePort.inc"

MAIN

DEFINE wsstatus INTEGER

IF num_args() != 3 THEN
   CALL ExitHelp()
END IF

LET ns1buildImage.arg0 = arg_val(1)
LET ns1buildImage.arg1 = arg_val(2)
LOCATE ns1buildImageResponse.return IN MEMORY

LET wsstatus = buildImage_g()
IF wsstatus <> 0 THEN
  DISPLAY "Error ("||wsError.code||") : ",wsError.description
ELSE IF
  IF ns1buildImageResponse.return IS NULL THEN
    DISPLAY "Encoding failed"
  ELSE
    CALL ns1buildImageResponse.return.writeFile(arg_val(3))
  END IF
END IF

FREE ns1buildImageResponse.return

END MAIN

FUNCTION ExitHelp()
  DISPLAY arg_val(0)||" <type> <data> <filename>"
  DISPLAY "type : barcode type such as EAN8 or CODE128"
  DISPLAY "data : data to be encoded with a barcode [0-9A-D]"
  DISPLAY "filename : resulting image filename"
  DISPLAY "example : createImage EAN8 12358723A mybarcode.jpg"
 EXIT PROGRAM (-1)
END FUNCTION
```

Conclusion

In a SOA environment, you can call any Java™ library from Genero using Web Services, and without a strong dependency to a JVM. This follows SOA principles - it allows you to reuse the Java™ library in another BDL application without any new development, you can update the Java™ part without recompiling any .4gl source, and integrate any function available from a SOA platform.

How to call .NET APIs from Genero in a SOA environment

Overview

This document explains how to call a .NET library from Genero in a SOA environment, using Genero and Web services, and IIS and Visual Studio .NET.

Notice that there is no strong linkage between Genero and .NET, and you can even call a .NET library from a non-Windows Genero platform.

For the tutorial, we will use a .NET barcode creation library to build a picture from a numeric code, and C# as the development language. This will also work with any other .NET language.

> **Note:** Accessing a Java™ library could be done in the same manner.

Prerequisites

- IIS (Internet Information Services) Web server
- Visual Studio Professional Edition C#
  - Visual Studio in only needed for development. Once the service is built, you can deploy on any IIS Web Server.
- The .NET barcode library (available here)
  - The trial version has some functions partially implemented.
  - The .NET library is called BarcodeLib.Barcode.dll, and must be added to the Visual Studio Project.

Using the barcode library

This section depends on the library you want to use in Genero. In our tutorial, we create one function called **buildImage**. This is the C# implementation:

```
buildImage( type : String, code : String) : byte[]
```

```
Linear barcode = new Linear();
barcode.Data = code;
barcode.Type = GetBarcodeBuilderType(type);
barcode.AddCheckSum = true;
// save barcode image into your system
barcode.ShowText = true;
byte[] ret = barcode.drawBarcodeAsBytes();
if (ret != null) return ret;
else return null;
```

You will also need to convert the type of a code bar to the right type as expected by the library. Therefore, you will need this function.

```
private BarcodeType GetBarcodeBuilderType(String str)
{
if (str.Equals("CODABAR")) {
      return BarcodeType.CODABAR;
   } else if (str.Equals("CODE11")) {
      return BarcodeType.CODE11;
   } else if (str.Equals("CODE128")) {
      return BarcodeType.CODE128;
   } else if (str.Equals("CODE128A")) {
      return BarcodeType.CODE128A;
   } else if (str.Equals("CODE128B")) {
      return BarcodeType.CODE128B;
   } else if (str.Equals("CODE128C")) {
      return BarcodeType.CODE128C;
   } else if (str.Equals("CODE2OF5")) {
      return BarcodeType.CODE2OF5;
   } else if (str.Equals("CODE39")) {
      return BarcodeType.CODE39;
```

```
    } else if (str.Equals("CODE39EX")) {
       return BarcodeType.CODE39EX;
    } else if (str.Equals("CODE93")) {
       return BarcodeType.CODE93;
    } else if (str.Equals("EAN13")) {
       return BarcodeType.EAN13;
    } else if (str.Equals("EAN13_2")) {
       return BarcodeType.EAN13_2;
    } else if (str.Equals("EAN13_5")) {
       return BarcodeType.EAN13_5;
    } else if (str.Equals("EAN8")) {
       return BarcodeType.EAN8;
    } else if (str.Equals("EAN8_2")) {
       return BarcodeType.EAN8_2;
    } else if (str.Equals("EAN8_5")) {
       return BarcodeType.EAN8_5;
    } else if (str.Equals("INTERLEAVED25")) {
       return BarcodeType.INTERLEAVED25;
    } else if (str.Equals("ITF14")) {
       return BarcodeType.ITF14;
    } else if (str.Equals("ONECODE")) {
       return BarcodeType.ONECODE;
    } else if (str.Equals("PLANET")) {
       return BarcodeType.PLANET;
    } else if (str.Equals("POSTNET")) {
       return BarcodeType.POSTNET;
    } else if (str.Equals("RM4SCC")) {
       return BarcodeType.RM4SCC;
    } else if (str.Equals("UPCA")) {
       return BarcodeType.UPCA;
    } else if (str.Equals("UPCE")) {
       return BarcodeType.UPCE;
    } else {
       throw new Exception();
    }
}
```

Calling .NET from Genero

Step 1: Create an ASP.NET Web Service Application

Start Visual Studio, and create a new web project with the name BarCodeService.

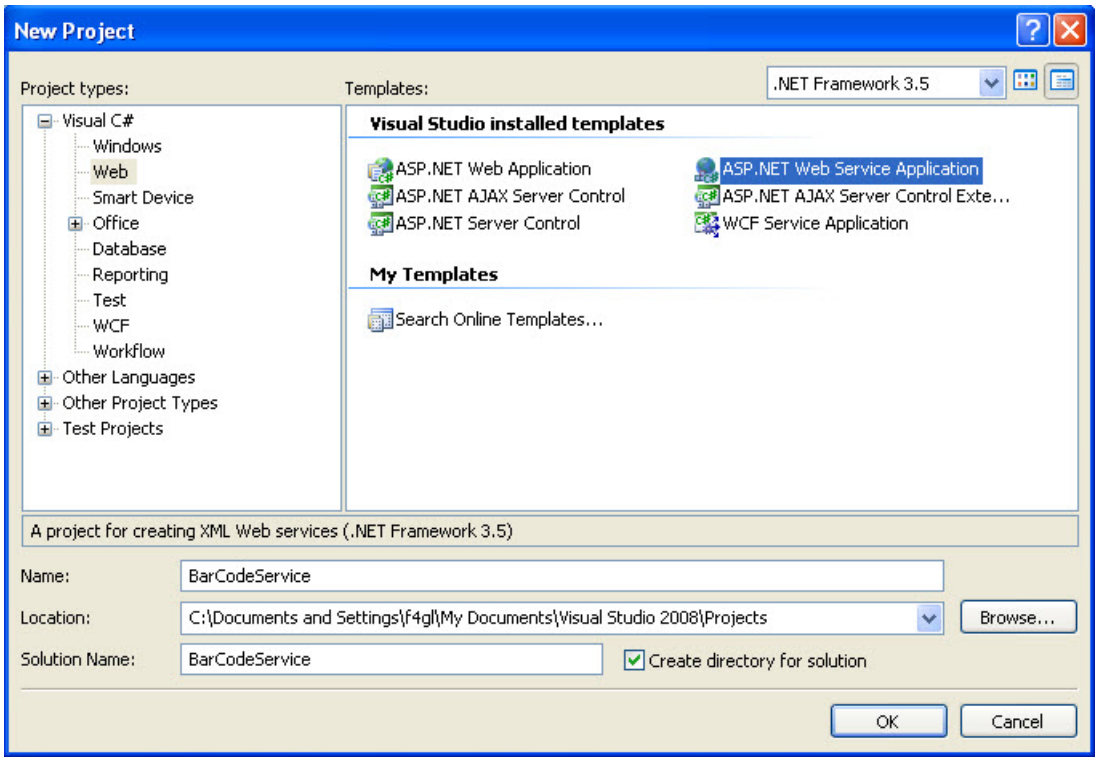


Figure 109: Visual Studio New Project; ASP .NET Web Service Application selected

Step 2: Rename the generated files

Rename the generated class called Service1 with an appropriate name such as BarCode, and the file Service1.asmx to BarCodeService.asmx, for instance. The .asmx file is the file that is accessible from the IIS web server once the application is deployed. The .asmx file also contains a reference to the default generated class, Service1, which must also be renamed to the new name (BarCode in our tutorial), in case Visual Studio didn't make the change automatically.

The class view after renaming the class:

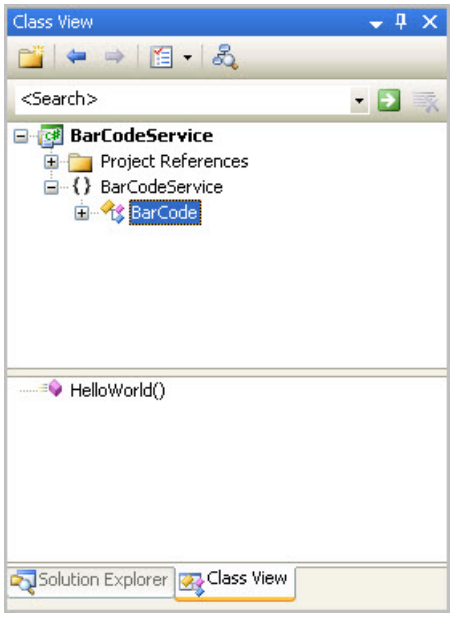


Figure 110: Class View; BarCode selected

The file view after renaming the asmx file:

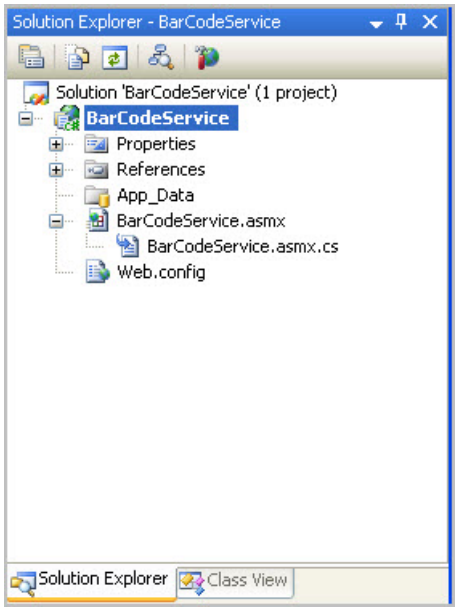
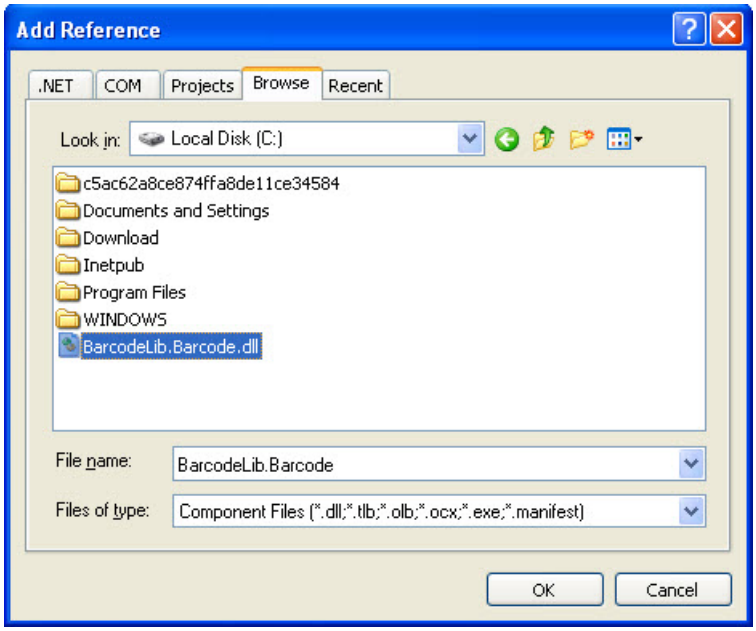


Figure 111: File View; BarCodeService selected

Step 3: Add the barcode library as a reference

Right-click on the solution explorer, select **Add Reference** and use the **Browse** panel to enter the location of the barcode library called **BarcodeLib.Barcode.dll**:



> **Note:** By default, the barcode library will be copied to the right place when deploying on the IIS web server.

Step 4: Add the buildImage method

Remove the default generated HelloWorld method, and create the buildImage method.

Add the three `using` instructions to import the barcode library, and to declare **buildImage** as a WebMethod. Use the **GetBarcodeBuilderType()** method to convert a string to a code as expected by the barcode library.

```
using BarcodeLib;
using BarcodeLib.Barcode;
using BarcodeLib.Barcode.Linear;

namespace BarCodeService
{
    /// <summary>
    /// Summary description for Service1
    /// </summary>
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [ToolboxItem(false)]
    // To allow this Web Service to be called from script, using ASP.NET
    // [System.Web.Script.Services.ScriptService]
    public class Barcode : System.Web.Services.WebService
    {

         [WebMethod]
         public byte[] buildImage(String type, String code)
         {
             try
             {
                 Linear barcode = new Linear();
                 barcode.Data = code;
                 barcode.Type = GetBarcodeBuilderType(type);
                 barcode.AddCheckSum = true;

                 // save barcode image into your system
                 barcode.ShowText = true;
                 byte[] ret = barcode.drawBarcodeAsBytes();
                 if (ret !- null) return ret;
                 else return null;
             }
             catch (Exception e)
             {
                 return null;
             }
        }
    }
}
```

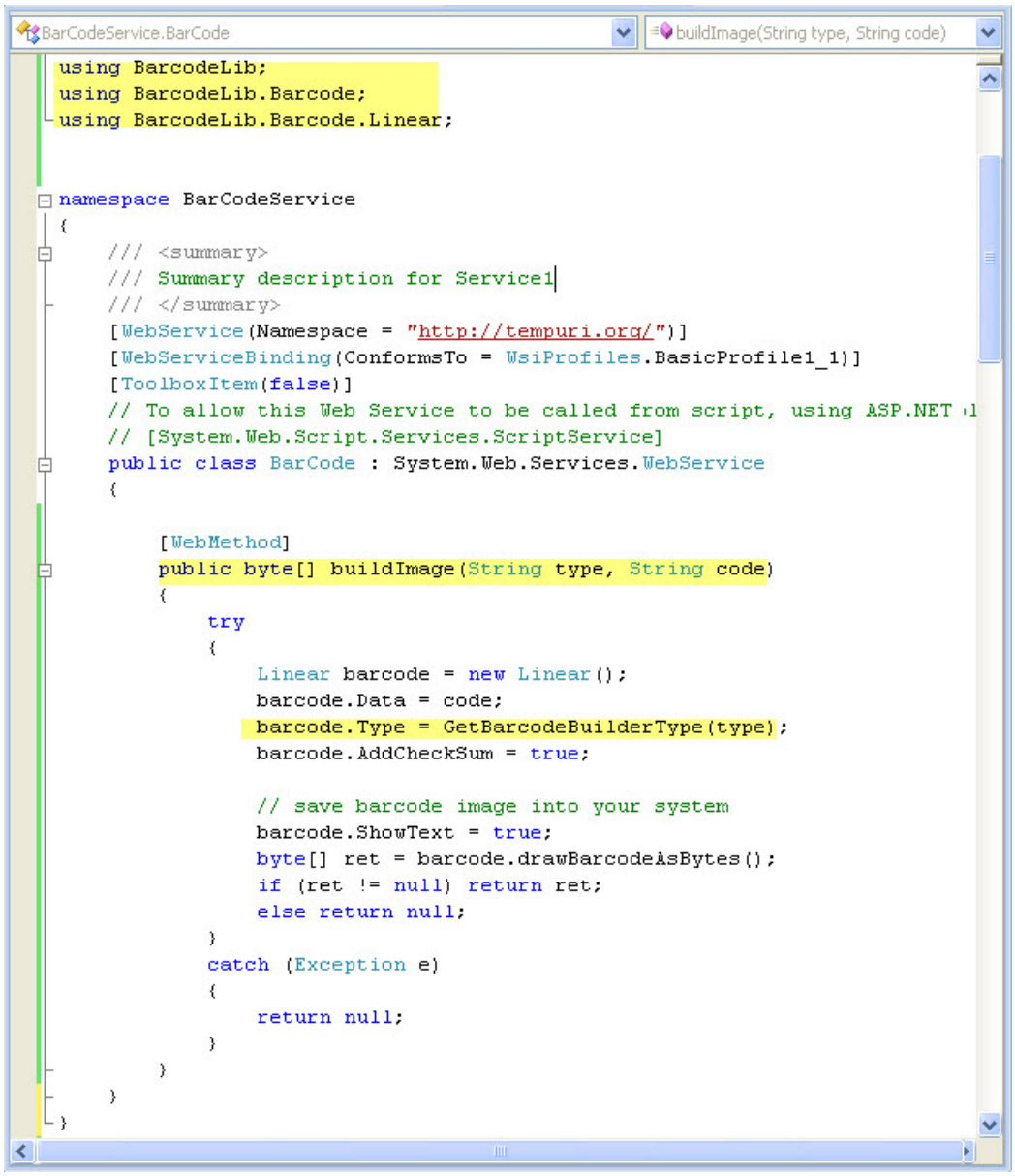


Figure 112: BarCodeService.BarCode

Step 5: Publish the service

Build the entire application, right-click on the solution, and select the publish operation. This will copy all necessary files to your IIS web server and make your application available at an URL, depending on where you deploy it on your IIS web server.

In our tutorial, the service will be located at the root of the server. In other words, it will be available at **http://localhost/BarCodeService.asmx** and the WSDL at URL **http://localhost/BarCodeService.asmx?WSDL**

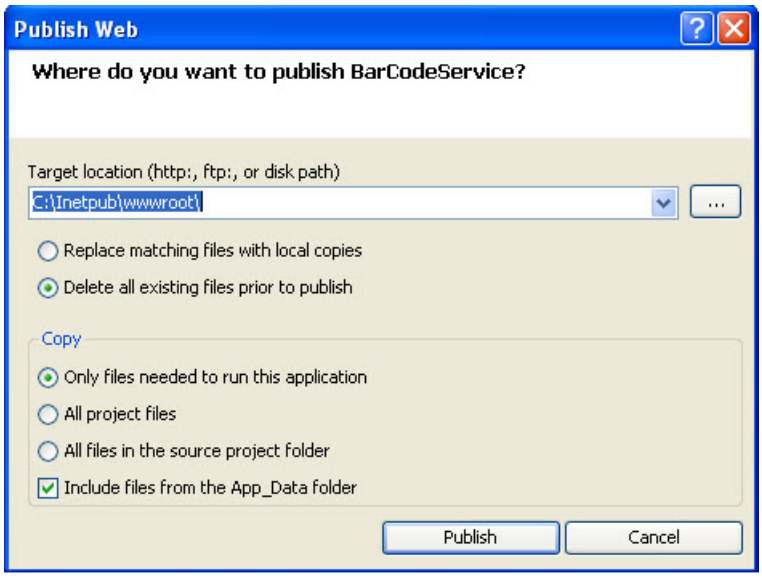


Figure 113: Publish Web dialog

Step 6: Generate .4gl stub to access the .NET library

Use the fglwsdl tool to generate the client stub to access the BarcodeService, as follows:

```
$ fglwsdl http://localhost/BarCodeService.asmx?WSDL
```

This will create two .4gl files, which must be compiled and linked into your BDL application in order to call the .NET barcode library functions. These files contain the BDL interface to access the .NET library where you will find the function **buildImage**, defined in BDL.

Step 7: Modify your BDL application

Modify your existing application, where you want to use the .NET library, by calling the BDL functions generated in the stub. Then compile your application and the previously generated stubs, and link everything together.

Your application is now ready to use the different features of your .NET library.

Example BDL program

This program calls the buildImage function of the Barcode .NET library.

```
GLOBALS "BarCode_BarCodeSoap.inc"
MAIN
  DEFINE wsstatus INTEGER

  IF num_args() != 3 THEN
    CALL ExitHelp()
  END IF

  LET buildImage.type = arg_val(1)
  LET buildImage.code = arg_val(2)
  LOCATE buildImageResponse.buildImageResult IN MEMORY

  LET wsstatus = buildImage_g()
  IF wsstatus <> 0 THEN
    DISPLAY "Error ("||wsError.code||") : ",wsError.description
  ELSE
    IF buildImageResponse.buildImageResult IS NULL THEN
      DISPLAY "Encoding failed"
    ELSE
```

```
      CALL buildImageResponse.buildImageResult.writeFile(arg_val(3))
    END IF
  END IF

  FREE buildImageResponse.buildImageResult

END MAIN

FUNCTION ExitHelp()
  DISPLAY arg_val(0)||" <type> <data> <filename>"
  DISPLAY "type : barcode type such as EAN8 or CODE128"
  DISPLAY "data : data to be encoded with a barcode [0-9A-D]"
  DISPLAY "filename : resulting image filename"
  DISPLAY "exemple : createImage EAN8 12358723A mybarcode.jpg"
  EXIT PROGRAM (-1)
END FUNCTION
```

Conclusion

It is quite easy to interact with a .NET library from Genero using .NET Visual Studio and the web services. Of course you also need an IIS web server installed on your Windows™ system. This means that you can, in the same Genero application, interact with .NET and Java™ libraries without any strong linkage between Genero and the third party libraries you want to use. This meets the SOA principles that provide more flexibility to your entire BDL application.

You can integrate any new library from another vendor, without the risk of conflicts between different libraries that could happen if you had to link everything together in C or Java™.

You can upgrade a third party library without recompiling the BDL application, which will still work.

You can use all these third party libraries in other BDL or other applications.

Compute a hash value from a BDL string
Overview

This document explains how to compute a hash value of a BDL string using the security.Digest API.

Signing a XML document is nothing more than computing a hash over a fragment of XML. If you set the string you need to hash in an XML node, and use the correct XPath expression, the security.Digest API will do it for you.

Some special characters are escaped in XML. If you use one of them, the computed hash value will be wrong because the result is actually computed over the escaped string. The special characters to be aware of are: **"**, **'**, **&**, **<** and **>**.

Sample code

Then uses the security.Digest API to compute a XML digital signature for the content of the root node, or in other words the string you wish to hash, using a XPath expression.

And finally, retrieves the hash value from the signature and returns it.

The computed hash value is encoded in Base64, so you may have additional conversion to do.

```
IMPORT SECURITY

MAIN

  DEFINE result STRING

  IF num_args() != 2 THEN
    DISPLAY "Usage: ComputeHash <string> <hashcode>"
    DISPLAY "  string: the string to digest"
    DISPLAY "  hashcode: SHA1, SHA512, SHA384, SHA256, SHA224, MD5"
  ELSE
```

```
    LET result = ComputeHash(arg_val(1), arg_val(2))
    IF result IS NOT NULL THEN
      DISPLAY "Hash value is: ",result
    ELSE
      DISPLAY "Error"
    END IF
  END IF

END MAIN

FUNCTION ComputeHash(toDigest, algo)

  DEFINE toDigest, algo, result STRING
  DEFINE dgst security.Digest

  TRY
    LET dgst = security.Digest.CreateDigest(algo)
    CALL dgst.AddStringData(toDigest)
    LET result = dgst.DoBase64Digest()
  CATCH
    DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
    EXIT PROGRAM(-1)
  END TRY

  RETURN result
END FUNCTION
```

Example of usage:

```
$ fglrun ComputeHash "Hello, world !!!" SHA1
```

```
$ Hash value is: Ck1VqNd45QIvq3AZd8XYQLvEhtA=
```

Example
Computing a hash value of a string.

Program example `ComputeHash.4gl` :

```
IMPORT SECURITY

MAIN

  DEFINE result STRING

  IF num_args() != 2 THEN
    DISPLAY "Usage: ComputeHash <string> <hashcode>"
    DISPLAY "  string: the string to digest"
    DISPLAY "  hashcode: SHA1, SHA512, SHA384, SHA256, SHA224, MD5"
  ELSE
    LET result = ComputeHash(arg_val(1), arg_val(2))
    IF result IS NOT NULL THEN
      DISPLAY "Hash value is :",result
    ELSE
      DISPLAY "Error"
    END IF
  END IF

END MAIN

FUNCTION ComputeHash(toDigest, algo)

  DEFINE toDigest, algo, result STRING
  DEFINE dgst security.Digest
```

```
  TRY
    LET dgst = security.Digest.CreateDigest(algo)
    CALL dgst.AddStringData(toDigest)
    LET result = dgst.DoBase64Digest()
  CATCH
    DISPLAY "ERROR : ", STATUS, " - ", SQLCA.SQLERRM
    EXIT PROGRAM(-1)
  END TRY

  RETURN result
END FUNCTION
```

Example execution:

```
$ fglrun ComputeHash "Hello World" SHA1
```

Hash value is :Ck1VqNd45QIvq3AZd8XYQLvEhtA=

Fix Genero 2.10 to 2.11 WSDL generation issue

These topics explain how to convert a WSDL generated from Genero 2.11 and later, to a WSDL as generated in Genero 2.10.

Overview

Since Genero 2.11, each BDL variable generates a associated named complexType in the WSDL and references it. Notice that this does not impact the web service at all, but some tools will then generate additional client stubs to follow the WSDL definition with the name of such complexType. This means that a client program written from a WSDL generated in 2.10 must be reviewed if it uses now a WSDL generated in 2.11 or later.

If you do not want to modify your application, you can use following program that will remove the named complexType and add the unnamed equivalent as child node of the parameter variable of all web service operations, so in other words, as if the WSDL would have been generated in 2.10.

WSDL conversion tool

This program reads a WSDL, looks for all named complexType used in all the web operation parameters and modifies them in order to have unamed complexType instead.

```
IMPORT XML
MAIN
  DEFINE
    doc xml.DomDocument,
    list, elist, tlist xml.DomNodeList,
    node, enode, nnode xml.DomNode,
    i, j, k, idx INTEGER,
    ename, tname STRING

  IF num_args() <> 1 THEN
    CALL display_help()
    RETURN 0
  END IF

  TRY
    LET doc = xml.DomDocument.Create()
    CALL doc.setFeature("whitespace-in-element-content",FALSE)
    CALL doc.load(arg_val(1))
    # get the list of input/output message
    # check if their names (x) are defined as elements with types (y)
    # if yes then
    # copy the complextype y definition to element name x
    # and remove the complexe type y definition
    # for example:
    # message
    # <wsdl:message name="is_OKIn">
```

```
    # <wsdl:part name="parameters" element="fjs:is_OKRequest" />
    # </wsdl:message>
    # <wsdl:message name="is_OKOut">
    # <wsdl:part name="parameters" element="fjs:is_OKResponse" />
    # </wsdl:message>
    # element
    # <xsd:element name="is_OKResponse"
    #    type="s1:is_OKResponse_is_OKResponse" />
    # type
    # <xsd:complexType name="is_OKRequest_is_OKRequest">
    LET list =
      doc.selectByXPath("//wsdl:part[@name='parameters']/@element",
      "wsdl","http://schemas.xmlsoap.org/wsdl/")
    IF list IS NULL THEN
      DISPLAY "Nothing to convert."
    END IF
    FOR i=1 TO list.getCount()
      LET node = list.getItem(i)
      LET ename = node.getNodeValue()
      LET idx = ename.getIndexOf(":",1)
      IF idx <> 0 THEN
      LET ename = ename.subString(idx+1,ename.getLength())
      END IF
      # get the element
      LET elist =
        doc.selectByXPath("//xsd:element[@name='" || ename || "']",
        "xsd","http://www.w3.org/2001/XMLSchema")
      IF elist IS NOT NULL THEN
        FOR j=1 TO elist.getCount()
          LET enode = elist.getItem(j)
          LET tname = enode.getAttribute("type")
          CALL enode.removeAttribute("type")
          LET idx = tname.getIndexOf(":",1)
          IF idx <> 0 THEN
          LET tname = tname.subString(idx+1,tname.getLength())
          END IF
          # get the type
          LET tlist =
            doc.selectByXPath("//xsd:complexType[@name='" || tname || "']",
            "xsd","http://www.w3.org/2001/XMLSchema")
          IF tlist IS NOT NULL THEN
            FOR k=1 TO tlist.getCount()
              LET node = tlist.getItem(k)
              LET nnode = node.clone(TRUE)
              CALL nnode.removeAttribute("name")
              CALL enode.appendChild(nnode)
            END FOR
          END IF
          FOR k=1 TO tlist.getCount()
            LET node = tlist.getItem(k)
            LET nnode = node.getParentNode()
            CALL nnode.removeChild(node)
          END FOR
        END FOR
      END IF
    END FOR
    CALL doc.setFeature("format-pretty-print",TRUE)
    CALL doc.save("result.wsdl")
    DISPLAY "Document is saved in result.wsdl"
  CATCH
    DISPLAY "ERROR[" || STATUS || "]"
    FOR i=1 TO doc.getErrorsCount()
      DISPLAY "[", i, "] ", doc.getErrorDescription(i)
    END FOR
```

```
  END TRY
END MAIN

FUNCTION display_help()
  DISPLAY "Usage: fglrun " || arg_val(0) || " wsdlfile"
END FUNCTION
```

Example of usage:

```
$ fglrun Convert Genero2_21.wsdl
```

```
$ Document is saved in result.wsdl
```

How to handle WS security

Genero Web Services does not entirely manage WS-Security. We provide XML APIs to help the development of Web Services with security.

Introduction

This topic describes how to handle WS Security using the demo wssecuritymessage. It is a sample that you can adapt to your needs. The demo will be enhanced to illustrate new features that will be introduced to fully support WS-Security.

The demo involves three clients exchanging secured messages. Those clients post and retrieve messages on a secured server. Each client is identified by a certificate and sign their messages.

We assume that you are familiar with security concepts described in topic " Encryption and Authentication Concepts".

The demo assumes that all the clients have sent their public keys to the other clients and to the server. Those keys are kept in each host's (server or clients) keystore. The certificates included in this package are provided for demonstration purposes only. As they are distributed with this package, anybody using this product can decrypt the messages exchanged. Do NOT use them in production.

Server side

We provide 3 handlers to handle WS Security:

- Method com.WebService.registerWSDLHandler() to modify the wsdl to add WS policy.
- Method com.WebService.registerInputRequestHandler() to handle WS Security in an incoming request
- Method com.WebService.registerOutputRequestHandler() to handle WS Security in an outgoing request

In this demo, a received message is processed:

1. Identify the sender and validate the sender (search in keystore)
2. Decrypt the symmetric key with the server private key
3. Decrypt the body
4. Check the signature with the sender public key
5. Store the message in the box (thanks to the "To" field, "subject" and "message")
6. Create the outgoing message
7. Sign the outgoing message
8. Encrypt the outgoing message with a generated symmetric key. This symmetric key is then encrypted with the client public key.

Client side

The client consists in sending a message and retrieving messages clients sent to it.

Before that, create the client stub from the wsdl:

- fglwsdl **-domHandler** myservice.wsdl

The client stub reference handlers:

- SecureMessageBox_HandleRequest
- SecureMessageBox_HandleResponse
- SecureMessageBox_HandleResponseFault

For more details about client SOAP handlers see Client stub and handlers.

What to do when a message is sent:

- Sign and encrypt the request for the server (WS-Security)
  - sign with client private key
  - encrypt with server public key
- Send key information in the request
  - key to identify the sender/client
  - key to identify the recipient/server
  - key used to encrypt the data (usually a symmetric key encrypted by the recipient public key)
- If the message has to be encrypted for the final recipient (XML-Security)
  - sign the message
  - encrypt the message

What to do to retrieve messages:

- Identify the sender and validate the sender (search in keystore)
- Identify the recipient (should be the server itself)
- Decrypt the request
- Check the signature
- Retrieve messages for the recipient

SOAP security standards

The policy documentation can be found here.

The demo policy is divided into sections (make sure that the naming are correct and that the structure is understandable):

- Security bindings on page 2502
- SOAP message security options on page 2505
- SignedParts on page 2505
- EncryptedParts on page 2505

WS Security section begins with:

```
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy ... />
```

It defines rules:

```
<wsp:ExactlyOne>
```

Only one assertion should be fulfilled.

```
<wsp:All>
```

All the assertions should be fulfilled.

Security bindings

There are 3 types of security binding:

- TransportBinding
- SymmetricBinding

- AsymmetricBinding

The current demo uses the Asymmetric binding.

Asymmetric Binding

This section is divided in sub sections:

- InitiatorToken
- RecipientToken
- AlgorithmSuite
- Layout
- Additional assertions

`AsymmetricBinding` is the root node for protection description.

```
<sp:AsymmetricBindingxmlns:sp=
"http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
```

InitiatorToken

`InitiatorToken` is the message sender (client)

For example:

```
<sp:InitiatorToken>
  <wsp:Policy>
    <sp:X509Token sp:IncludeToken="http://schemas.xmlsoap.org/ws/
     2005/07/securitypolicy/IncludeToken/AlwaysToRecipient">
      <wsp:Policy>
        <sp:RequireThumbprintReference />
        <sp:WssX509V1Token10 />
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:InitiatorToken>
```

> **Note:** The value for the sp:IncludeToken attribute is one contiguous string with no spaces. For this document, it is shown covering two lines.

The token is used for the message signature from initiator to recipient and encryption from recipient to initiator.

The initiator key is a X509 certificate that is always sent to the recipient.

`sp:IncludeToken` attribute indicates if the token must be included.

`IncludeToken/AlwayToRecipient` means each requests sent to the recipient must include the initiator token. But the token should not be included in messages from recipient to initiator.

The token must send its Thumbprint Reference.

The token must be of type X509 version 1 as defined in "X509 token profile 1.0".

What should be done in BDL is decribed in Client Side section.

To retrieve the thumbprint reference you can use the API function xml.CryptoX509.getThumbprintSHA1

To create the x509 certificate use an appropriate tool like openssl.

RecipientToken

`RecipientToken` is the message receiver (server)

```
<sp:RecipientToken>
  <wsp:Policy>
    <sp:X509Token sp:IncludeToken="http://schemas.xmlsoap.org/
     ws/2005/07/securitypolicy/IncludeToken/Never">
      <wsp:Policy>
        <sp:RequireThumbprintReference />
        <sp:WssX509V3Token10 />
      </wsp:Policy>
    </sp:X509Token>
  </wsp:Policy>
</sp:RecipientToken>
```

> **Note:** The value for the sp:IncludeToken attribute is one contiguous string with no spaces. For this document, it is shown covering two lines.

The token is used for encryption from initiator to recipient, and for the message signature from recipient to initiator.

The recipient key is a X509 certificate that is never sent to the initiator.

`sp:IncludeToken` attribute indicates if the token must be included.

`IncludeToken/Never` means the token should not be included in any requests between the initiator and the recipient.

Instead the recipient ThumbprintReference is sent.

The token must be of type X509 version 3 as defined in "X509 token profile 1.0"

What should be done in BDL is described in Server Side section. To retrieve the thumbprint reference you can use the API function xml.CryptoX509.getThumbprintSHA1. To create the appropriate certificate use an appropriate tool like openssl.

AlgorithmSuite

`AlgorithmSuite` tells which algorithm is used to encrypt the data.

```
<sp:AlgorithmSuite>
  <wsp:Policy>
    <sp:TripleDesRsa15 />
  </wsp:Policy>
</sp:AlgorithmSuite>
```

TripleDesRsa15 refers to key http://www.w3.org/2001/04/xmlenc#tripledes-cbc.

Layout

`Layout` describes the way information are added to the message header.

```
<sp:Layout>
  <wsp:Policy>
    <sp:Strict />
  </wsp:Policy>
</sp:Layout>
```

For example, with Strict layout, token that are included in the message must be declared before use. For more details on the rules to follow see the security policy specifications section 7.7.

Additional Assertations

PartsToSign

```
<sp:OnlySignEntireHeadersAndBody />
```

The assertion means if there is any signature on the header or the body it should be on the entire header and the entire body not on their child element.

SOAP message security options

```
<sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <sp:MustSupportRefKeyIdentifier />
  <sp:MustSupportRefIssuerSerial />
```

- MustSupportRefKeyIdentifier means that initiator and recipient are able to generate and process key identifier reference.
- MustSupportRefIssuerSerial means that initiator and recipient are able to generate and process issuer and token serial reference.

SignedParts

The SignedParts section tells which part of the message should be signed.

```
<sp:SignedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/
securitypolicy">
<sp:Body />
```

- Only the body needs to be signed

EncryptedParts

The section EncryptedParts tells which part of the message should be encrypted.

```
<sp:EncryptedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/
securitypolicy">
  <sp:Body />
```

- sp:Body indicates the body message needs to be encrypted

Encrypt the body using the algorithm referenced in assertion AlgorithmSuite:

- create an encryption key using TripleDesRsa15 algorithm (i.e. generate a TripleDES symmetric key and then encrypt it with a RSA1.5 public key) like example2 in crypto key chapter that uses AES256.
- encrypt the body with the created key

To find the exact syntax of security message read the specifications "Web Services Security: SOAP Message Security 1.0".

Useful links

- Security Policy specifications v1.2
- SOAP Message Security 1.0
- X.509 Token Profile 1.1

RESTful Web Services

While RESTful Web Services are supported, the RESTful Web Services documentation is not yet completed.

The Genero APIs for creating Web services can be found in the Library section of this manual. See The com package on page 2009 and The xml package on page 2103.

Deploy a Web Service

Web services server program deployment

Introduction

In a production environment, Genero Web Services becomes a part of a global application architecture handled by the **Genero Application Server (GAS)**. The GWS DVMs are managed by the GAS.

This architecture takes care of:

- Security issues
- Scalability
  - Load management
  - Balancing of the Web service requests amongst the available virtual machines
- Runtime monitoring

GAS configuration

For deployment, the GWS Server application must be added to the GAS configuration. See *Adding Applications* in the GAS manual.

The web services application can be added to the GAS in different ways:

- GWS Server application implementing a single Web Service.

  This application could be deployed on various physical machines. A Genero Web Services VMProxy (GWSProxy) is started on each machine where the GWS Server application is executed, to manage the requests for a service and manage the DVMs that handle the requests. A single VMProxy can communication with multiple GWS DVMs, and manage the load balancing.
- GWS Server application implementing multiple Web Services.

  The GWSproxy would manage the client requests, dispatching the request to the appropriate DVM and the appropriate web service.

  Note: A Web Service Server must be stateless; several instances of the same Service can be created to support load balancing.

The basic deployment strategy can be implemented in varying permutations, depending on your business needs and the volume of requests.

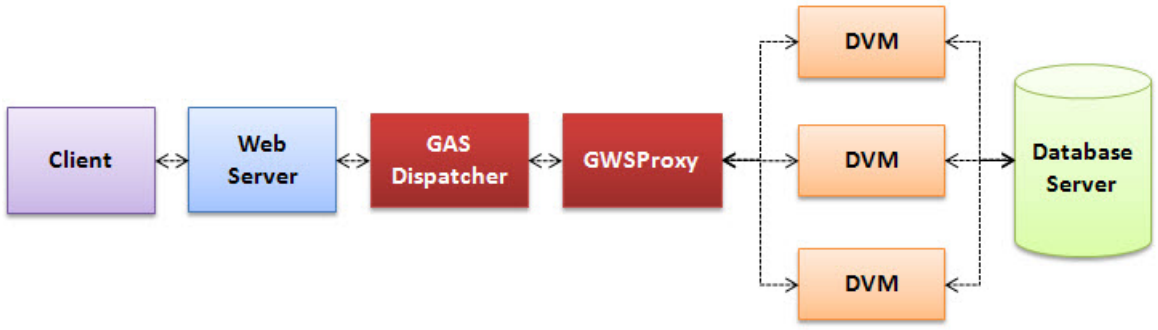


Figure 114: Deployment strategy

- Using the World Wide Web, a Web Service client requests WSDL information for a particular Web Service from the Web Server.
- The Web Service client uses this information to make a Web Service request from the Web Server.
- The Web server passes the request to the GAS dispatcher.

- The GAS dispatcher starts a GWSProxy, which will be in charge of the pool of DVMs that will serve the web service application.
- The GWSProxy will start the number of DVMs specified by the START element defined for the web service application.

For a more detailed explanation of the Services Pool for web services, refer to the *GAS Architecture* topic in the Genero Application Server manual.

Access the web services server from a client application

To reach the web service from the internet, client applications must use the following URL form:

```
http://host_name/ws/r/app_id
```

1. *host_name* defines the web server host name where the GAS is running.
2. *app_id* is the XCF file of the GAS web services application.

Configuring the apache web server for HTTPS

The first three steps are for the creation of all X.509 certificates.

- Step 1: Create the Root Certificate Authority
- Step 2: Create the server's certificate and private key
- Step 3: Create the server's certificate authority list

The next three steps are for server configuration.

- Step 4: Register the server as a Web Service in the GAS
- Step 5: Configure apache for HTTPS
- Step 6: Configure apache for HTTP basic authentication

Step 1: Create the root certificate authority

- Create the root certificate authority serial file:

```
$ echo 01 > MyCompanyCA.srl
```

- Create the Root Authority's Certificate Signing Request and private key:

```
$ openssl req -new -out MyCompanyCA.csr -keyout MyCompanyCA.pem
```

- Create the Root Certificate Authority for a period of validity of 2 years:

```
$ openssl x509 -trustout -in MyCompanyCA.csr -out MyCompanyCA.crt
 -req -signkey MyCompanyCA.pem -days 730
```

Note: The private key file (**MyCompanyCA.pem**) of a RootCertificate Authority must be handled with care. This file is responsible for the validity of all other certificates it has signed. As a result, it must not be accessible by other users.

Step 2: Create the server's certificate and private key

- Create the server's serial file:

```
$ echo 01 > MyServer.srl
```

- Create the server's Certificate Signing Request and private key:

```
$ openssl req -new -out MyServer.csr
```

Note: By default, **openssl** outputs the private key in the **privkey.pem** file.

- Remove the password from the private key:

```
$ openssl rsa -in privkey.pem -out MyServer.pem
```

Note: The key is also renamed in **MyServer.pem**.

- Create the server's Certificate trusted by the Root Certificate Authority:

```
$ openssl x509 -in MyServer.csr -out MyServer.crt
 -req -signkey MyServer.pem -CA MyCompanyCA.crt -CAkey MyCompanyCA.pem
```

Note: The purpose of the server's Certificate is to identify the server to any client that connects to it. Therefore, **the subject of that server's certificate must match the hostname of the server as it is known on the network**; otherwise the client will be suspicious about the server's identity and stop the communication. For instance, if the URL of the server is *https:///www.MyServer.com/cgi-bin/fglccgi.exe/ws/r/MyWebService*, the subject must be *www.MyServer.com*.

Step 3: Create the server's certificate authority list

- Create the server's Certificate Authority List:

```
$ openssl x509 -in MyCompanyCA.crt -text >> ServerCAList.pem
```

Note: As the server trusts only the Root Certificate Authority, the list contains only that one certificate authority; all other certificates that were trusted by the Root Certificate Authority will also be considered as trusted by the server.

Step 4: Register the server as a web service in the GAS

As the Web Server is in charge of the complete HTTPS protocol with all the clients, there is no additional GAS configuration needed to add security. Simply register the BDL server to the list of Web Services of the GAS. For more information, refer to the *Genero Application Server User Guide*.

For more details, see Web services server program deployment on page 2506.

Step 5: Configure apache for HTTPS

You must configure Apache to support HTTPS by adding the required modules. Please refer to the Apache Web server documentation for more information.

- For the Apache 1.3 manual, go to http://httpd.apache.org/docs/1.3.
- For the Apache 2.0 manual, go to http://httpd.apache.org/docs/2.0/.

Once the Apache Web server supports HTTPS, you must change or add the following directives to the apache configuration file:

- Set the Apache Web server Certificate Authority List directive created in Step 4 :

  SSLCACertificateFile D:/Apache-Server/conf/ssl/ServerCAList.pem
- Set the Apache Web server Certificate and associated private key directives created in Step 2 :

  SSLCertificateFile D:/Apache-Server/conf/ssl/MyServer.crt

  SSLCertificateKeyFile D:/Apache-Server/conf/ssl/MyServer.pem
- Require the Apache Web server to verify the validity of all client certificates:

  SSLVerifyClient require

  Note: The Apache Web server must be started on a machine where the host is the same asthe one defined in the subject of the server's certificate (*www.MyServer.com* in our case).

Step 6 : Configure apache for HTTP basic authentication

You must configure Apache to support HTTP basic authentication by adding the required modules.

Please refer to the Apache Web server documentation for more information.

- For the Apache 1.3 manual, go to http://httpd.apache.org/docs/1.3.
- For the Apache 2.0 manual, go to http://httpd.apache.org/docs/2.0/.

Once the Apache Web server supports HTTP basic authentication, you must:

1. Add an user to the Apache Web server basic authentication file with the same login and password as defined for the client.

   Apache provides the tool **htpasswd** that you can use to create the file and add the user. To add the user **mylogin** with the password **mypassword** to a new file called **myusers**:

   ```
   $ htpasswd -c myusers mylogin mypassword
   ```

   Note: To add additional users, remove the option '`-c`'.

2. Add an Apache Web server location directive that enables you to group several directives for one URL. (In our case, the URL is /cgi-bin/fglccgi.exe/ws/r/MyWebService).

   The following example (based on Apache 2.0) defines the HTTP authentication type (Basic), with a user file (user-basic) containing the login and password of those who are allowed to access the service.

   ```
     <Location /cgi-bin/fglccgi.exe/ws/r/MyWebService>
        AllowOverride None
        Order allow,deny
        Allow from all
        #
        # Basic HTTP authenticate configuration
        #
        AuthName "Top secret"
        AuthType Basic
        AuthUserFile "D:/Apache-Server/conf/authenticate/myusers"
        Require valid-user # Means any user in the password file
   </Location>
   ```

   For more information about Apache Web server directives, refer to the Apache Web Server manual.

Reference

These topics are the reference guides for Genero Web Services.

- Web services configuration on page 2509
- Attributes to customize XML serialization on page 2517
- Error handling in GWS calls (STATUS) on page 2546
- Interruption handling in GWS calls (INT_FLAG) on page 2546
- Server API functions - version 1.3 only on page 2546
- Configuration API functions - version 1.3 only on page 2552
- Using fglwsdl to generate code from WSDL or XSD schemas on page 2555

Web services configuration

The Genero Web Services secured communication and the support of XML-Security is based on the OpenSSL engine. It allows a BDL Web Services client, or a BDL application using the `com` or `xml` API, to

communicate with any secured server over HTTP or HTTPS, and to handle encrypted and/or signed XML document in BDL coming from any other application.

The configuration is defined from entries in the `fglprofile` file. When using BDL Web Services on server side, it is the Web Server that is in charge of the BDL Web Services server security, not the BDL server application itself. You must refer to your Web Server manual to secure the server part of the Web Services.

> **Note:** This is useful for deployment purposes, as no additional code modification is necessary, even if the location of the different servers changes, or if different cryptography keys or X509 certificates are necessary for a same application but intended to several customers with their own needs.

FGLPROFILE entries

The fglprofile entries relating to Genero Web Services are divided between five categories: security, basic or digest HTTP authentication, proxy configuration, server configuration, and XML cryptography.

> **Important:** Web Services FGLPROFILE configuration options are not supported on GMI mobile devices.

- HTTPS and password encryption on page 2510
- Basic or digest HTTP authentication on page 2512
- Proxy configuration on page 2512
- Server configuration on page 2513
- XML configuration on page 2514

HTTPS and password encryption

The following table lists the FGLPROFILE entries specifying the security certificates and algorithms the Web Services client uses for HTTPS and password encryption. These entries specify how an application using the low-level com or xml APIs performs secured communications.

Table 556: Security Configuration FGLPROFILE entries

Entry	Description
`security.global.script`	Filename of a script executed each time a password of a private key is required by the client. The security script accepts one argument corresponding to the filename of the private key for which the password is required, and must return the correct password or the client stops. For script examples, see Windows™ Password Script Example or UNIX™ Password Script Example. This entry cannot be used if security.global.agent is set.
`security.global.agent`	Port number where the fglpass agent is waiting for requests. It returns the password that grants access to a private key when needed by a BDL application. The DVM and the fglpass agent perform authentication and exchange encrypted data over the local host network only. Refer to Using the password agent for details. This entry cannot be used if security.global.script is set.
`security.global.protocol`	The SSL protocol to use for secured communications. Possible values are: • TLSv1.2 • TLSv1.1

Entry	Description
	• TLSv1 (version 1.0) • SSLv3 • SSLv23 (The default, enabling all supported protocols)
`security.global.ca`	Filename of the Certificate Authority list, with the concatenated PEM-encoded third party X.509 certificates considered as trusted, and in order of preference.
`security.global.windowsca`	If set to `true`, build the Certificate Authority list from the Certificate Authorities stored in the Windows™ key store. This entry is only valid on Windows™ systems where *security.global.ca* is not set.
`security.global.cipher`	The list of encryption, digest, and key exchange algorithms the client is allowed to use during a secured communication. If this entry is omitted, all algorithms are supported. For more details about cipher, refer to www.openssl.org.
`security.global.certificate`	Filename of the PEM-encoded client X.509 certificate to be used for any secured connection if not redefined in a specific server configuration.
`security.global.privatekey`	Filename of the PEM-encoded private key associated to the above X509 certificate and to be used for any secured connection if not redefined in a specific server configuration.
`security.global.keysubject`	The subject string of a X.509 certificate and its associated private key registered in the Windows™ key store to be used for any secured connection if not redefined in a specific server configuration. This entry is valid only on Windows™ systems.
`security.`*`ident`*`.certificate`	Filename of the PEM-encoded client X.509 certificate.
`security.`*`ident`*`.privatekey`	Filename of the PEM-encoded private key associated to the above X509 certificate.
`security.`*`ident`*`.keysubject`	The subject string of a X.509 certificate and its associated private key registered in the Windows™ key store. This entry is valid only on Windows™ systems.

Note:

1. The **ident** keyword must be replaced with your own identifier, and all necessary entries must be set. See FGLPROFILE setting.
2. If an entry is defined more that once, only the last occurrence is taken into account.

Basic or digest HTTP authentication

The following table lists the FGLPROFILE entries that specify the login and password to use in the case of HTTP authentication to a server or a proxy. The entries also specify the login and password to use in an application using the low-level com or xml API.

Table 557: HTTP basic or digest Authentication FGLPROFILE entries

Entry	Description
`authenticate.`*`ident`*`.login`	The login identifying the client to a server during HTTP Authentication.
`authenticate.`*`ident`*`.password`	The password validating the login of a client to a server during HTTP Authentication. As passwords should never be in clear text, it is recommended that you encrypt them with the **fglpass** tool. For more information, see FGLPROFILE password encryption.
`authenticate.`*`ident`*`.realm`	The string identifying the server to the client during HTTP Authentication. If the string does not match the server's string, authentication fails. This parameter is optional, but it is recommended that you check the server identity, especially if the server's location is suspicious.
`authenticate.`*`ident`*`.scheme`	One of the following strings representing the different HTTP Authentication mechanisms. • **Anonymous** (default value) - The client does not know anything about the server, and performs a first request to retrieve the server authentication mechanism. It then uses the login and password to authenticate to the server using the Basic or Digest mechanism, depending on the server returned value. • **Basic** - The client authenticates itself to the server at first request, by sending the login and the password using the Basic authentication mechanism. • **Digest** - The client performs a first request without any login and password, to retrieve the server information before authenticating itself to the server in a second request using the Digest mechanism.

Note:

1. The **ident** keyword must be replaced with your own identifier, and all necessary entries must be set. See FGLPROFILE setting.
2. If an entry is defined more that once, only the last occurrence is taken into account.

Proxy configuration

The following table lists the FGLPROFILE entries that specify how the Web Services client communicates with a proxy. The entries specify the way an application using the low-level com or xml API communicates with a proxy.

Table 558: Proxy Configuration FGLPROFILE entries

Entry	Description
`proxy.http.location`	Location of the HTTP proxy defined as **host:port** or **ip:port**. If the port is omitted, the port 80 is used.
`proxy.http.list`	The list of beginning host names, separated with semicolons, for which the Web Services client does not go via the HTTP proxy.
`proxy.http.authenticate`	The HTTP Authenticate identifier the Web Services client uses to authenticate itself to the HTTP proxy.
`proxy.https.location`	Location of the HTTPS proxy defined as **host:port** or **ip:port**. If the port is omitted, the port 443 is used
`proxy.https.list`	The list of beginning host names, separated with semicolons, for which the Web Services client does not go via this HTTPS proxy.
`proxy.https.authenticate`	The HTTP Authenticate identifier the Web Services client uses to authenticate itself to the HTTPS proxy.

Note: If an entry is defined more that once, only the last occurrence is taken into account.

IPv6 configuration

The following table lists the FGLPROFILE entries that specify how the Web Services client uses the IPv6 network protocol.

Table 559: IPv4 and IPv6 FGLPROFILE entries

Entry	Description
`ip.global.version`	Defines the IP version to be used. Possible values are "4" (IPv4) or "6" (IPv6). By default, when this entry is not defined, the WS library will try to use IPv6 and fallback to IPv4, according to the operating system.
`ip.global.v6.interface.name` **Important:** This entry is not supported on Microsoft™ Windows™ platforms.	Defines the name of the network interface to be used for IPv6 link-local addresses. For example, this entry can get values such as "eth0", "en0", "ethernet_5".
`ip.global.v6.interface.id`	Defines the id of the network interface to be used for IPv6 link-local addresses. For example, this entry can get values such as "1", "2", "11".

Note: If an entry is defined more that once, only the last occurrence is taken into account.

Server configuration

The following table lists the FGLPROFILE entries that specify the correct way a Web Services client connects to an end point (usually a server). Notice that the entries specify also the way an application using the low-level com or xml API connects to an end point.

Table 560: Server Configuration FGLPROFILE entries

Entry	Description
`ws.`*`ident`*`.url`	The endpoint URL of the server. By using a wildcard in the URL, you can create a URL base that applies to multiple server applications. URLs that have the same URL base can share server configuration (such as authentication and HTTPS). See Wildcards in the URL base on page 2517.
`ws.`*`ident`*`.cipher`	The list of encryption, digest and key exchange algorithms the client is allowed to use during a secured communication to that server. It overwrites the global definition.
`ws.`*`ident`*`.verifyserver`	If set to `true`, the client performs a strict server identity validation. If not fulfilled, it stops the communication; otherwise no server identity verification is performed. The default value is `true`.
`ws.`*`ident`*`.security`	The security identifier the client uses to perform an HTTPS communication to the server.
`ws.`*`ident`*`.authenticate`	The HTTP authenticate identifier the client uses to authenticate itself to the server.

Note:

1. The **ident** keyword must be replaced with your own identifier. All necessary entries, depending on the remote server's configuration, must be set. See FGLPROFILE setting.
2. You can use the unique identifier in the .4gl code instead of the server URL, with the **alias://** prefix. For example, **alias://ident**.
3. If an entry is defined more that once, only the last occurrence is taken into account.

XML configuration

The following table lists the FGLPROFILE entries that control XML to Genero values conversion, and XML cryptography key or certificate mapping.

Table 561: XML configuration FGLPROFILE entries

Entry	Description
`xml.keystore.calist`	The list of PEM-encoded third party X.509 certificates, separated with semicolons, of the Certificate Authority considered as trusted, in order of preference.
`xml.keystore.x509list`	The list of PEM-encoded third party X.509 certificates, separated with semicolons, to be used to find out the correct X.509 certificate when

Entry	Description
	getting an incomplete one in a XML signature or an encrypted XML document.
`xml.`*`ident`*`.key`	The filename of a cryptography key. For instance *RSA.pem*, *DSA.der* or *HMAC.bin*.
`xml.`*`ident`*`.x509`	The filename of a cryptography x509 certificate. For instance *Cert.crt*.
`xml.serializer.supportEmptyStrings`	Controls empty string XML nodes conversion to Genero `STRING` values. The default is `false`, empty XML tags are converted to `NULL`. If set to `true`, an empty XML tag is converted to an empty `STRING` value. As result, in Genero, the `LENGTH()` function will return zero and the `IS NULL` comparison operator will evaluate to `FALSE`. Note that this entry only works for the `STRING` data type, and if the tag is not present, the `STRING` is set to `NULL`.
`xml.signature.prefix = { "`*`prefix`*`" | "<none>" }`	Defines the prefix for an XML Signature. Use `"<none>"` to specify no prefix. By default, the XML Signature prefix is `"dsig"`.
`xml.encryption.prefix = { "`*`prefix`*`" | "<none>" }`	Defines the prefix for an XML Encrypted data. Use `"<none>"` to specify no prefix. By default, the XML Encrypted data prefix is `"xenc"`.

Note:

1. The **ident** keyword must be replaced with your own identifier. See FGLPROFILE sample 2.
2. You can use the unique identifier in the .4gl code instead of the filename.
3. If an entry is defined more that once, only the last occurrence is taken into account.

Examples
Windows™ password script example

```
@echo off
REM -- Windows password script
IF "%1" == "Cert/MyPrivateKeyA.pem" GOTO KeyA
IF "%1" == "Cert/MyPrivateKeyB.pem" GOTO KeyB
GOTO end
:KeyA
ECHO PasswordA
GOTO end
:KeyB
ECHO PasswordB
GOTO end
:end
GOTO :EOF
```

UNIX™ password script example

```
# UNIX password script
if [ "$1" == "Cert/MyPrivateKeyA.pem" ]
 then
  echo PasswordA
fi
if [ "$1" == "Cert/MyPrivateKeyB.pem" ]
 then
  echo PasswordB
fi
```

FGLPROFILE sample

The following is an FGLPROFILE sample, configured for a connection to a HTTPS server via a proxy, and with HTTP and Proxy Authentication.

```
# Security configuration
security.global.script   =  "Cert/password.sh"
security.global.ca       =  "Cert/CAList.pem"
security.global.cipher   =  "HIGH" # Use only HIGH encryption ciphers
security.mykey.certificate =  "Cert/MyCertificateA.crt"
security.mykey.privatekey  =  "Cert/MyPrivateKeyA.pem"

# Proxy HTTP Authentication
authenticate.proxyauth.login    =  "myapplication"
authenticate.proxyauth.password =  "mypswd"
authenticate.proxyauth.scheme   =  "Basic"

# HTTPS Proxy configuration
proxy.https.location     =  "10.0.0.170"
proxy.https.list         =  "www.mycompany.com;www.mycompany.com"
proxy.https.authenticate =  "proxyauth"

# Server HTTP Authentication
authenticate.serverauth.login    =  "mylogin"
authenticate.serverauth.password =  "password"

# Server configuration
ws.myserver.url =  "https://www.MyMachine.com/cgi-bin/fglccgi.exe/ws/r/
MyWebService"
ws.myserver.authenticate =  "serverauth"
ws.myserver.security     =  "mykey"
```

FGLPROFILE sample 2

The following is an FGLPROFILE sample, configured for XML cryptography and using the fglpass agent to get the private key passwords.

```
# Security configuration
security.global.agent     = "4444"

# Crypto configuration
xml.keystore.calist     = "RSARootCertificate.crt;DSARootCertificate.crt"
xml.keystore.x509list   = "RSA1024Certificate.crt;DSA1024Certificate.crt"
xml.id1.x509          = "RSA1024Certificate.crt"
xml.id2.x509          = "DSA1024Certificate.crt"
xml.id3.key           = "RSA1024Key.pem"
xml.id4.key           = "DSA1024Key.der"
xml.id5.key           = "HMAC.bin"
```

Wildcards in the URL base

By using a wildcard in the URL, you can create a URL base that applies to multiple server applications. URLs that have the same URL base can share server configuration (such as authentication and HTTPS).

To create a URL base, add a wildcard (`/*`) to the end of a URL in the fglprofile entry. A server application that starts with this URL (and that is not explicitly defined elsewhere) shares the configuration with other applications that also start with the same base URL. If an application has its own server configuration explicitly defined, it uses its specific entries instead of those defined by the wildcard configuration.

Consider this excerpt from a hypothetical `fglprofile`:

```
authenticate.auth.login     = "xxx"
authenticate.auth.password  = "yyy"
authenticate.auth.scheme    = "Basic"

security.sec.certificate    = "client.crt"
security.sec.privatekey     = "client.pem"

ws.myapp.url                = "http://mycompany.com/sample/*"
ws.myapp.authenticate       = "auth"
ws.myapp.security           = "sec"

ws.thirdapp.url             = "http://mycompany.com/sample/application3"
ws.thirdapp.authenticate    = "auth3"

authenticate.auth3.login    = "aaa"
authenticate.auth3.password = "bbb"
authenticate.auth3.scheme   = "Basic"
```

Based on this example:

- Requests to "`http://mycompany.com/sample/application1`" and "`http://mycompany.com/sample/demos/shoppingcart`" use the same authentication and HTTPS configuration.
- A request to "`http://mycompany.com/sample/application3`" uses its specific authentication "`auth3`". No security configuration is defined for this URL, nor does it fall back on the shared security configuration defined for the base URL.

  Note: This applies to:

  - com.HTTPRequest.Create on page 2057`com.HTTPRequest.Create()`
  - xml.DomDocument.load on page 2123`xml.DomDocument.load()`
  - xml.DomDocument.save on page 2126`xml.DomDocument.save()`
  - xml.StaxReader methods
  - xml.StaxWriter methods
  - com.TCPRequest.Create on page 2087

Attributes to customize XML serialization

See The Serializer class on page 2202 for information on setting serialization options when mapping BDL and XML data.

BDL to/from XML type mappings

Starting with Genero 2.0, you can add optional attributes to the definition of program variables to be used for XML serialization. These attributes can be used to map a BDL data type used in the input or output message of a Genero Web Service application to a specific XML data type, rather than using the default.

For example, if an XML Schema boolean data type is required for an application, and the corresponding BDL type is a SMALLINT, you can use an attribute to map the BDL SMALLINT variable to the XML boolean.

The following example uses the XSDBoolean attribute to map a BDL SMALLINT variable to an XML Schema boolean type, and assigns an uppercase name as the XMLName attribute:

```
GLOBALS
DEFINE invoice_out RECORD
  ok SMALLINT ATTRIBUTES(XSDBoolean,XMLName="OK")
END RECORD

END GLOBALS
```

If you assign your own XMLName attributes, be sure to respect the conventions when using the RPC Service Style.

See the Tutorial: Writing a GWS Server application for additional information about input and output messages.

Default BDL/XML mapping

By default, Genero Web Services maps BDL variables in the input or output messages of a WS application to their corresponding XML data types, enabling values to be passed between applications and Web Services. The XML data types conform to the standard XML Schema Definition (XSD):

Table 562: Default XML Mapping

data type of BDL variable	Default XML data type
BYTE	xsd:base64binary
CHAR	xsd:string
DATE	xds.date
DATETIME YEAR TO FRACTION(1-5)	xsd:dateTime
DATETIME YEAR TO SECOND	xsd:dateTime
DATETIME YEAR TO HOUR	xsd:dateTime
DATETIME YEAR TO MINUTE	xsd:dateTime
DATETIME YEAR TO YEAR	xsd:gYear
DATETIME YEAR TO MONTH	xsd:gYearMonth
DATETIME YEAR TO DAY	xsd:date
DATETIME MONTH TO MONTH	xsd:gMonth
DATETIME MONTH TO DAY	xsd:gMonthDay
DATETIME DAY TO DAY	xsd:gDay
DATETIME HOUR TO HOUR	xsd:time
DATETIME HOUR TO MINUTE	xsd:time
DATETIME HOUR TO SECOND	xsd:time
DATETIME HOUR TO FRACTION(1-5)	xsd:time
DECIMAL	xsd:decimal
FLOAT	xsd:double
INTEGER	xsd:int

data type of BDL variable	Default XML data type
INTERVAL	xsd:duration
SMALLFLOAT	xsd:float
SMALLINT	xsd:short
STRING	xsd:string
TEXT	xsd:string
VARCHAR	xsd:string
TINYINT	xsd:byte
BIGINT	xsd:long
BOOLEAN	xsd:boolean

In addition, the Web Service Style that you use determines what default XMLName attributes are assigned to variables.

Type mapping attributes

The attributes listed in this table cannot have values.

Table 563: Mapping between simple BDL and XML data types

Attribute	Definition
XSDAnySimpleType	Map BDL STRING or VARCHAR to XML Schema simpleType.
XSDAnyType	Map BDL STRING or VARCHAR to XML Schema anyType.
XSDAnyURI	Map BDL STRING or VARCHAR to XML Schema anyURI.
XSDBase64binary	Map BDL BYTE to the XML Schema base64binary.
XSDBoolean	Map BDL BOOLEAN, SMALLINT or INTEGER to XML Schema boolean.
XSDByte	Map BDL TINYINT, SMALLINT or BIGINT to XML Schema byte.
XSDDate	Map BDL DATE or DATETIME to XML Schema date.
XSDDateTime	Map BDL DATETIME to XML Schema dateTime.
XSDDecimal	Map BDL DECIMAL to XML Schema decimal.
XSDDouble	Map BDL FLOAT to XML Schema double.
XSDDuration	Map BDL INTERVAL to XML Schema duration.
XSDEntities	Map BDL STRING or VARCHAR to XML Schema entities.
XSDEntity	Map BDL STRING or VARCHAR to XML Schema entity.
XSDFloat	Map BDL SMALLFLOAT to XML Schema float.

Attribute	Definition
XSDGday	Map BDL DATETIME to XML Schema gDay.
XSDGMonth	Map BDL DATETIME to XML Schema gMonth.
XSDGMonthDay	Map BDL DATETIME to XML Schema gMonthDay.
XSDGYear	Map BDL DATETIME to XML Schema gYear.
XSDGYearMonth	Map BDL DATETIME to XML Schema gYearMonth.
XSDHexBinary	Map BDL BYTE to XML Schema hexBinary.
XSDID	Map BDL STRING or VARCHAR to XML Schema id.
XSDIDREF	Map BDL STRING or VARCHAR to XML Schema idRef.
XSDIDREFS	Map BDL STRING or VARCHAR to XML Schema idRefs.
XSDInt	Map BDL INTEGER or BIGINT to XML Schema int.
XSDInteger	Map BDL DECIMAL to XML Schema integer.
XSDLanguage	Map BDL STRING or VARCHAR to XML Schema language.
XSDLong	Map BDL BIGINT or DECIMAL to XML Schema long.
XSDNCName	Map BDL STRING or VARCHAR to XML Schema NCName.
XSDName	Map BDL STRING or VARCHAR to XML Schema Name.
XSDNegativeInteger	Map BDL DECIMAL to XML Schema negativeInteger.
XSDNMTOKEN	Map BDL STRING or VARCHAR to XML Schema NMToken.
XSDNMTOKENS	Map BDL STRING or VARCHAR to XML Schema NMTokens.
XSDNonNegativeInteger	Map BDL DECIMAL to XML Schema nonNegativeInteger.
XSDNonPositiveInteger	Map BDL DECIMAL to XML Schema nonPositiveInteger.
XSDNormalizedString	Map BDL STRING or VARCHAR to XML Schema normalizedString.
XSDNotation	Not supported.
XSDPositiveInteger	Map BDL DECIMAL to XML Schema positiveInteger.
XSDQName	Map BDL STRING or VARCHAR to XML Schema QName.

Attribute	Definition
XSDShort	Map BDL SMALLINT or BIGINT to XML Schema short.
XSDString	Map BDL STRING , Char, Text or VarChar to XML Schema string.
XSDTime	Map BDL DATETIME to XML Schema time.
XSDToken	Map BDL STRING or VARCHAR to XML Schema token.
XSDUnsignedByte	Map BDL SMALLINT or BIGINT to XML Schema unsignedByte.
XSDUnsignedInt	Map BDL BIGINT or DECIMAL to XML Schema unsignedInt.
XSDUnsignedLong	Map BDL DECIMAL to XML Schema unsignedLong.
XSDUnsignedShort	Map BDL INTEGER or BIGINT to XML Schema unsignedShort.

XSDAnySimpleType

Map BDL STRING or VARCHAR to XML Schema anySimpleType.

XSDAnyType

Map BDL STRING or VARCHAR to XML Schema anyType.

XSDAnyURI

Map BDL STRING or VARCHAR to XML Schema anyURI.

XSDBase64binary

Map BDL BYTE to XML Schema base64binary.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 BYTE ATTRIBUTES(XSDBase64binary,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>F0FFC8D27FF001547FC219E1FFF009F0FFC8D27FF001547D</Val>
</Root>
```

XSDBoolean

Map BDL BOOLEAN, SMALLINT or INTEGER to XML Schema boolean.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
    val1 INTEGER ATTRIBUTES(XSDBoolean,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>true</Val>
```

```
</Root>
```

XSDByte

Map BDL TINYINT, SMALLINT or BIGINT to XML Schema byte.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 SMALLINT ATTRIBUTES(XSDByte,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>-126</Val>
</Root>
```

XSDDate

Map BDL DATE or DATETIME to XML Schema date.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATE ATTRIBUTES(XSDDate,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>2006-06-29+01:00</Val>
</Root>
```

XSDDateTime

Map BDL DATETIME to XML Schema dateTime.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME ATTRIBUTES(XSDDateTime,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>2006-06-29T09:35:26.13584+01:00</Val>
</Root>
```

XSDDecimal

Map BDL DECIMAL to XML Schema decimal.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(5,3) ATTRIBUTES(XSDDecimal,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>12.345</Val>
```

```
</Root>
```

XSDDouble

Map BDL FLOAT to XML Schema double.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 FLOAT ATTRIBUTES(XSDDouble,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>12.78e-2</Val>
</Root>
```

XSDDuration

Map BDL INTERVAL to XML Schema duration.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTERVAL DAY TO SECOND
 ATTRIBUTES(XSDDuration,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>P3DT10H30M45S</Val>
</Root>
```

XSDEntities

Map BDL STRING or VARCHAR to XML Schema ENTITIES.

XSDEntity

Map BDL STRING or VARCHAR to XML Schema ENTITY.

XSDFloat

Map BDL SMALLFLOAT to XML Schema float.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 SMALLFLOAT ATTRIBUTES(XSDFloat,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>126.435</Val>
</Root>
```

XSDGDay

Map BDL DATETIME to XML Schema gDay.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME DAY TO DAY ATTRIBUTES(XSDGDay,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>---25</Val>
</Root>
```

XSDGMonth

Map BDL DATETIME to XML Schema gMonth.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME MONTH TO MONTH
 ATTRIBUTES(XSDGMonth,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>--12</Val>
</Root>
```

XSDGMonthDay

Map BDL DATETIME to XML Schema gMonthDay.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME MONTH TO DAY
 ATTRIBUTES(XSDGMonthDay,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>--12-31</Val>
</Root>
```

XSDGYear

Map BDL DATETIME to XML Schema gYear.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME YEAR TO YEAR ATTRIBUTES(XSDGYear,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>2006</Val>
</Root>
```

XSDGYearMonth

Map BDL DATETIME to XML Schema gYearMonth.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME YEAR TO MONTH
 ATTRIBUTES(XSDGYearMonth,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>2006-06</Val>
</Root>
```

XSDHexBinary

Map BDL BYTE to XML Schema hexBinary.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 BYTE ATTRIBUTES(XSDHexBinary,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>0FB6</Val>
</Root>
```

XSDID

Map BDL STRING or VARCHAR to XML Schema ID.

XSDIDREF

Map BDL STRING or VARCHAR to XML Schema IDREF.

XSDIDREFS

Map BDL STRING or VARCHAR to XML Schema IDREFS.

XSDInt

Map BDL INTEGER or BIGINT to XML Schema int.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER ATTRIBUTES(XSDInt,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>-1258</Val>
</Root>
```

XSDInteger

Map BDL DECIMAL to XML Schema integer.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0) ATTRIBUTES(XSDInteger,XMLName="Val")
```

```
END RECORD
```

```
<Root>
  <Val>12678</Val>
</Root>
```

XSDLanguage

Map BDL STRING or VARCHAR to XML Schema language.

XSDLong

Map BDL BIGINT or DECIMAL to XML Schema long.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(19,0) ATTRIBUTES(XSDLong,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>1267488</Val>
</Root>
```

XSDNCName

Map BDL STRING or VARCHAR to XML Schema NCName.

XSDName

Map BDL STRING or VARCHAR to XML Schema Name.

XSDNegativeInteger

Map BDL DECIMAL to XML Schema negativeInteger.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0)
 ATTRIBUTES(XSDNegativeInteger,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>-4828</Val>
</Root>
```

XSDNMTOKEN

Map BDL STRING or VARCHAR to XML Schema NMToken.

XSDNMTOKENS

Map BDL STRING or VARCHAR to XML Schema NMTokens.

XSDNonNegativeInteger

Map BDL DECIMAL to XML Schema nonNegativeInteger.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
```

```
  val1 DECIMAL(32,0)
 ATTRIBUTES(XSDNonNegativeInteger,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>1589</Val>
</Root>
```

XSDNonPositiveInteger

Map BDL DECIMAL to XML Schema nonPositiveInteger.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0)
 ATTRIBUTES(XSDNonPositiveInteger,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>-8574</Val>
</Root>
```

XSDNormalizedString

Map BDL STRING or VARCHAR to XML Schema normalizedString.

XSDnotation

Not supported.

XSDPositiveInteger

Map BDL DECIMAL to XML Schema positiveInteger.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0)
 ATTRIBUTES(XSDPositiveInteger,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>+41893</Val>
</Root>
```

XSDQName

Map BDL STRING or VARCHAR to XML Schema QName.

XSDShort

Map BDL SMALLINT or BIGINT to XML Schema short.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 SMALLINT ATTRIBUTES(XSDShort,XMLName="Val")
```

```
END RECORD
```

```
<Root>
  <Val>12678</Val>
</Root>
```

XSDString

Map BDL STRING, CHAR, TEXT or VARCHAR to XML Schema string.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 STRING ATTRIBUTES(XSDString,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>Hello world, how are you ?</Val>
</Root>
```

XSDTime

Map BDL DATETIME to XML Schema time.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DATETIME ATTRIBUTES(XSDTime,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>23:16:03.589+01:00</Val>
</Root>
```

XSDToken

Map BDL STRING or VARCHAR to XML Schema token.

XSDUnsignedByte

Map BDL SMALLINT or BIGINT to XML Schema unsignedByte.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 SMALLINT ATTRIBUTES(XSDUnsignedByte,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>254</Val>
</Root>
```

XSDUnsignedInt

Map BDL BIGINT or DECIMAL to XML Schema unsignedInt.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0) ATTRIBUTES(XSDUnsignedInt,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>1267896754</Val>
</Root>
```

XSDUnsignedLong

Map BDL DECIMAL to XML Schema unsignedLong.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 DECIMAL(32,0) ATTRIBUTES(XSDUnsignedLong,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>12678967543233</Val>
</Root>
```

XSDUnsignedShort

Map BDL INTEGER or BIGINT to XML Schema unsignedShort.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER ATTRIBUTES(XSDUnsignedShort,XMLName="Val")
END RECORD
```

```
<Root>
  <Val>65535</Val>
</Root>
```

XML facet constraint attributes

The following attributes are facet constraints depending on the XSD data type used on a simple BDL variable to restrict the allowed value-space.

(Notice that some attributes are allowed only on some XSD data types).

Several facet constraints can be set on the same data type, and a mandatory values is expected (for example, **XSDMinLength="8"**.)

Table 564: Facet constraints between simple BDL and XML data types

Attribute	Definition
XSDLength	Define the exact number of XML character or bytes.
XSDMinLength	Define the minimum number of XML character or bytes.

Attribute	Definition
XSDMaxLength	Define the maximum number of XML character or bytes.
XSDEnumeration	Define a list of allowed values separated by the character **|**.
XSDWhiteSpace	Perform a XML string manipulation before serialization or deserialization.
XSDPattern	Define the regular expression the value has to match.
XSDMinInclusive	Define the inclusive minimum value according to the data type where it is set.
XSDMaxInclusive	Define the inclusive maximum value according to the data type where it is set.
XSDMinExclusive	Define the exclusive minimum value according to the data type where it is set.
XSDMaxExclusive	Define the exclusive maximum value according to the data type where it is set.
XSDTotalDigits	Define the total number of digits.
XSDFractionDigits	Define the number of digits of the fraction part.

XSDLength

Restrict the length of the data to the exact number of XML characters allowed when set on a BDL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a BDL BYTE.

Note:

1. XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
2. XSDMaxLength cannot be used with XSDLength

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDLength="12",
  XMLName="MyString")

DEFINE myByte BYTE ATTRIBUTES(XSDBase64Binary, XSDLength="8000",
  XMLName="MyPicture")
```

XSDMinLength

Restrict the length of the data to the minimum number of XML characters allowed when set on a BDL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a BDL BYTE.

Note:

1. XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
2. XSDMaxLength cannot be used with XSDLength

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDMinLength="12",
  XMLName="MyString")
```

```
DEFINE myByte BYTE ATTRIBUTES(XSDBase64Binary,
 XSDMinLength="8000",
  XMLName="MyPicture")
```

XSDMaxLength

Restrict the length of the data to the maximum number of XML characters allowed when set on a BDL STRING, VARCHAR, CHAR or TEXT, or the number of bytes allowed when set on a BDL BYTE.

Note:

1. XSDMinLength and XSDMaxLength can be used together, but XSDMaxLength value must be greater then XSDMinLength
2. XSDMaxLength cannot be used with XSDLength

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDMaxLength="12",
  XMLName="MyString")
```

```
DEFINE myByte BYTE ATTRIBUTES(XSDBase64Binary,
 XSDMaxLength="8000",
  XMLName="MyPicture")
```

XSDEnumeration

Restrict the allowed value-space to a list of values separated by the characters **|**.

Note:

1. To escape the separator character, simply double it like the following **||**
2. This attribute can be set on any simple BDL variable excepted on XSDBoolean.

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString,
  XSDEnumeration="one|two|three|four", XMLName="MyString")
```

```
DEFINE myDec DECIMAL(3,1) ATTRIBUTES(XSDDecimal,
  XSDEnumeration="12.1|11.8|-24.7", XMLName="MyDecimal")
```

XSDWhiteSpace

Perform a XML string manipulation before serialization or deserialization according to one of the three allowed values:

- **preserve**: the XML string is not modified.
- **replace**: the XML string is modified by replacing each **\n**, **\t**, **\r** by a single space.
- **collapse**: the XML string is modified by replacing each **\n**, **\t**, **\r** by a single space, then each sequence of several spaces are replaced by one single space. Leading and trailing spaces are removed too.

Note:

1. The whiteSpace facet is always performed before any other facet constraints, or serialization or deserialization process.
2. For any BDL variable excepted STRING, CHAR and VARCHAR, only collapse is allowed.

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString,
 XSDWhiteSpace="replace",
  XMLName="MyString")
```

```
DEFINE myDec DECIMAL(3,1) ATTRIBUTES(XSDDecimal,
  XSDWhiteSpace="collapse", XMLName="MyDecimal")
```

XSDPattern

Define a regular expression the value has to match to be serialized or deserialized without any error.

Note:

1. The regular expression is defined in the XML Schema Part 2 specification available here.
2. Backslash characters '\' in a regular expression must be escaped by duplicating it.

Example

```
DEFINE myStr STRING ATTRIBUTES(XSDString, XSDPattern="A.*Z",
  XMLName="MyString")
```

```
DEFINE myZipCode INTEGER ATTRIBUTES(XSDInt, XSDPattern="[0-9]
{5}",
  XMLName="MyZipCode")
```

```
DEFINE myOtherZipCode INTEGER ATTRIBUTES(XSDInt,
 XSDPattern="\\d{5}", XMLName="myOtherZipCode") # regex is \d{5}
 see note
```

XSDMinInclusive

Define the minimum inclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The minimum value cannot exceed the implicit minimum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the minimum value is -32768.

Example

```
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort,
 XSDMinInclusive="-1000",
  XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal,
 XSDMinInclusive="100.01",
  XMLName="MyRate")
```

XSDMaxInclusive

Define the maximum inclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The maximum value cannot exceed the implicit maximum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the maximum value is 32767.

Example

```
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort,
 XSDMaxInclusive="1000",
  XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal,
 XSDMaxInclusive="299.99",
  XMLName="MyRate")
```

XSDMinExclusive

Define the minimum exclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The minimum value cannot exceed or be equal to the implicit minimum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the minimum value is -32768.

Example

```
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort,
 XSDMinExclusive="-1000",
  XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal,
 XSDMinExclusive="100.01",
  XMLName="MyRate")
```

XSDMaxExclusive

Define the maximum exclusive value allowed and depending on the data type where it is set, namely all numeric, date and time data types.

Note: The maximum value cannot exceed or be equal to the implicit maximum value supported by the data type itself or the compiler will complain. For instance, with XSDShort the maximum value is 32767.

Example

```
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort,
 XSDMaxExclusive="1000",
  XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal,
 XSDMaxExclusive="299.99",
  XMLName="MyRate")
```

XSDTotalDigits

Define the maximum number of digits allowed on a numeric data type, fraction part inclusive if there is one.

Note:

1. The total digits value cannot be equal or lower then 0.

2. On a BDL decimal, the total digits value cannot be lower than the precision of the BDL decimal itself.
3. Notice that a decimal without any precision and scale value is a decimal(16), therefore the total digits value must be equal or greater than 16.

Example

```
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort, XSDTotalDigits="4",
  XSDMaxExclusive="1000", XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal,
 XSDTotalDigits="5",
  XSDMaxExclusive="299.99", XMLName="MyRate")
```

XSDFractionDigits

Define the maximum number of digits allowed on the fraction part of a numeric data type.

Note:

1. The fraction digits value set on a BDL data type without XSDDecimal set, can only be 0.
2. On a BDL DECIMAL, the fraction digits value cannot be lower than the scale of the BDL DECIMAL itself, and must be lower than the XSDTotalDigits value if set.

Example

```
DEFINE myCode SMALLINT ATTRIBUTES(XSDShort,
 XSDFractionDigits="0",
  XSDMaxExclusive="1000", XMLName="MyCode")
```

```
DEFINE myRate DECIMAL(4,2) ATTRIBUTES(XSDDecimal,
 XSDTotalDigits="5",
  XSDFractionDigits="3", XSDMaxExclusive="299.99",
 XMLName="MyRate")
```

Customizing XML serialization

The following attributes are used to change the default serialization of BDL into XML, and vice-versa. Some of these attributes cannot have values; for the others a value is mandatory.

The following attributes cannot have values:

Table 565: XML Serialization customizing - Attributes that cannot have values

Attribute	Definition
XMLOptional	Define whether the variable can be missing.
XMLElement	Map a BDL simple data type to an XML Element.
XMLAttribute	Map a BDL simple data type to an XML Attribute.
XMLBase	Set the base type of an XML Schema simpleContent.
XMLAll	Map a BDL Record to an XML Schema all structure.

Attribute	Definition
XMLChoice	Map a BDL Record to an XML Schema choice structure.
XMLSequence	Map a BDL Record to an XML Schema sequence structure.
XMLSimpleContent	Map a BDL Record to an XML Schema simpleContent structure.
XSComplexType	Map a BDL Record type definition to an XML Schema complexType.
XMLList	Map a one-dimensional array to an XML Schema list.
XMLSelector	Define which member of an XMLChoice record is selected.
XMLAny	Map a Xml.DomDocument object to a wildcard XML element node.
XMLAnyAttribute	Map a BDL one-dimensional dynamic array of a record with 3 strings to XML wildcard attributes.

Values are mandatory for the following attributes: (for example, **XMLName="myname"**)

Table 566: XML Serialization customizing - Attributes that must have values

Attribute	Definition
XMLName	Define the XML Name of a variable in an XML document.
XMLNamespace	Define the XML Namespace of a variable in an XML document.
XMLType	Force the XML type name of a variable.
XMLTypenamespace	Force the XML type namespace of a variable.
XSTypename	Define the XML Type Name of a BDL type definition.
XSTypenamespace	Define the XML Type Namespace of a BDL type definition.
XMLElementNamespace	Define the default XML namespace of all children defined as XMLElement in a Record.
XMLAttributeNamespace	Define the default XML namespace of all children defined as XMLAttribute in a Record.

XMLElement (Optional)

Map a BDL simple data type to an XML Element.

Note: The attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
```

```
  val1 INTEGER ATTRIBUTES(XMLElement, XSDunsignedShort,
 XMLName="Val1"),
  rec  RECORD ATTRIBUTES(XMLName="Rec")
    val2 FLOAT  ATTRIBUTES(XMLElement, XMLName="Val2"),
    val3 STRING ATTRIBUTES(XMLElement, XMLName="Val3")
  END RECORD
END RECORD
```

```
<Root>
  <Val1>148</Val1>
  <Rec1>
    <Val2>25.8</Val2>
    <Val3>Hello world</Val3>
  </Rec1>
</Root>
```

XMLAttribute

Map a BDL simple data type to an XML Attribute.

Note:

1. The attribute cannot be set on a type definition.
2. The attribute can only be set on a RECORD's member.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER
 ATTRIBUTES(XMLAttribute,XSDunsignedShort,XMLName="Val1"),
  rec  RECORD ATTRIBUTES(XMLName="Rec1")
    val2  FLOAT ATTRIBUTES(XMLAttribute,XMLName="Val2"),
    val3  STRING ATTRIBUTES(XMLElement,XMLName="Val3")
  END RECORD
END RECORD
```

```
<Root Val1="148">
  <Rec1 Val2="25.8">
    <Val3>Hello world</Val3>
  </Rec1>
</Root>
```

XMLBase

Define the simple BDL variable used as the base type of an XML Schema simpleContent structure.

The attribute can be set on one and only one member of a RECORD defined with the XMLSimpleContent attribute

XMLAll

Map a BDL Record to an XML Schema all structure.

The order in which the record members appear in the XML document is not significant.

Example

```
DEFINE myall RECORD ATTRIBUTES(XMLAll,XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT   ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  val3  STRING  ATTRIBUTES(XMLName="Val3")
```

```
END RECORD
```

```
<Root Val2="25.8">
  <Val3>Hello world</Val3>
  <Val1>148</Val1>
</Root>
```

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <Val3>Hello world</Val3>
</Root>
```

XMLChoice

Map a BDL Record to an XML Schema choice structure. The choice of the record's member is performed at runtime, and changes dynamically according to a mandatory member. This specific member must be of type SMALLINT or INTEGER, and have an XMLSelector attribute set. The XMLChoice attribute also supports a "nested" value that removes the surrounding XML tag.

Note:

1. Valid selector values are indexes referring to members considered as XML element nodes. All other values will raise XML runtime errors.
2. Nested choice records cannot be defined as main variables; there must always be a surrounding variable.

Example

```
DEFINE mychoice RECORD ATTRIBUTES(XMLChoice,XMLName="Root")
  val1  INTEGER   ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT     ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  sel   SMALLINT  ATTRIBUTES(XMLSelector),
  val3  STRING    ATTRIBUTES(XMLName="Val3")
END RECORD
```

Case where "sel" value is 4

```
<Root Val2="25.8">
  <Val3>Hello world</Val3>
</Root>
```

Case where "sel" value is 1

```
<Root Val2="25.8">
  <Val1>148</Val1>
</Root>
```

Nested example:

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER   ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT     ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  choice RECORD   ATTRIBUTES(XMLChoice="nested")
    choice1   INTEGER   ATTRIBUTES(XMLName="ChoiceOne"),
    choice2   FLOAT     ATTRIBUTES(XMLName="ChoiceTwo"),
    nestedSel SMALLINT  ATTRIBUTES(XMLSelector)
  END RECORD,
  val3  STRING    ATTRIBUTES(XMLName="Val3")
END RECORD
```

Case where "nestedSel" value is 1

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <ChoiceOne>6584</ChoiceOne>
  <Val3>Hello world</Val3>
</Root>
```

Case where "nestedSel" value is 2

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <ChoiceTwo>85.8</ChoiceTwo>
  <Val3>Hello world</Val3>
</Root>
```

XMLSequence (Optional)

Map a BDL RECORD to an XML Schema sequence structure.The order in which the record members appear in the XML document must match the order of the BDL RECORD. The XMLSequence attribute also supports a "nested" value that removes the surrounding XML tag.

Note: Nested sequence records cannot be defined as main variables; there must always be a surrounding variable.

Example

```
DEFINE mysequence RECORD ATTRIBUTES(XMLSequence,XMLName="Root")
  val1  INTEGER   ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT     ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  val3  STRING    ATTRIBUTES(XMLName="Val3")
END RECORD
```

```
<Root Val2="25.8">
  <Val1>-859</Val1>
  <Val3>Hello world</Val3>
</Root>
```

Nested example:

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER   ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT     ATTRIBUTES(XMLAttribute,XMLName="Val2"),
  sequence RECORD   ATTRIBUTES(XMLSequence="nested")
    seq1   INTEGER   ATTRIBUTES(XMLName="SeqOne"),
    seq2   FLOAT     ATTRIBUTES(XMLName="SeqTwo")
  END RECORD,
  val3  STRING    ATTRIBUTES(XMLName="Val3")
END RECORD
```

```
<Root Val2="25.8">
  <Val1>148</Val1>
  <SeqOne>6584</SeqOne>
  <SeqTwo>85.597</SeqTwo>
  <Val3>Hello world</Val3>
</Root>
```

XMLSimpleContent

Map a BDL RECORD to an XML Schema simpleContent structure.

Note: One member must have the XMLBase attribute; all other members must have an XMLAttribute attribute. If not, the compiler complains.

Example

```
DEFINE mysimpletype RECORD
 ATTRIBUTES(XMLSimpleContent,XMLName="Root")
  base  STRING  ATTRIBUTES(XMLBase),
  val1  INTEGER ATTRIBUTES(XMLAttribute,XMLName="Val1"),
  val2  FLOAT   ATTRIBUTES(XMLAttribute,XMLName="Val2")
END RECORD
```

```
<Root Val1="148" Val2="25.8">
  Hello
</Root>
```

XSComplexType

Map a BDL RECORD type definition to an XML Schema complexType.

Note: You can have one member as a nested sequence or choice, or as an XMLList array with a nested sequence or choice as the array's elements; all other members must have an XMLAttribute attribute. If not, the compiler complains.

Example

```
TYPE mycomplextype RECORD ATTRIBUTES(XSComplexType,
  XSTypeName="MyComplexType",XSTypeNamespace="http://
tempuri.org")
         name DYNAMIC ARRAY ATTRIBUTES(XMLList) OF RECORD
              ATTRIBUTES(XMLSequence="nested")
     firstname STRING ATTRIBUTES(XMLName="FirstName"),
     lastname  STRING ATTRIBUTES(XMLName="LastName")
END RECORD,
  date  DATE ATTRIBUTES(XMLAttribute,XMLName="Date")
END RECORD
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://tempuri.org"
 elementFormDefault="qualified" >
  <xsd:complexType name="MyComplexType">
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="FirstName" type="xsd:string" />
      <xsd:element name="LastName"  type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="Date" type="xsd:date" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

XMLList

Map a one dimensional array to an XML Schema element that has more than one occurrence.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  list  DYNAMIC ARRAY ATTRIBUTES(XMLList) OF STRING
 ATTRIBUTE(XMLName="Element"),
  val2  FLOAT   ATTRIBUTES(XMLName="Val2")
```

```
END RECORD
```

```
<Root>
  <Val1>148</Val1>
  <Element>hello</Element>
  <Element>how</Element>
  <Element>are</Element>
  <Element>you</Element>
  <Val2>0.58</Val2>
</Root>
```

Note: It is not possible to define an XMLList attribute on a main array.

XMLSelector

Define the index of the candidate among all members of an XMLChoice record that will be serialized or de-serialized at runtime.

The index starts at 1.

The selector data type must be a SMALLINT or a INTEGER.

XMLAny

Map a Xml.DomDocument object to a wildcard XML element:

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",XMLNamespace="http://
tempuri.org")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  any   Xml.DomDocument ATTRIBUTES(XMLAny,XMLNamespace="##other"),
  val2  FLOAT   ATTRIBUTES(XMLName="Val2")
END RECORD
```

```
<pre:Root xmlns:pre="http://tempuri.org" >
  <pre:Val1>148</pre:Val1>
  <pre2:Doc xmlns:pre2="http://www.mycompany.com">
     <pre2:Element>how</pre2:Element>
     <pre2:Element>are</pre2:Element>
     <pre2:Element>you</pre2:Element>
  </pre2:Doc>
  <pre:Val2>
0.58</pre:Val2>
</pre:Root>
```

Note: Associated with XMLAny, the XMLNamespace attribute requires either:

- A list of space-separated URIs to accept each attribute belonging to one of this namespace URI as a wildcard attribute.
- The value **##any** to accept any attribute as a wildcard attribute.
- The value **##other** to accept any attribute not in the main schema namespace as a wildcard attribute.

For example:

- If XMLNamespace="http://tmpuri.org http://www.mycompany.com", then only the XML documents belonging to one of those namespaces will be accepted and serialized (or de-serialized) into the Xml.DomDocument object.
- If XMLNamespace="##any", then any XML document will be accepted and serialized (or de-serialized) into the Xml.DomDocument object.
- If XMLNamespace="##other", then any XML document not belonging to the targetNamespace of the XML Schema where the any definition is used will be accepted and serialized (or de-serialized) into the Xml.DomDocument object.

XMLAnyAttribute

Map a one-dimensional dynamic array to wildcard XML attributes.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
 XMLNamespace="http://tempuri.org")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT   ATTRIBUTES(XMLName="Val2"),
  attr  STRING  ATTRIBUTES(XMLName="Attr", XMLAttribute),
  any   DYNAMIC ARRAY ATTRIBUTES(XMLAnyAttribute,
 XMLNamespace="##other")OF RECORD
    ns  STRING,
    name  STRING,
    value  STRING
  END RECORD
END RECORD
```

```
<pre:Root xmlns:pre="http://tempuri.org" pre:Attr="10"
 xmlns:pre2="http://www.mycompany.com" pre2:AnyAttr1="10"
 pre2:AnyAttr2="">
  <pre:Val1>148</pre:Val1>
  <pre:Val2>0.58</pre:Val2>
</pre:Root>
```

Note:

1. The attribute **XMLAnyAttribute** is only allowed on a one-dimensional dynamic array of a record with three members of type STRING. The **first** member is for the **namespace** of the wildcard attribute, the **second** member is for the **name** of the wildcard attribute, and the **third** member is for the **value** of the wildcard attribute. **The name cannot be null.**
2. Associated with the XMLAnyAttribute, the XMLNamespace attribute requires either:
   - A list of space-separated URIs to accept each attribute belonging to one of the namespace URIs as a wildcard attribute.
   - The value **##any** to accept any attribute as a wildcard attribute.
   - The value **##other** to accept any attribute not in the main schema namespace as a wildcard attribute.

   For example:
   - If XMLNamespace="http://tmpuri.org http://www.mycompany.com", then only the attributes belonging to one of those namespaces will be accepted and serialized (or deserialized) into the array.
   - If XMLNamespace="##any", then any attribute will be accepted and serialized (or deserialized) into the array.
   - If XMLNamespace="##other", then any attributes not belonging to the targetNamespace of the XML Schema where the anyAttribute definition is used will be accepted and serialized (or deserialized) into the array.

XMLName

Define the name of a variable in an XML document.

Note: The attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1  INTEGER ATTRIBUTES(XMLName="Val1"),
  val2  FLOAT,
  val3  INTEGER ATTRIBUTES(XMLName="Val3")
END RECORD
```

```
<Root>
  <Val1>148</Val1>
  <val2>0.5</val2>
  <Val3>-18547</Val3>
</Root>
```

XMLNamespace

Define the namespace of a variable in an XML document.

Note:

1. If the attribute is set on a Record, by default all members defined as XMLElement of that record are in the same namespace.
2. If the attribute is set on an Array, by default all elements defined as XMLElement of that array are in the same namespace.
3. The attribute cannot be set on a type definition.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
 XMLNamespace="http://tempuri.org")
  attr1 INTEGER ATTRIBUTES(XMLAttribute,XMLName="Attr1"),
  val1  FLOAT   ATTRIBUTES(XMLName="Val1", XMLNamespace="http://
www.mycompany.com"),
  val2  INTEGER ATTRIBUTES(XMLName="Val2"),
  attr2 STRING  ATTRIBUTES(XMLAttribute, XMLName="Attr2",
    XMLNamespace="http://anyuri.org")
END RECORD
```

```
<fjs1:Root xmlns:fjs1="http://tempuri.org" Attr1="158"
 xmlns:fjs3="http://anyuri.org" fjs3:Attr2="Hello">
  <fjs2:Val1 xmlns:fjs2="http://www.mycompany.com">0.5</
fjs2:Val1>
  <fjs1:Val2>-18547</fjs1:Val2>
</fjs1:Root>
```

XMLType

Force the XML type name of a variable by adding xsi:type at serialization or by checking xsi:type at deserialization.

Note: The attribute must be used with the XMLTypenamespace attribute; otherwise, the compiler complains.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
                               XMLNamespace="http://tempuri.org")
  val1  FLOAT   ATTRIBUTES(XMLName="Val1"),
  val2  INTEGER ATTRIBUTES(XMLName="Val2",
```

```
                            XMLType="MyRecord",
                            XMLTypenamespace="http://
mynamespace.org")
END RECORD
```

```
<fjs1:Root xmlns:fjs1="http://tempuri.org">
  <fjs1:Val1>0.5</fjs1:Val1>
  <fjs1:Val2 xmlns:fjs2="http://mynamespace.org"
   xsi:type="fjs2:MyRecord">-18547</fjs1:Val2>
</fjs1:Root>
```

XMLTypenamespace

Force the XML type namespace of a variable by adding xsi:type at serialization or by checking xsi:type at de-serialization.

Note: The attribute must be used with the XMLType attribute; otherwise the compiler complains.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
                               XMLNamespace="http://tempuri.org")
  val1  FLOAT   ATTRIBUTES(XMLName="Val1"),
  val2  INTEGER ATTRIBUTES(XMLName="Val2",
                            XMLType="MyRecord",
                            XMLTypenamespace="http://
mynamespace.org")
END RECORD
```

```
<fjs1:Root xmlns:fjs1="http://tempuri.org">
  <fjs1:Val1>0.5</fjs1:Val1>
  <fjs1:Val2 xmlns:fjs2="http://mynamespace.org"
   xsi:type="fjs2:MyRecord">-18547</fjs1:Val2>
</fjs1:Root>
```

XSTypename

Define the XML Schema name of a BDL type definition.

Note:

1. The attribute must be used with the XSTypenamespace attribute; otherwise the compiler complains.
2. The attribute is only allowed on a type definition.

Example

```
TYPE myType RECORD ATTRIBUTES(XMLSequence,
                              XSTypeName="MyFirstType",
                              XSTypeNamespace="http://
tempuri.org" )
  val1  FLOAT   ATTRIBUTES(XMLElement,XMLName="Val1"),
  val2  INTEGER
 ATTRIBUTES(XMLElement,XMLName="Val2",XMLOptional),
  attr  STRING  ATTRIBUTES(XMLAttribute,XMLName="Attr")
END RECORD
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
 targetNamespace="http://tempuri.org"
 elementFormDefault="qualified" >
  <xsd:complexType name="MyFirstType">
    <xsd:sequence>
      <xsd:element name="Val1" type="xsd:double" />
      <xsd:element name="Val2" type="xsd:int" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="Attr" type="xsd:string"
 use="required" />
  </xsd:complexType>
</xsd:schema>
```

XSTypenamespace

Define the XML Schema namespace of a BDL type definition.

Note:

1. The attribute must be used with the XSType attribute; otherwise the compiler complains.
2. The attribute is only allowed on a type definition.

Example

```
TYPE myType RECORD ATTRIBUTES(XMLChoice,
                              XSTypeName="MyFirstChoice",
                              XSTypeNamespace="http://
tempuri.org" )
  val1  FLOAT   ATTRIBUTES(XMLElement,XMLName="Val1"),
  val2  INTEGER
 ATTRIBUTES(XMLElement,XMLName="Val2",XMLOptional),
  attr  STRING
 ATTRIBUTES(XMLAttribute,XMLName="Attr",XMLOptional),
  set   INTEGER ATTRIBUTES(XMLSelector)
END RECORD
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://tempuri.org"
 elementFormDefault="qualified" >
  <xsd:complexType name="MyFirstChoice">
    <xsd:choice>
      <xsd:element name="Val1" type="xsd:double" />
      <xsd:element name="Val2" type="xsd:int" minOccurs="0" />
    </xsd:choice>
    <xsd:attribute name="Attr" type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
```

XMLElementNamespace

Define the default namespace of all members of a record also defined as XML elements.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
    XMLNamespace="http://tempuri.org",
    XMLElementNamespace="http://www.mycompany.com")
  val1  FLOAT   ATTRIBUTES(XMLElement,XMLName="Val1"),
  val2  INTEGER ATTRIBUTES(XMLElement,XMLName="Val2"),
  attr  STRING  ATTRIBUTES(XMLAttribute,XMLName="Attr"),
```

```
                END RECORD
```

```
<fjs1:Root xmlns:fjs1="http://tempuri.org" Attr="Hello"
 xmlns:fjs2="http://www.mycompany.com">
  <fjs2:Val1>0.5</fjs2:Val1>
  <fjs2:Val2>-18547</fjs2:Val2>
</fjs1:Root>
```

XMLAttributeNamespace

Define the default namespace of all members of a record also defined as XML attributes.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root",
    XMLNamespace="http://tempuri.org",
    XMLAttributeNamespace="http://www.mycompany.com")
  val1  FLOAT   ATTRIBUTES(XMLElement,XMLName="Val1"),
  val2  INTEGER ATTRIBUTES(XMLElement,XMLName="Val2"),
  attr1 STRING  ATTRIBUTES(XMLAttribute,XMLName="Attr1"),
  attr2 DATE    ATTRIBUTES(XMLAttribute,
    XMLName="Attr2",XMLNamespace="http://anyuri.org"),
END RECORD
```

```
<fjs1:Root xmlns:fjs1="http://tempuri.org"
 fjs2:Attr1="Hello" xmlns:fjs2="http://www.mycompany.com"
 xmlns:fjs3="http://anyuri.org" fjs3:Attr2="2006-06-24">
  <fjs1:Val1>0.5</fjs1:Val1>
  <fjs1:Val2>-18547</fjs1:Val2>
</fjs1:Root>
```

XMLOptional

Define whether the variable can be missing or not. It specifies how a NULL value is interpreted in XML.

Note:

1. The attribute cannot be set on a type definition.
2. The attribute cannot be set if the main variable is not a RECORD.

Example

```
DEFINE myVar RECORD ATTRIBUTES(XMLName="Root")
  val1 INTEGER ATTRIBUTES(XSDint,XMLName="ValOne"),
  val2 FLOAT ATTRIBUTES(XSDdouble,XMLName="ValTwo",XMLOptional)
END RECORD
```

```
<Root>
  <ValOne>458</ValOne>
  <ValTwo>58.48</ValTwo>
</Root>
```

```
<Root>
  <ValOne>458</ValOne>
</Root>
```

Error handling in GWS calls (STATUS)

In case of problem, the methods of GWS classes can throw an exception and set the STATUS variable with the appropriate error number.

By default, the program will stop if an exception is thrown. You can trap the GWS errors with a `WHENEVER ERROR` handler or with a `TRY/CATCH` block. In the next example, the `readTextRequest()` API is surrounded by a `TRY/CATCH` block:

```
DEFINE req com.HTTPServiceRequest,
       data STRING
...
LET req = com.WebServiceEngine.getHTTPServiceRequest(5)
...
TRY
   ...
   CALL req.readTextRequest() RETURNING data
   ...
CATCH
   CALL show_err(SFMT("Unexpected HTTP request read exception: %1", STATUS))
END TRY
```

For some errors, a human-readable description of the error code is available in the `SQLCA.SQLERRM` register.

Interruption handling in GWS calls (INT_FLAG)

In order to check if an application has been interrupted, GWS tests the INT_FLAG. If the INT_FLAG is set to TRUE, then the GWS function processing is interrupted and an exception is raised with error code -15553.

> **Important:** Make sure that the INT_FLAG register is set to FALSE before calling a GWS function: For example, after a dialog was stopped with cancel action, the INT_FLAG is set to TRUE. If you do not reset INT_FLAG to FALSE, the next GWS function may be canceled.

As a general rule, surroung GWS calls with a `TRY/CATCH` block (or `WHENEVER ERROR` handler), to detect both communication errors and interruptions:

```
TRY
   LET INT_FLAG=FALSE
   ...
   CALL req.sendXMLRequest(doc)
   ...
CATCH
   CASE STATUS
      WHEN -15553 -- TCP socket error
        IF INT_FLAG THEN
           MESSAGE "An interruption occured."
        ELSE
           ERROR "TCP socket error: ", SQLCA.SQLERRM
        END IF
      ...
   END CASE
END TRY
```

Server API functions - version 1.3 only

The following table lists the APIs to create a Web Services server in BDL.

> **Note:** These functions are valid for backwards compatibility, but they are not the preferred way to handle Genero Web Services. See the GWS COM Library classes and methods.

Table 567: APIs to create a Web Services server in BDL (version 1.3 only)

Function	Description
fgl_ws_server_setNamespace()	Defines the namespace of the service on the Web.
fgl_ws_server_start()	Creates and starts the Web Service server.
fgl_ws_server_publishFunction()	Publishes the BDL function as a Web Function.
fgl_ws_server_generateWSDL()	Generates the WSDL file.
fgl_ws_server_process()	Waits for and processes incoming SOAP requests.
fgl_ws_server_setFault()	Sets the SOAP fault string for a Web Function.
fgl_ws_server_getFault()	Retrieves the fault string that was set for a Web Function, for testing purposes.

fgl_ws_server_setNamespace() (version 1.3)

Purpose

This function defines the namespace of the service on the Web and must be called first, before all other functions of the API.

Syntax

```
FUNCTION fgl_ws_server_setNamespace(namespace VARCHAR)
```

Parameters

- *namespace* is the name of the namespace.

Return values

- None

Example

```
CALL fgl_ws_server_setNamespace("http://tempuri.org/")
```

fgl_ws_server_start() (version 1.3)

Purpose

This function creates and starts the server. For development or testing purposes, you may start a Web Service server as a single server where only one request at a time will be able to be processed. For deployment, you may start a Web Service server with an application server able to handle several connections at one time using a load-balancing algorithm. The value of the parameter passed to the function determines which method is used.

Syntax

```
FUNCTION fgl_ws_server_start(tcpPort VARCHAR)
```

Parameters

- *tcpPort* is a string representing either:
  - the **socket port number** (for a single Web Service server)
  - the **host** and **port** value separated by a colon (for a Web Service server connecting to an application server). The value of **port** is an offset beginning at 6400.

    Note: If the FGLAPPSERVER environment variable is set, the tcpPort value is ignored, and replaced by the value of FGLAPPSERVER.

Return values

- None

Examples:

To start a standalone Web Service server:

```
CALL fgl_ws_server_start("8080") # A single Server is listening
                                 # on port number: 8080
```

To start a Web Service server attempting to connect to an application server:

```
CALL fgl_ws_server_start("zeus:5") # The server attempt to
connect
                                   # to an application server
located
                                   # on host zeus and listening
                                   # on the port number 6405
```

Possible runtime errors

- -15504: PORT_ALREADY_USED
- -15514: PORT_NOT_NUMERIC
- -15515: NO_AS_FOUND
- -15516: LICENSE_ERROR

fgl_ws_server_publishFunction() (version 1.3)

Purpose

This function publishes the given BDL function as a Web-Function on the Web.

Syntax

```
FUNCTION fgl_ws_server_publishFunction(
  operationName VARCHAR,
  inputNamespace VARCHAR,
  inputRecordName VARCHAR,
  outputNamespace VARCHAR,
  outputRecord VARCHAR,
  functionName VARCHAR)
```

Parameters

- *operationName* is the name by which the operation will be defined on the Web. The name is case sensitive.

- *inputNamespace* is the namespace of the incoming operation message.
- *inputRecordName* is the name of the BDL record representing the Web Function input message or "" if there is none.
- *outputNamespace* is the namespace of the outgoing operation message.
- *outputRecord* is the name of the BDL record representing the Web Function output message or "" if there is none.
- *functionName* is the name of the BDL function that is executed when the Web Service engine receives a request with the operation name defined above.

Return values

- None

Example

```
CALL fgl_ws_server_publishFunction(
 "MyWebOperation",
 "http://www.tempuri.org/webservices/","myfunction_input",
 "http://www.tempuri.org/webservices/","myfunction_output",
 "my_bdl_function")
```

Possible runtime errors

- -15503: FUNCTION_ALREADY_EXISTS
- -15501: FUNCTION_ERROR
- -15502: FUNCTION_DECLARATION_ERROR
- -15512: INPUT_VARIABLE_ERROR
- -15513: OUTPUT_VARIABLE_ERROR
- -15503: BDL_XML_ERROR
- -15518: INPUT_NAMESPACE_MISSING
- -15519: OUTPUT_NAMESPACE_MISSING

fgl_ws_server_generateWSDL() (version 1.3)

Purpose

This function generates the WSDL file according to the BDL-server program.

Syntax

```
FUNCTION fgl_ws_server_generateWSDL(
  serviceName VARCHAR,
  serviceLocation VARCHAR,
  fileName VARCHAR )
  RETURNING resultStatus INTEGER
```

Parameters

- *serviceName* is the name of the web service.
- *serviceLocation* is the URL of the server.
- *fileName* is the name of the file that will be generated.

Return values

- *resultStatus* is a status containing:

- 0 if the file has been correctly generated.
- Any other values if the operation has failed.

Example

```
DEFINE mystatus INTEGER

LET mystatus=fgl_ws_server_generateWSDL(
 "CustomerService",
 "http://localhost:8080",
 "C:/mydirectory/myfile.wsdl")

IF mystatus=0 THEN
  DISPLAY "Generation of WSDL done..."
ELSE
  DISPLAY "Generation of WSDL failed!"
END IF
```

fgl_ws_server_process() (version 1.3)

Purpose

This function waits for an incoming SOAP request for a given time (in seconds) and then processes the request, or returns, if there has been no request during the given time. If a DEFER INTERRUPT or DEFER QUIT instruction has been defined, the function returns even if it is an infinite wait.

Syntax

```
FUNCTION fgl_ws_server_process(timeout INTEGER)
   RETURNING resultStatus INTEGER
```

Parameters

- *timeout* is the maximum waiting time for an incoming request (or -1 for an infinite wait)

Return values

- *resultStatus* is a status containing:
  - 0 Request has been processed
  - -1 Timeout has been reached
  - -2 The application server asks the runner to shutdown
  - -3 A client connection has been unexpectedly broken
  - -4 An interruption has been raised
  - -5 The HTTP header of the request was incorrect
  - -6 The SOAP envelope was malformed
  - -7 The XML document was malformed

Example

```
DEFER INTERRUPT
DEFINE mystatus INTEGER
LET mystatus=fgl_ws_server_process(5)# wait for 5 seconds
                                     # for incoming request
IF mystatus=0 THEN
  DISPLAY "Request processed."
```

```
END IF
IF mystatus=-1 THEN
  DISPLAY "No request."
END IF
IF mystatus=-2 THEN # terminate the application properly
  EXIT PROGRAM      # if connected to application server
END IF
IF mystatus=-3 THEN
  DISPLAY "Client connection unexpectedly broken."
END IF
IF mystatus=-4 THEN
  DISPLAY "Server process has been interrupted."
END IF
IF mystatus=-5 THEN
  DISPLAY "Malformed or bad HTTP request received."
END IF
IF int_flag<>0 THEN
  LET int_flag=0
  EXIT PROGRAM
END IF
```

fgl_ws_server_setFault() (version 1.3)

Purpose

This function can be called in a published Web-Function in order to return a SOAP fault string to the client at the end of the function's execution.

Syntax

```
FUNCTION fgl_ws_server_setFault(faultMessage VARCHAR)
```

Parameters

- *faultMessage* is a string containing the SOAP Fault string that will be returned to the client.

Return values

- None

Example

```
CALL fgl_ws_server_setFault(
   "The server is not able to manage this request.")
```

fgl_ws_server_getFault() (version 1.3)

Purpose

This function retrieves the last fault string the user has set in a Web-Function, or an empty string if there is none.

> **Note:** This function is only for testing the Web Services functions before they are published on the Web.

Syntax

```
FUNCTION fgl_ws_server_getFault()
  RETURNING faultMessage VARCHAR
```

Parameters

- None

Return values

- *faultMessage* is the string containing the SOAP Fault string.

Example

```
DEFINE div_input RECORD
  a INTEGER,
  b INTEGER
  END RECORD

DEFINE div_output RECORD
  result INTEGER
  END RECORD

FUNCTION TestServices()
   DEFINE string VARCHAR(100)
   ...
   # Test divide by zero operation
   LET div_input.a=15
   LET div_input.b=0
   CALL service_operation_div()
   LET string=fgl_ws_server_getFault()
   DISPLAY "Operation div error: ", string
   ...
END FUNCTION

FUNCTION service_operation_div()
   ...
   IF div_input.b = 0 THEN
     CALL fgl_ws_server_setFault("Divide by zero")
     RETURN
   END IF
   ...
END FUNCTION
```

Configuration API functions - version 1.3 only

The following table lists those configuration API functions that can modify the behavior of the Web Services engine for the client as well as for the server.

> **Note:** These functions are valid for backwards compatibility, but they are not the preferred way to handle Genero Web Services. See the COM Library classes and methods.

Table 568: Configuration API functions for Web Services engine behavior modification

Function	Description
`fgl_ws_setOption()`	Sets an option flag with a given value.
`fgl_ws_getOption()`	Returns the value of an option flag.

fgl_ws_setOption()
This function sets an option flag with a given value, changing the global behavior of the Web Services engine.

Syntax

```
FUNCTION fgl_ws_setOption(optionName VARCHAR,
                          optionValue INTEGER)
```

Parameters

- *optionName* is one of the global flags.
- *optionValue* is the value of the flag.

Return values

- None

Example

```
CALL fgl_ws_setOption("http_invoketimeout",5)
```

Possible runtime errors

- -15511: INVALID_OPTION_NAME

fgl_ws_getOption()
This function returns the value of an option flag.

Syntax

```
FUNCTION fgl_ws_getOption(optionName VARCHAR)
     RETURNING optionValue INTEGER
```

Parameter

- *optionName* is one of the global flags.

Return values

- *optionValue* is the value of the flag.

Example

```
DEFINE value INTEGER
LET value=fgl_ws_getOption("http_invoketimeout")
```

Possible runtime errors

- -15511: INVALID_OPTION_NAME

Option flags

Table 569: Option flags

Flags	Client or Server	Commentary
http_invoketimeout	Client	Defines the maximum time in seconds a client has to wait before the client connection raises an error because the server is not responding. A value of -1 means that it has to wait until the server responds. The default value is -1.
tcp_connectiontimeout	Client	Defines the maximum time in seconds a client has to wait for the establishment of a TCP connection with a server. A value of -1 means infinite wait. The default value is 30 seconds except for Windows™, where it is 5 seconds.
soap_ignoretimezone	Both	Defines if, during the marshalling and unmarshalling process of a BDL DATETIME data type, the SOAP engine should ignore the time zone information. A value of zero means false. The default value is false.
soap_usetypedefinition	Both	Defines if the Web Services engine must specify the type of data in all SOAP requests. This will add an "xsi:type" attribute to each parameter of the request. A value of zero means false. The default value is false.
wsdl_decimalsize	Server	Defines if, during the WSDL generation, the precision and scale of a DECIMAL variable will be taken into account. See WSDL generation option notes on page 2555. A value of zero means false. The default value is true.
wsdl_arraysize	Server	Defines if, during the WSDL generation, the size of a BDL

Flags	Client or Server	Commentary
		array will be taken into account. See WSDL generation option notes on page 2555. A value of zero means false. The default value is true.
wsdl_stringsize	Server	Defines if, during the WSDL generation, the size of a CHAR or VARCHAR variable will be taken into account. See WSDL generation option notes on page 2555. A value of zero means false. The default value is true.

WSDL generation option notes

1. For a BDL type **DECIMAL(5,2)**, when **wsdl_decimalsize** is TRUE, the generated WSDL file contains the total size and the size of the fractional part of the decimal:

```
<types>
   <schema xmlns="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.mycompany.com/types/">
      <simpleType name="echoDecimal5_2_a_dec5_2_out_FGLDecimal">
         <restriction base="decimal">
            <totalDigits value="5" />
            <fractionDigits value="2" />
         </restriction>
      </simpleType>
   </schema>
</types>
<message name="echoDecimal5_2">
  <part name="dec5_2" type="f:echoDecimal5_2_a_dec5_2_in_FGLDecimal" />
</message>
```

   When **wsdl_decimalsize** is FALSE, the total size and the size of the fractional part are not mentioned:

```
<message name="echoDecimal5_2">
  <part name="dec5_2" type="xsd:decimal" />
</message>
```

2. If the WSDL file does not contain the size, the client application has no way of knowing the size. In this scenario, a default value for the size is generated. For example, the exported server type **DECIMAL(5,2)** becomes a **DECIMAL(32)** on the client side.
3. It is better to keep the options **wsdl_arraysize**, **wsdl_stringsize** and **wsdl_decimalsize** set to TRUE (default) so that the BDL client application can do an exact type mapping.

Using fglwsdl to generate code from WSDL or XSD schemas

This section covers the different options of the fglwsdl tool. This tool is used to generate .4gl code from WSDL / XSD schemas.

Generate TYPE definitions from global XML elements or attributes

If a WSDL or a XSD has global XML elements or attributes defined with an inlined type, the -fInlineTypes option of fglwsdl generates a TYPE definition representing that inline type, using the original

WSDL/XSD name of the element or attribute, concatenated with the string 'GlobalAttributeType' or 'GlobalElementType'.

For example, when using fglwsdl -fInlineTypes, the following schema:

```
<xs:element name="getAlertListRequestFlow">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="getAlertListRequest"
 type="amp:getAlertListRequest" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Will produce:

```
TYPE tgetAlertListRequestFlowGlobalElementType RECORD
         ATTRIBUTES(XMLSequence)
   getAlertListRequest tgetAlertListRequest
      ATTRIBUTES(XMLName="getAlertListRequest")
END RECORD
DEFINE getAlertListRequestFlow tgetAlertListRequestFlowGlobalElementType
          ATTRIBUTES(XMLName="getAlertListRequestFlow")
```

Instead of:

```
DEFINE getAlertListRequestFlow RECORD
          ATTRIBUTES(XMLName="getAlertListRequestFlow",XMLSequence)
   getAlertListRequest tgetAlertListRequest
      ATTRIBUTES(XMLName="getAlertListRequest")
END RECORD
```

Mobile applications

These topics cover programming subjects about mobile applications

- Types of Genero Mobile apps on page 2557
- Language limitations on page 2560
- Environment variables on mobile on page 2560
- App localization on page 2560
- Apps user interface on page 2561
- Database support on mobile devices on page 2567
- Web Services on mobile devices on page 2569
- Accessing device functions on page 2569
- Debugging a mobile app on page 2570
- Deploying mobile apps on page 2572
- Push notifications on page 2599

Types of Genero Mobile apps

A mobile app is an app that runs on a mobile device, such as a tablet or a phone. There are different types of mobile apps.

When you are developing your app, and you execute the app on your development machine for display on a device or emulator, you are running the app in *development mode*.

When you follow the procedure to deploy your application to the device for testing or to distribute your app to your end users, you have a *deployed app*. A deployed app might be an app that executes irrespective of network availability, it might be an app that accesses device peripherals, it might be an app that requires network access.

Here are the categories, or application types, for deployed apps.

Standalone apps

A *standalone app* has the DVM and display client entirely on the mobile device. This app executes irrespective of network availability and can access device peripherals such as the camera, contacts, email, calendar, GPS, and storage via exposed APIs (front calls). For database needs, this app can only connect to a local SQLite database.

> **Note:** The DVM refers to the dynamic virtual machine, which is the process that runs the app.

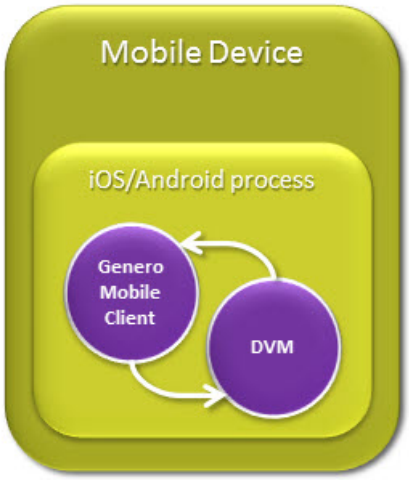


Figure 115: Standalone app

Partially-connected apps

A *partially-connected app* has the DVM and display client entirely on the mobile device, yet this app includes items that require a network connection. This app must be able to run when no network connection is available. This app uses a network API to talk to any back-end.

Examples include:

- Web Services performed with JSON over HTTP; use RESTful methods to write data synchronization routines. With this example, business logic executes within the device's Virtual Machine and the user is able to store captured data to a local SQLite database. When the network becomes available, the user synchronizes the stored data with the remote server's database.

  Note: As of Genero Mobile 1.1, you can also write a Web Service using SOAP.

- A web component that runs Google Maps.

This app first operates without a network connection, and must be able to run without a network connection. Once network connectivity is restored, the app can perform network-dependent tasks such as synchronizing with a remote database, make a web service call, or use a web component.

If you are using GMI, and the device goes into standby mode, the application does not run in the background and activities with the network are suspended.

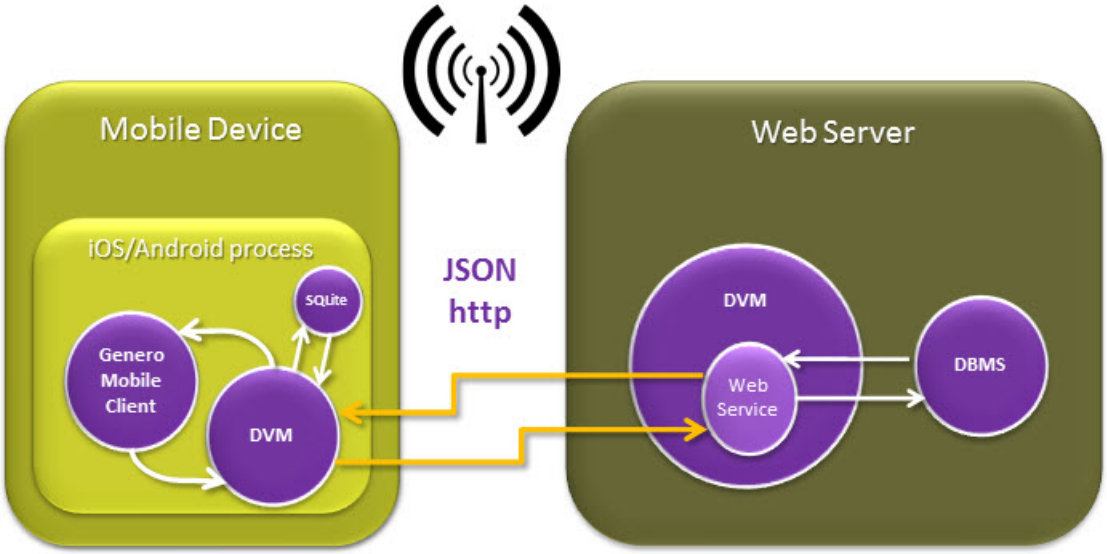


Figure 116: Partially-connected app (example showing data synchronization via JSON)

Client-server apps

With a *client-server app* or *connected app*, the bulk of the app runs on a remote server and the display client sits on the mobile device.

As with any deployed app, this app first starts on the mobile device; the DVM for the deployed app runs on the mobile device. The role of the deployed app, however, is to connect to a remote corporate server as an online terminal. It is the deployed app that launches the remote application using the `runOnServer` frontcall. The remote application's DVM and business logic reside on the remote server, somewhere in the network. The remote application is not limited to a SQLite database.

In the event that the network is interrupted, the Genero Mobile client app is suspended until service resumes.

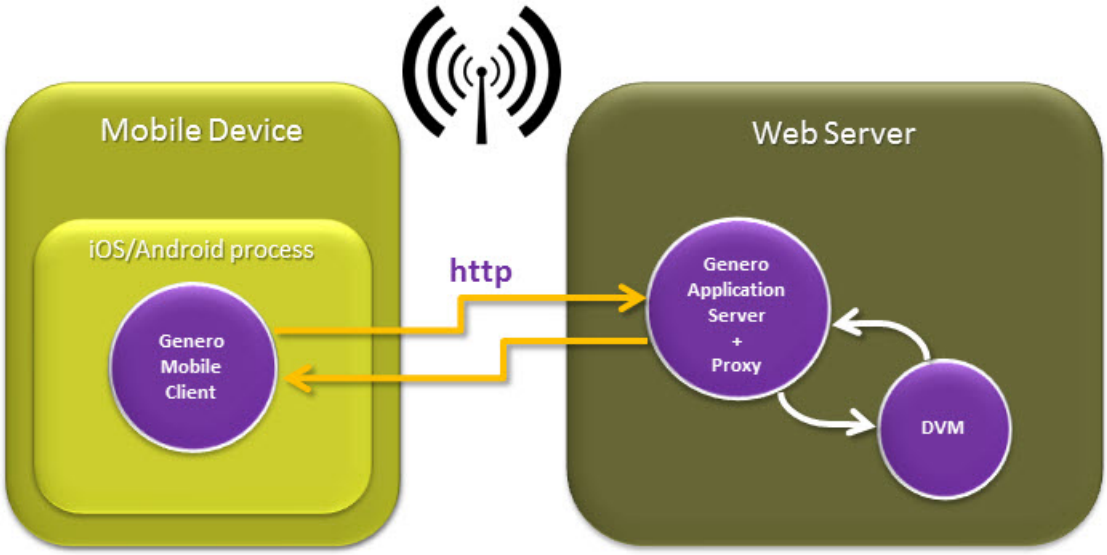


Figure 117: Client-server app

See *Running mobile apps on an application server* in the *Genero Business Development Language User Guide* for more information.

Language limitations

Parts of the Genero language are not supported on mobile devices.

> **Important:** This topic is provided as a quick glance at Genero Business Development Language limitations in mobile applications. Details can be found in the BDL reference topics.

The following language options have limited support:

- The `RUN` instruction has limited support on mobile platforms.
  - The `RUN` instruction is not supported on mobile devices, because of operating system limitations.
  - `RUN` *`command`* `WITHOUT WAITING` is not supported when programs run on an application server and display on a mobile device, because the Genero GUI protocol is not able to handle multiple connections at the same time.

The following language features are not supported.

- The `INPUT ARRAY` instruction is not supported.
- The `base.Channel.openPipe` method is not supported.

Environment variables on mobile

You may need to set environment variables for your app.

Set environment variables

Set environment variables for your app must be done in an `fglprofile` file. This fglprofile file must be located beside the main program module.

To add an environment variable for your mobile app, use the following syntax:

```
mobile.environment.DBFORMAT="$:,:.:"
```

Any existing environment variable setting is overwritten by the value set (using `mobile.environment.`*`envvarname`*) in the `fglprofile` file.

For more details, see Setting environment variables in FGLPROFILE (mobile) on page 170.

> **Note:** Environment variables set in an FGLPROFILE file are only read when the deployed application runs the mobile device. They are not read during development mode (i.e. when the VM runs on the development machine and the mobile client displays on the device). The FGLPROFILE environment variable settings are only for the VM component and are ignored by the GMA/GMI front-end component.

App localization

Mobile apps can be designed to display localized texts according to the current language selected on the device.

Localized string files (`.42s`) must be deployed in directories matching the language identifiers (`en` for English, `zh_TW` for simplified chinese, etc), beside the program module.

The list of `.42s` files required by the application must be defined in the unique `fglprofile` configuration file located beside the program module of your application.

For more details, see Localized string files on mobile devices on page 333 and Deploying mobile apps on page 2572.

Apps user interface

This section includes topics about user interface programming for mobile.

In general, the user interface of a mobile app written in Genero displays and reacts as a desktop or web application, while simultaneously respecting the device operating system look-and-feel. There are parts of the interface, however, that display and react in a specific way.

Take a look at each of the user interface items in this chapter, to understand how they are portrayed in a mobile app. A user interface feature not listed means there is nothing mobile-specific to its display or behavior.

Action rendering

How actions are rendered varies between OS type of the mobile device.

The top and/or bottom parts of the mobile app screen is dedicated to displaying default action views to the user. A default action view is an implicit graphical item that can be tapped to fire the corresponding action.

The default action views are rendered on the mobile device according to platform-specific standards, which are covered in Rendering default action views on mobile on page 1279.

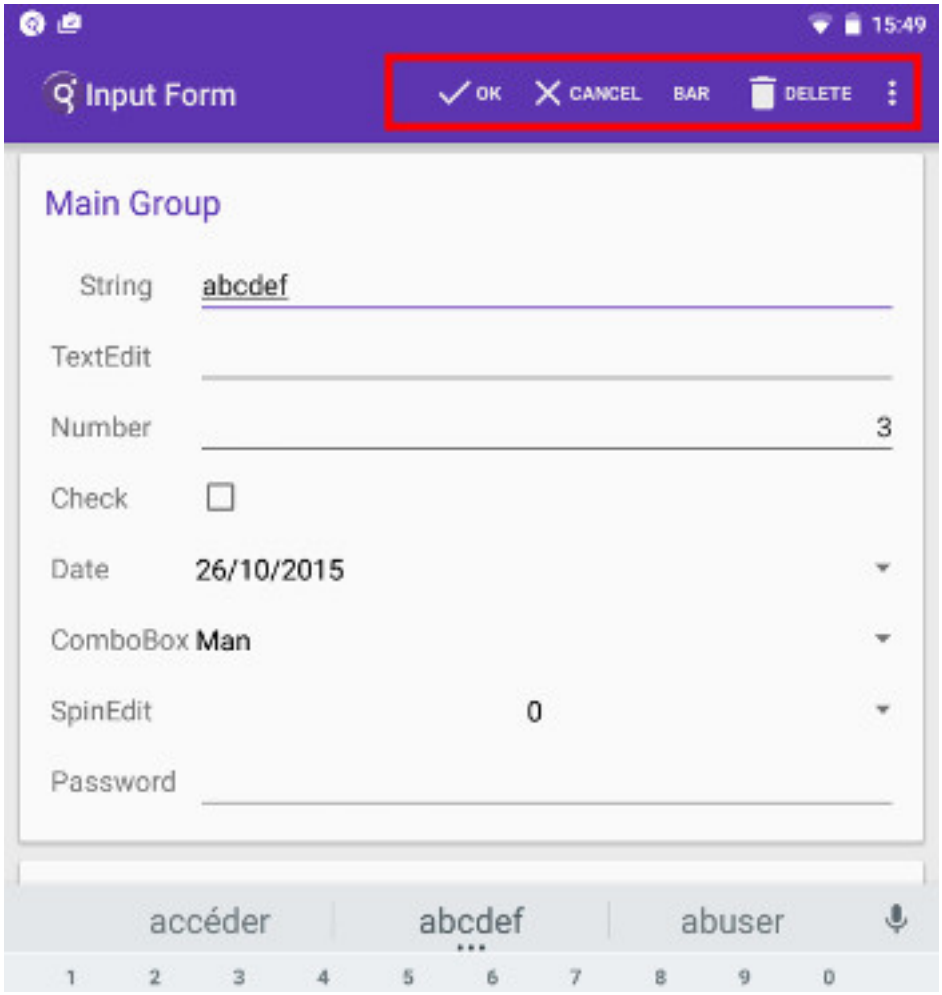


Figure 118: Action rendering example on an Android device

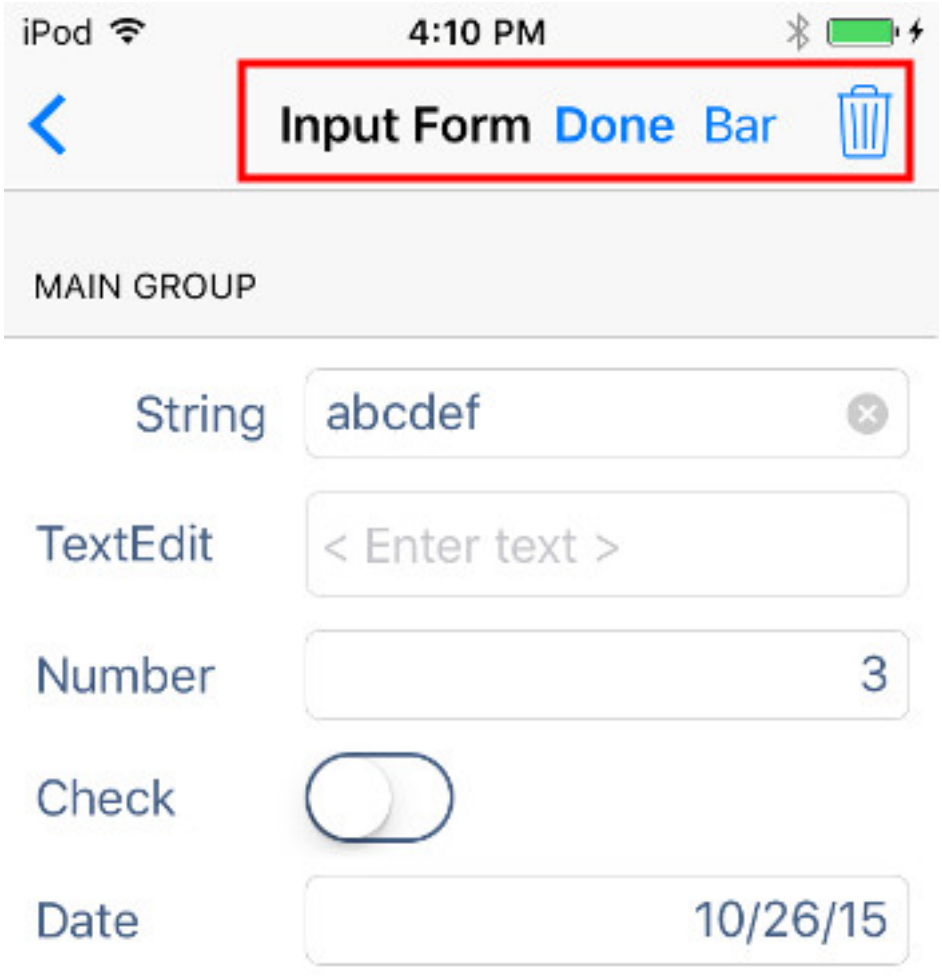


Figure 119: Action rendering example on an iOS device

Images and icons

For this topic, an image can refer to the `IMAGE` item type or the icons used in the app.

Image format support

Mobile apps written in Genero supports all image formats supported by the device OS, however each platform has its own restrictions on which image formats it supports:

- Android image format support
- iOS image format support

Mobile devices have a much higher pixel density (a higher resolution) than classic desktop monitors. An image which looks nice on a desktop can appear small or as an upscaled image on a mobile device.

Providing the image resource

Genero supports different solutions to provide the image data in a mobile app, depending on the need (button icon, application picture, etc). To understand how to get image resource on mobile apps with Genero, see Providing the image resource on page 784.

Image sizing on mobile devices

The `IMAGE` item type defines an area for the display of an image on a form.

Image layout and sizing can be controlled with form item attributes to adapt to the type of mobile device.

For more details, see Controlling the image layout on page 783.

Default action icons

In general, you want the icons used for your mobile app to be the standard icons used by all apps for the mobile platform. Genero is set up to use such icons by default. For more details, see Rendering default action views on mobile on page 1279.

Genero also supports icon centralization based on TTF icons, to get a global consistent look and feel for all your mobile apps. For more details, see Providing the image resource on page 784.

Keyboard type

Depending on the data being entered, a mobile device should display the keyboard that is appropriate for the data.

There are a variety of keyboard types for mobile devices. A field dedicated to phone number input should display a keyboard easying phone number input.

The `KEYBOARDHINT` form field attribute provides a hint regarding the kind of data the form field contains. Valid values include `DEFAULT`, `EMAIL`, `NUMBER`, and `PHONE`:

```
ATTRIBUTES
EDIT f01 = customer.cust_phone, KEYBOARDHINT = PHONE;
...
```

Although Genero mostly respects the provided hint, the variable data type that is bound to the form field is also examined, to determine what keyboard to display:

- If the field is defined as a `DATE` or `DATETIME` field, the date picker displays regardless of the `KEYBOARDHINT` setting.
- If the field is a `TEXT` data, a text keyboard displays regardless of the `KEYBOARDHINT` setting.

For more details, see KEYBOARDHINT attribute on page 973.

List views

Form tables in a mobile app render as list views.

List views are commonly used in mobile apps to present an indexed list of items or selectable list of options. They are also used to let users navigate through hierarchically structured data.

List views are displayed as either *full list views* or *embedded list views*.

The list view only displays the first two columns' content and any associated row image, regardless of the number of columns defined.

No column header/title is displayed in mobile list views. Thus the mobile user cannot manipulate columns (hide, reorder, resize, or sort).

With full list views, the built-in reduce filter allows the user to filter the rows displayed.

The `JUSTIFY` attribute of the second column can influence how the rows are displayed.

Various options affect the rendering and behavior, by defining `TABLE` container attributes, `DISPLAY ARRAY` dialog attributes and `ON ACTION` handler attributes.

For complete details on implementing table views in a mobile app, see Using tables on mobile devices on page 1362.

Split views

Split views refer to the ability to access two forms side by side on a mobile device.

Side by side views on mobile apps

Many mobile apps offer a specific form layout, splitting the screen in two in order to shows a list of the left side and a detail form on the right side. Such kind of layout can be implemented in Genero with the Split views on page 1395.

Differences in how split views are handled by the clients

There are differences between the Genero Mobile for iOS (GMI) and Genero Mobile for Android (GMA) implementations of split views and parallel dialogs, to include:

- When the application displays in a single pane or in two panes.
- How a user switches between the two panes.

There are also differences in how the split view renders between GMA and GMI.

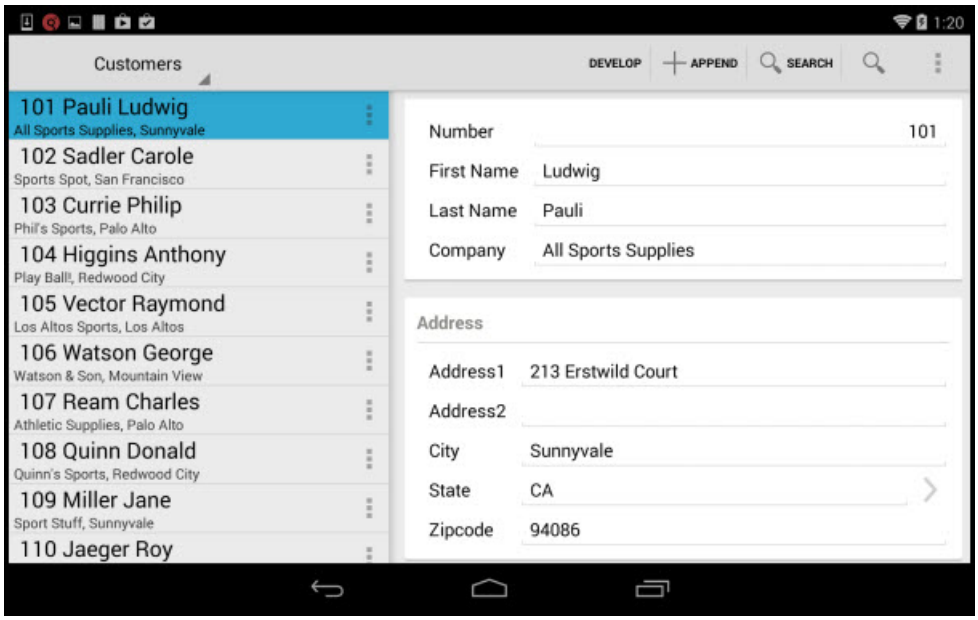
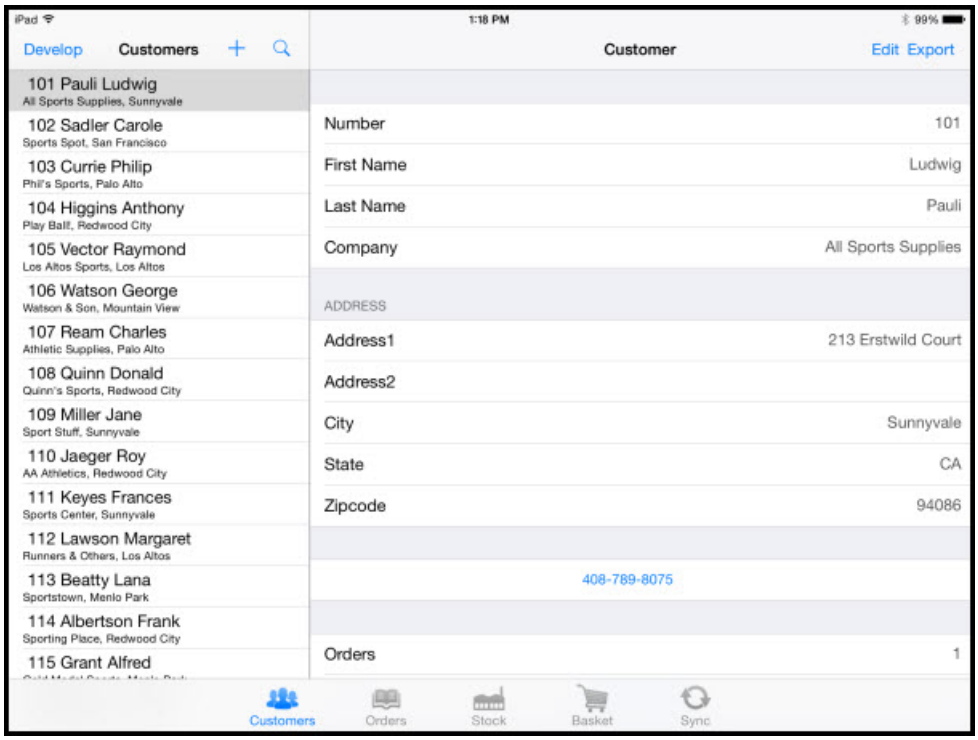


Figure 120: The stores2 demo rendered on an Android device

With Genero Mobile for Android:

- The navigator pane renders as a menu in the left-hand side of the title bar.
- All buttons are merged.
- The title is not displayed when there is a navigator pane. If there is no navigator pane, the title of the current window is displayed.



With Genero Mobile for iOS:

- The navigator pane renders along the bottom of the app.
- Each window has its own title and its own buttons.

Figure 121: The stores2 demo rendered on an iOS device

Toolbars

Toolbars allow to control over where actions display (and in what order).

For desktop applications, the toolbar is a series of buttons typically contained in a toolbar object, located at the top of the form. For Genero mobile apps, the toolbars are rendering according to the mobile platform standards.

Table 570: Mobile platform differences for toolbars

GMA	GMI
Genero Mobile for Android does not have toolbars. The action views appear in the Android **action bar**. Toolbar action views are listed first and ordered as they are defined in the toolbar, followed by the action views from the action panel. Disabled actions are greyed out.	The toolbar items render in the iOS toolbar pane. The `iosSeparatorStretch` toolbar style attribute can be used to stretch the separators to give more space between actions. Disabled actions are greyed out.

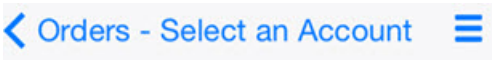
For more information, see Toolbars on mobile devices on page 1025. Default action views are rendered on mobile devices according to the rules covered in Rendering default action views on mobile on page 1279.

Topmenus

Topmenus provide a hierarchical menu in the app.

The rendering of a TOPMENU depends on the mobile operating system.

Table 571: Genero Mobile and topmenus

GMA	GMI
 **Figure 122: Navigation drawer for Android** A **navigation drawer** is visible in the action bar if a topmenu is available to the displayed form. The navigation drawer is a panel to the left of the app icon (seen as three bars). Tap and the navigation drawer transitions in from the left edge of the screen and displays the app's main navigation actions. Restricted to two levels.	 **Figure 123: Menu button for iOS** A **menu button** is visible in the navigation bar if a topmenu is available to the displayed form. There are no restrictions on the number of levels.

For complete details on implementing top menus in your app, see Topmenus on page 1027.

Front call support

Genero Mobile provides front calls to interface with the device capabilities.

Use front calls to access mobile devices capabilities. For example, with the `mobile/takePhoto` front call, you can open the camera app of the device to take a picture.

> **Note:** In a classical Genero client/server configuration, a frontcall is a remote procedure call that involves a round-trip between the front-end and the server where the application executes. For a standalone mobile app, this does not cause any latency. For a server-side app (i.e. runOnServer), however, latency can result.

The details for each frontcall can be found in Built-in front calls on page 1881.

Color and theming

Mobile applications must follow the platform colors and theming.

User interface design on mobile devices

Genero BDL provides several ways to define colors and styles for a mobile app. This section introduces features that can be used to customize your mobile app and adapt to the target platform user interface design. As a general rule, avoid to use non-standard ergonomics and decorations, using defaults to let Genero render your forms according to the platform standards. For example, the GMA front-end will use Google material design on Android devices.

Defining TTF icon colors

By default, TTF icons get a the color of the platform theme. A default color can be defined for all TTF icons of a window with the `defaultTTFColor` style attribute. In order to define a color for a specific icon, add an #RGB color specification in your image to font glyph mapping file.

For more details, see Using a simple image name (centralized icons) on page 785

App color theme on Android

Android apps can be created with a specific color theme following the Google material design.

When building the APK with the gmabuildtool, you can specify the general app colors with the `--build-app-colors` option.

For more details, see Define app's color theme on page 2578.

Database support on mobile devices

On the device, a Genero app can use SQL for data management.

Databases supported on mobile devices

Only SQLite can be used on mobile devices. SQLite has a small footprint, is free and readily available.

The database driver (dbmsqt) and the SQLite library are built into the runtime system for mobile Genero apps. No database driver specification is required when running on mobile.

To read more about SQLite programming, see Using SQLite database in mobile apps on page 2567.

Synchronizing data with a central database

When local mobile app data needs to be synchronized with a central database, you must write your own synchronization routines using Web Services. You must implement a back-end service to collect mobile database updates and to send central database changes back to the mobile app.

> **Important:** If you are using GMI, and the device goes into standby mode, the application does not run in the background and activities with the network are suspended. If you are synchronizing data with the server, and the device goes into standby, the synchronization is suspended until the device resumes from standby. If you have a long synchronization, you need to either disable the sleep to allow the synchronization to complete, or accept that the synchronization will suspend when the device goes into standby mode.

Using SQLite database in mobile apps

On the device, Genero Mobile uses the SQLite database only.

Running an app in development mode

When running an app in development mode (where the app runs on a computer), you can use any database server that Genero supports for the operating system of the server-side app.

Running an app on a mobile device

When running the application on the device, only SQLite can be used. The database must be created at the first application execution, or it must be delivered as the default database in the `.ipa` or `.apk` package.

Locale character set and length semantics

SQLite stores data in UTF-8 codeset, mobile apps will by default run in UTF-8 and with character length semantics (`FGL_LENGTH_SEMANTICS=CHAR`).

Creating the database

Mobile applications usually create their database at first execution. The SQLite database file must be created in the application sandbox, in a writable directory. If the database file does not exist in the current working directory (`os.Path.pwd()`), create an empty file and then perform a `CONNECT TO` instruction.

For more details, see Creating a database from programs on page 414.

Providing a default database

SQLite database file format is cross-platform. Instead of creating the database the first time the application starts, you might want to prepare a default database file in your development environment, and include it in the `.ipa` / `.apk` package.

Data types with SQLite

SQLite does not have strict data type checking as traditional databases. If you define a table column as a `DECIMAL`, you can still store character values in that column. Pay attention to this SQLite specific feature, to avoid invalid storage and type conversion errors in your application.

Consider using the following data types for maximum portability, especially when data needs to be synchronized with a central database server, where the data types must match to the types used in the mobile application: `CHAR`, `VARCHAR`, `DATE`, `DATETIME YEAR TO MINUTE`, `DATETIME YEAR TO FRACTION(3)`, `DECIMAL`, `SMALLINT`, `INTEGER`, `BIGINT`, `BYTE`, `TEXT`.

Optimizing data changes

SQLite can be slow at doing commits, due to the data integrity technique used for transactions. Since each `INSERT` / `UPDATE` / `DELETE` statements acts as an individual transaction (i.e., auto-commit), there will be as many transactions/commits as data manipulation statements. For example, it takes about 10 seconds to insert 1000 rows on an Intel core i7 2.60GHz CPU / 5400.0 RPM HDD computer.

When executing code that modifies a lot of rows (for example, when inserting default data at first application execution, or when doing synchronization with a central database), enclose the SQL statements within a `BEGIN WORK` / `COMMIT WORK` transaction block to speed up the process:

```
BEGIN WORK
FOR i=1 TO mylog.getLength()
    -- INSERT / UPDATE / DELETE statements
END FOR
COMMIT WORK
```

Enforcing foreign key constraints

SQLite 3.6.19 and + support foreign key constraints, with `ON DELETE CASCADE` and `ON UPDATE CASCADE` options. By default, however, foreign key constraints are not enforced. Each application must explicitly turn on the feature with a `PRAGMA` command. Immediately after the database connection, you can perform the `PRAGMA` command in an `EXECUTE IMMEDIATE` statement:

```
CONNECT TO connstr AS "c1"
EXECUTE IMMEDIATE "PRAGMA foreign_keys = ON"
```

Truncating the SQLite database file

By default, when deleting rows, SQLite keeps the unused database file pages for future storage. As result, when deleting a large amount of data, the database file might be larger than necessary. Consider truncating the database file with the `VACUUM SQL` command (in an `EXECUTE IMMEDIATE` statement), if disk space is limited and when a lot of database rows were deleted.

Depending on the application, the `VACUUM` command can be executed:

- when starting the application, just after connecting to the database,
- after doing a large database operation (such as a synchronization with a central database),
- as a manual option that the user can trigger.

For example, after connecting to the database:

```
CONNECT TO connstr AS "c1"
```

```
EXECUTE IMMEDIATE "VACUUM"
```

Sharing database files between Android apps

Two different Android apps (each packaged as a separate `.apk`) execute in their own sandbox, but have access to the storage area (SD-CARD) and therefore could share a common database file.

SQLite handles concurrent access to the same database file by setting a lock on the entire db file when modifying data (`INSERT`/`UPDATE`/`DELETE`). By default, if a writer process locks the file, other processes must wait until the lock owner process completes its transaction and releases the lock.

Because of Informix compatibility, Genero BDL uses a default lock timeout or zero (i.e., not waiting for locks to be released). As result, when writing to a database file that is locked by another process, if the isolation level is `SERIALIZATION` (the default with SQLite), an application will get the SQL error -244.

To avoid this problem, you must change the default lock timeout with the `SET LOCK MODE` instruction, after starting the database session:

```
CONNECT TO connstr AS "c1"
SET LOCK MODE TO 5 -- seconds
```

The second process will then wait until the first process releases the lock. If transactions are short (milliseconds), having processes waiting for each other is transparent to the user.

Accessing device functions

Mobile apps can access device functions by using front calls.

Mobile applications typically want to acces device functions such as geolocation, multi-media content (photos, videos), messaging (contacts database, email, sms).

This can be easily achieved by using front calls dedicated to mobile features. Note that some functions are platform specific, for example to launch an Android activity, or access to iOS device settings.

As a general rule, execute your front call in a `TRY / CATCH` block to catch errors:

```
DEFINE status STRING,
       latitude, longitude FLOAT
TRY
    CALL ui.Interface.frontCall("mobile", "getGeolocation",
         [], [status, latitude, longitude] )
CATCH
    ERROR "Could not get coordinates..."
END TRY
```

For more details, see Genero Mobile common front calls on page 1925, Genero Mobile Android front calls on page 1940, Genero Mobile iOS front calls on page 1945.

Web Services on mobile devices

Web Services can be used within mobile applications.

Genero Mobile for Android

Requirements for Web Services on Android platforms:

- V3 SSL Certificates

For complete details about the requirements for Web Services on GMA, see GMA / Android Web Services requirements on page 2419

Genero Mobile for iOS

Requirements for Web Services on iOS platforms:

- Some `com` classes are not supported.
- Some methods of the `com.HTTPRequest` class have a different behavior on GMI.
- Some `xml` classes are not supported.
- For supported classes of the `xml` package, methods using an URL parameter accept only a file URI.
- SOAP errors and faults are not handled; an application may handle the error -15559.
- GWS configuration entries of FGLPROFILE are not supported.
- A long running HTTP request popup displays after some seconds, giving the user the option to cancel the request.
- HTTP request compression for POST/PUT is not supported.
- Multipart HTTP requests are not supported.
- Limited configuration of SOAP client.

For complete details about limitations for Web Services on GMI, see GMI / iOS Web Services limitations on page 2417.

Debugging a mobile app

Different solutions are available to debug a mobile app.

Debugging a mobile app in development mode

When executing a mobile app program on a server, displaying the user interface on a mobile front-end defined by FGLSERVER, it is possible to debug the BDL code with the `fglrun -d` option:

```
$ export FGLSERVER=device-ip-address
$ fglrun -d main.42m
```

For more details, see Starting fglrun in debug mode on page 1532.

AUI protocol debugging

With app running on a server or on the device, it is possible to show AUI protocol exchanges in the console running the program on the server, by setting the FGLGUIDEBUG envrionment variable to 1. When this variable set, you can watch user interface events that occur during program execution and how they are treated by the runtime system.

To set the FGLGUIDEBUG environment variable for an app running on the device, use an FGLPROFILE `fglrun.environment` entry. The output can be inspected with the program logs as described later in this section.

For more details, see FGLGUIDEBUG on page 181.

AUI protocol logging in development mode

With app running on a server, it can be useful to log AUI protocol exchanges betweem the runtime system and the mobile front-end, to inspect the content, or replay a scenario. This is possible with the `--start-guilog` and `--run-guilog` options of fglrun:

```
$ fglrun --start-guilog=case1.log
```

The AUI protocol log file produced by the `--start-guilog` option can then be shared for analysis.

For more details, see Front-end protocol logging on page 759.

Debugging a mobile app running on the device

When executing the mobile app on a device, and if the app has been created with debug mode, it is possible to establish a connection to the runtime system executing on the mobile device, by using the `fgldb` command line tool.

> **Important:** On iOS devices, after installing the app, you need to enable the debug port in the app settings, otherwise the app will not listen to the debug port.

For example:

```
$ fgldb -m 192.168.1.23:6400
108     DISPLAY ARRAY contlist TO sr.*
(fgldb)
```

This way you can debug an app running on a device, by using the source code located on the server where the `fgldb` command is executed.

For more details, see Debugging on a mobile device on page 1534.

Building mobile apps in debug mode

In order to enable debug features of an app running on a mobile device, you need to build the app in debug mode:

- For Android:

  The `gmabuildtool` provides the `--mode debug` option, to create a debug version of the APK.

  For more details, see Building Android apps with Genero on page 2574.
- For iOS:

  The `gmibuildtool` provides the `--mode debug` option, to create a debug version of the IPA. The certificate defined in the provisioning profile must be a development certificate.

  > **Note:** After installing the debug version of the app on your iOS device, you need to enable the debug port in the app settings.

  For more details, see Building iOS apps with Genero on page 2586.

Browse the AUI tree created on the mobile front-end side

The content of the Abstract User Interface tree created on the mobile front-end side can be inspected from a web browser, when the app has been created with debug mode, or in development mode by executing the app on a server and displaying on the device.

To inspect AUI tree, open a web browser and enter the following URL:

```
http://device-ip-address:6480 (or 6400)
```

For more details, see Inspecting the AUI tree of a front end on page 751.

Viewing embedded app program logs

The program logs of an app running on a device can be viewed in a browser, if the app was created in debug mode. VM messages (runtime errors, standard output and standard error) are available. This feature is not availbale if the app is build in release mode.

To inspect program logs, open a web browser and enter the following URL:

```
http://device-ip-address:6480 (or 6400)
```

A menu will then appear in the web page, where you can choose the VM output to be inspected.

Deploying mobile apps

This section describes how to build and deploy mobile apps with Genero.

- Deploying mobile apps on Android devices on page 2572
- Deploying mobile apps on iOS devices on page 2584
- Running mobile apps on an application server on page 2595

Deploying mobile apps on Android™ devices

This section contains information to create a mobile application to be deployed on Android devices.

Directory structure for GMA apps

Platform-specific rules need to be considered when deploying on Android devices (GMA).

The application sandbox

On Android devices, applications are deployed in an application sandbox. The application can access and store data outside of its space, but then the data is also accessible by the other applications.

Directory structure for a GMA application

Inside its application sandbox, an Android app uses the following directory structure:

```
appdir/
|-- main.42m                              --
|-- *.42m                                   |
|-- *.42f                                   |
|-- fglprofile                              |
|   ...                                     |
|-- *.42s                                   |
|-- de/                                     | Program files
|   |-- *.42s                               |
|-- fr/                                     |
|   |-- *.42s                               |
|-- zh/                                     |
|   |-- *.42s                               |
|   ...                                     |
|-- ... other resource files/dirs ...       |
|   ...                                     |
|-- webcomponents                           |
|   |-- component-type                      |
|       |-- component-type.html             |
|       |-- other-web-comp-resource         |
|   ...                                    --

|-- appdata/
    |-- ... writable app files ...

tmpdir/
|-- ... temporary files ...
```

Program files directory (*appdir*)

Application program files (`.42m`, `.42f`, and so on) need to be deployed in the *`appdir`* application base directory.

The program files directory can be found in programs with the `base.Application.getProgramDir` on `page 1705` method.

> **Important:** On Android, the program files directory returned by the `base.Application.getProgramDir()` method is the same directory as the default working directory, returned by `os.Path.pwd()`.

The FGLAPPDIR environment variable is automatically set to the *`appdir`* directory.

Program name (MAIN)

When deploying on mobile devices, the name of the program file must be `main.42m` or `main.42r`.

> **Note:** When using the command-line app build scripts, the name of the program file must be main.42?. When using Genero Studio, the packaging script takes care of renaming this file, if you have not named it `main`.

As with other program files, the "MAIN" module must be located under the *`appdir`* application program directory.

Working directory

On Android devices, the default current working directory is the *`appdir`* directory, and can be used for writable files.

The current working directory can be found in programs with the `os.Path.pwd` on `page 2004` method.

Files that need to be writable (such as SQLite database files) can be created directly under the *`appdir`* directory. However, to better organize application files, create sub-directories such as *`appdir/appdata`*, keeping original files directly under the *`appdir`* directory. For example, create the application database under `os.Path.pwd() || "/database"`.

Temporary directory (*`tmpdir`*)

A temporary directory is available for the application.

In order to find the temporary directory for the app, use the standard.feInfo front call, with the "`dataDirectory`" parameter.

To create a temporary file name, use the `os.Path.makeTempName()` method.

Language directories for localized strings

When the app starts, the appropriate `.42s` string files will be loaded from the directory corresponding to the current language settings of the mobile device. String files to be loaded can be defined in app's `fglprofile`, or you can use the main program name to avoid fglprofile settings.

For each language supported by your application, a directory must exist under *`appdir`*, with a name including the locale codes. Consider also providing default string files (in English for ex) directly under *appdir*, in case if the regional settings of the device do not match one of the locale directories of the app, otherwise the application will stop with error -8006.

For example:

```
appdir/mystrings.42s
appdir/fr/mystrings.42s
appdir/de/mystrings.42s
```

For more details, see Localized string files on mobile devices on page 333.

Deploying a custom fglprofile file

If you need to set fglprofile entries for your mobile application, create a file with the name `fglprofile`, and deploy it under the *appdir* directory, along with the other program files.

See Understanding FGLPROFILE on page 164 for more details about fglprofile settings.

Creating the initial database file

When a mobile application starts for the first time, it typically creates a new database, or copies a existing database template file from the file directory (`base.Application.getProgramDir on page 1705`) to the working directory (`os.Path.pwd on page 2004`).

> **Note:** Different database file names should be used for the original and final application database, as folders pointed by `base.Application.getProgramDir()` and `os.Path.pwd()` can be the same on Android devices.

For more details about database creation on mobile devices, see Creating a database from programs on page 414.

Building Android™ apps with Genero

Genero provides a command-line tool to create applications for Android devices.

Basics

Genero mobile apps for Android are distributed as APK packages like any other Android app. Genero provides a command line tool to build the APK package for your mobile application. For testing purposes, the tool can also deploy and automatically launch the app on a specific device or simulator. The tool has also an option to update the Android SDK.

> **Note:** This documentation section implies that you are familiar with Android app programming concepts and requirements. For example, you will need the Android SDK tools to be installed (and up to date) to build your Android apps. For more details, visit the Android developer site at https://developer.android.com.

Prerequisites

Before starting the command line tool to build or deploy the app, fulfill the following prerequisites:

- The Genero BDL development environment (FGLDIR) must be installed on the computer to compile your program files.
- The Java JDK must be installed. The minimum required version is 1.7.
- The Android SDK must be installed (the buildtool uses the "Gradle" utility).

  > **Note:** The first time the Android tools are called, they will automatically check for updates. Therefore, you need an internet connection.
- All Android SDK packages required by GMA must be downloaded. In order to download the required Android SDK packages, execute the `gmabuildtool updatesdk` command.

  > **Note:** Execute the `gmabuildtool updatesdk` command every time a new version of the GMA buildtool and GMA binary archive is installed.
- The GMA buildtool and the GMA binary archive must be installed.

  The GMA buildtool and GMA binary archive are provided in the GMA distribution archive (`fjs-gma-*.zip`).

  To setup the GMA buildtool perform the following steps:

  1. Create a dedicated directory (*gma-install-dir*) and extract the content of the `fjs-gma-*.zip`. This will contain the `gmabuildtool` command. Add the *gma-install-dir* directory to your PATH environment variable.

2. Create a directory (*gma-scaffold-project*) for the GMA binary archive, and extract *`gma-install-dir`*`/artifacts/fjs-gma-*-android-scaffolding.zip` into this directory. This direcory will be specified with the `--build-project` option of `gmabuildtool`.

- Android specific app resources such as icons (in all required sizes) are required, along with the application program files.
- If you plan to publish your app on Google Play, register to Google Play as a developer and create a Google Play project.

Environment settings

Define the following environment variables before starting the command-line buildtool:

- Android SDK env settings (ANDROID_HOME, PATH)
- Java JDK env settings (JAVA_HOME, PATH)

Update the Android SDK with the GMA buildtool

After a fresh installation of the GMA buildtool and GMA binary archive, upgrade the Android SDK and downnload all Android SDK packages required by GMA, by executing the `gmabuildtool updatesdk` command:

```
gmabuildtool updatesdk
    --android-sdk /use/local/32bits/android-sdk/r22.6.2
```

The Android SDK installation directory is required for the SDK update, and is found in ANDROID_HOME environment variable, or with the `--android-sdk` option.

If you need to specify a proxy to download the Android SDK, use the `--proxy-host` and `--proxy-port` options:

```
gmabuildtool updatesdk
    --proxy-host amadeus --proxy-port 3232
    ...
```

Building and deploying with the GMA buildtool

The `gmabuildtool build ...` command creates the APK from a set of files, and according to the options passed as parameter.

```
gmabuildtool build
    ... build options ...
```

Once the APK file is created, use the `gmabuildtool test --test-apk` command to install the app on the Android device plugged to the computer, and start the app automatically.

```
gmabuildtool test
    --test-apk path-to-the-apk-file
```

For a complete description of command options, see gmabuildtool on page 2580.

Cleaning the scaffold files

The build process is optimized to avoid a complete APK rebuild every time you invoke the GMA buildtool: When application program file changes are detected, the GMA buildtool will create archive files that can be reused in the next build if no changes are detected. However, files used for the optimized build might be corrupted, for example in case of user interruption or graddle build failure.

In this situation, you can use the `--clean` option of the `gmabuildtool build ...` command, to cleanup the scaffold build directory, and continue with a fresh build:

```
gmabuildtool build --clean
    ... build options ...
```

Using an options file

To simplify option speciciation, create an file with the list of options to be passed to the `gmabuildtool` with the `--input-options` argument. The options file must contain a line for each option/value peer:

```
$ cat myoptions.txt
--build-output-apk-name MyApp
--build-app-name MyApp
--build-app-package-name com.example.myapp
...
$ gmabuildtool --input-options ./myoptions.txt
```

Elements used to building the Android app

The `gmabuildtool build` command builds the Android APK package from the following:

- The GMA binary archive, containing the GMA front end and the FGL runtime system.

  Note: You must unzip the `fjs-gma-*-android-scaffolding.zip` file.

- The compiled application program and resource files (`.42m`, `.42f`, etc) (`--build-app-genero-program*` options),
- The prefix for the APK file name to be generated (`--build-output-apk-name` option),
- The name of the app (`--build-app-name` option),
- The version code of the app (`--build-app-version-code` option),
- The version name of the app (`--build-app-version-name` option),
- Android app specific resources:
  - Android app icons (all sizes) (`--build-app-icon*` options).
- Android app specifics (to sign the app, not required in development mode):
  - The keystore alias, used with the `keytool` to generate the keystore file (`--build-jarsigner-alias` option).
  - The keystore file, generated from `keytool` (for the `--build-jarsigner-keystore` option).

Generate the keystore file to sign your app

In order to build an APK that can be deployed on the market (Google Play), you need to sign your Android app.

First, you need to generate a keystore file with the `keytool` Android utility.

The keystore file and keystore alias will be used by the `gmabuildtool` to sign the APK with the `jarsigner` utility. These signing credentials are passed to the buildtool with the `--build-jarsigner-keystore` and `--build-jarsigner-alias` options.

For more details, see manual Android application signing.

Generated APK file name

The file name of the APK package is formed from:

1. the APK file name prefix defined by the `--build-output-apk-name` option (by default, "app"),
2. the target type (`-arm` or `-x86`),

3. if building a debug version, the `-debug` suffix,
4. the `.apk` file extension.

For example, if the APK file name prefix is `MyApp` and the target architecture is `arm` in debug mode, the resulting APK file name will be: `MyApp-arm-debug.apk`.

Default build directory structure

For convenience, the buildtool supports a default directory structure to find all files required to build the APK:

```
top-dir
|
|-- main.42m and other program files, as described in Directory structure
 for GMA apps on page 2572
|
|-- gma
|   |-- project
|   |   ...
|   |-- temp
|   |   ...
|   |-- ic_app_hdpi.png
|   |-- ic_app_mdpi.png
|   |-- ic_app_xhdpi.png
|   |-- ic_app_xxhdpi.png
|   |   ...
```

In the above directory structure:

1. *`top-dir`* is the top directory of the default structure. It will typically hold your application program files. The program files directory can be specified with the `--build-app-genero` option.
2. *`top-dir`*`/gma` is the default directory containing the GMA binary archive, the temp directory and the app icons.
3. *`top-dir`*`/gma/project` must contain the unzipped GMA binary archive (`fjs-gma-*-android-scaffolding.zip`). This directory can be specified with the `--build-project` option.

Android permissions

In order to use a device feature such as the camera, an Android app must be created by specifying the corresponding Android permissions. Furthermore, Android distinguishes "normal" and "dangerous" permissions. While both type of permissions just need to be specified when building the app, "dangerous" permissions require a user validation: A popup dialog will appear to let the user confirm that the dangerous feature can be accessed. Before Android 6, dangerous permissions defined by the app were asked at app installation. Starting with Android 6, dangerous permissions must be asked by the app code on demand.

Android permissions required for the built-in front calls are automatically set by GMA, which ask automatically user confirmation if the permission is dangerous. For example, if the app code makes a `chooseContact` front call, the GMA will automatically ask the user for the Android permission to access the contacts database, and set the corresponding permission on confirmation. When building your app, there is no need to specify permissions required for built-in front calls.

Other permissions (not involved by built-in front calls) need to be defined when building the app, and "dangerous" permissions need to be asked to the user when needed. In order to ask the user for a given permission, the app must use the `askForPermission` front call.

Android permissions can be specified with the `--build-app-permissions` option of the `gmabuildtool`. Define the list of permissions as a single argument, by using the comma as separator.

For example:

```
gmabuildtool build \
```

```
    ...
    --build-app-permissions android.permission.READ_CALENDAR,... \
    ...
```

Android permissions listed below are defined by default by GMA and therefore do not need to be specified when building your app. For "dangerous" permissions, the GMA will automatically ask the user to access the feature, when corresponding front call is performed:

- Normal permissions set by default in GMA (no user confirmation required):
  - `android.permission.INTERNET`
  - `android.permission.ACCESS_NETWORK_STATE`
  - `android.permission.CHANGE_NETWORK_STATE`
  - `android.permission.ACCESS_WIFI_STATE`
  - `android.permission.WAKE_LOCK`
  - `com.google.android.c2dm.permission.RECEIVE`
  - `packageName.permission.C2D_MESSAGE`
- Dangerous permissions set by default in GMA (requires user confirmation):
  - `android.permission.ACCESS_FINE_LOCATION`
  - `android.permission.ACCESS_COARSE_LOCATION`
  - `android.permission.READ_CONTACTS`
  - `android.permission.GET_ACCOUNTS`
  - `android.permission.MOUNT_FORMAT_FILESYSTEMS`
  - `android.permission.READ_LOGS`
  - `android.permission.READ_PHONE_STATE`
  - `android.permission.WRITE_EXTERNAL_STORAGE`
  - `android.permission.READ_EXTERNAL_STORAGE`

Other permissions required by the app but not listed here need to be specified when building your app, and if the permission enters in the "dangerous" category, the app code must issue an `askForPermission` front call before using the feature. For a complete list of Android permissions, see Android's Manifest permissions.

Define app's color theme

Android apps can be created with a color theme defined by four basic colors to customize your app, that can be defined with the `--build-app-colors` option.

> **Note:** This feature is only available with Android 5.0 / SDK 21 and higher. With older versions of Android, the colors specified with the `--build-app-colors` option will not take effect.

The value provided to the `--build-app-colors` option must be a comma-separated list of four hexadecimal RGB colors.

The position of the color defines its purpose:

1. Primary color: This is the main color used in the app.
2. Primary dark color: This is the color used for the status bar and the navigation bar.
3. Accent color: This is the color used for widgets and table lines.
4. Action bar text color: This is the foreground color for the texts in the action bar.

By default, the color theme is the Genero purple color.

For example, to define a red color theme, use the following combination:

```
gmabuildtool build \
    ...
    --build-app-colors "#F44336,#B71C1C,#EF9A9A,#FFFFFF" \
```

```
    ...
```

For more details about Android color shemes, see Android Colors

Debug and release versions

Android apps can be generated in a debug or release version. Release version are prepared for distribution on Google Play, while debug versions are used in development. In debug mode, the app installed on the device will listen on the debug TCP port to allow `fgldb -m` connections.

Debug or release mode can be controlled with the `--build-mode` option of the `gmabuildtool` command:

```
gmabuildtool build \
    --build-mode debug \
    ...
```

By default the app is build in release mode.

Building an Android app with gmabuildtool

Follow the next steps to setup a GMA app build directory in order to create an Android app, based on the default directory structure:

1. Create the root distribution directory (*`top-dir`*)
2. Copy compiled program files (`.42m`, `.42f`, `fglprofile`, application images, web component files, etc) under *`top-dir`*.
3. Copy the default English `.42s` compiled string resource file under *`top-dir`*.
4. Create non-English langage directories (fr, ge, ...) under *`top-dir`* and copie the corresponding `.42s` files.
5. Copy default application data files (database file for ex) under *`top-dir`*.
6. Create the *`top-dir`*`/gma` directory.
7. Copy Android app resources (icons) under *`top-dir`*`/gma`.

Once the build directory is prepared, issue the following commands to build the APK:

```
$ cd top-dir
$ gmabuildtool build \
    --android-sdk /home/mike/android/sdk \
    --build-project /home/mike/work/example/scaffold_project \
    --build-apk-outputs /home/mike/work/example/outputs \
    --build-output-apk-name MyApp \
    --build-app-name MyApp \
    --build-app-package-name com.example.myapp \
    --build-app-version-code 1002 \
    --build-app-version-name "10.02" \
    --build-jarsigner-alias android_alias \
    --build-jarsigner-keystore /home/mike/work/example/sign/android.keystore
 \
    --build-mode release \
    --build-app-permissions
 android.permission.ACCESS_WIFI_STATE,android.permission.CALL_PHONE
```

> **Important:** The directory specified with the `--build-project` option must contain the unzipped GMA binary archive (`fjs-gma-*-android-scaffolding.zip`).

Building an app with GMA custom extensions

The `gmabuildtool build` command supports APK creation for applications using GMA custom extensions written in Java.

Before building the APK package, create the custom GMA binary archive with your extensions, as described in Packaging custom Java extensions for GMA on page 1590.

When your custom GMA binary archive is complete, build the APK package with the `gmabuildtool build` command. Use the `--build-project` option to specify the path to the Android Studio project that was used to build your custom GMA binary archive:

```
$ gmabuildtool build
    ...
    --build-project /home/mike/android_project/mycustgma
    ...
```

> **Note:** Other options have to be specified as for a regular build using the original standard GMI binary archive.

Deploy and launch the app

After building the APK package, for testing purposes, you can deploy and launch your app from the command line with the `gmabuildtool test` command.

> **Note:** The `test` command is provided for development only. To deploy your app in production for several devices, use the regular publication channel of Android apps.

In order to deploy and launch the app, you must provide:

1. the path to the APK file

There must be only one Android device connected or running Android emulator.

```
$ gmabuildtool test \
     --test-apk /home/mike/work/example/outputs/MyApp-arm-debug.apk
```

gmabuildtool

The gmabuildtool is a utility to create and test applications for an Android devices.

Syntax

```
gmabuildtool { build | test | updatesdk } [options]
```

1. `build` is the command to build an APK package.
2. `test` is the command to deploy and launch an app.
3. `updatesdk` is the command to update the Android SDK to download packages required by GMA.
4. *options* are described in Table 572: gmabuildtool options on page 2580.

Options

Table 572: gmabuildtool options

Option	Short option	Description
`--android-sdk` *path*	`-as`	The path to the Android SDK installation directory. If not specified, defaults to the ANDROID_HOME environment variable.
`--build-app-colors` *color-list*	`-bc`	Define the Android color theme for the app (Android 5.0+ / SDK 21+)

Option	Short option	Description
		The value must be a comma-separated list of four hexadecimal RGB colors. The position of the color defines its purpose: **1.** Primary color: This is the main color used in the app. **2.** Primary dark color: This is the color used for the status bar and the navigation bar. **3.** Accent color: This is the color used for widgets and table lines. **4.** Action bar text color: This is the foreground color for the texts in the action bar. By default, the color theme is the Genero purple color.
`--build-app-genero-program-main` *path*	`-bgpm`	Relative path to the main module of the application (can be .xcf, .42m or .42r). Defaults to `main.42m`
`--build-app-genero-program` *path*	`-bgp`	Defines the path to the application program files (.42m, .42f, etc) The contents of this directory will be zipped and bundled inside APKs. This option can handle an already zipped Genero program archive. If not specified, defaults to the current working directory.
`--build-app-icon-hdpi` *path*	`-bih`	Defines the path to application icon in hdpi. Default is `./gma/ic_app_hdpi.png`, in the current working directory.
`--build-app-icon-mdpi` *path*	`-bim`	Defines the path to application icon in mdpi. Default is `./gma/ic_app_mdpi.png`, in the current working directory.
`--build-app-icon-xhdpi` *path*	`-bixh`	Defines the path to application icon in xhdpi. Default is `./gma/ic_app_xhdpi.png`, in the current working directory.
`--build-app-icon-xxhdpi` *path*	`-bixxh`	Defines the path to application icon in xxhdpi. Default is `./gma/ic_app_xxhdpi.png`, in the current working directory.
`--build-app-name` *app-name*	`-bn`	Application name. If not specified, the application name defaults to the current working directory.
`--build-app-package-name` *name*	`-bpn`	APK package name. The package name should be formatted as "`com.`*`organization-name`*`.`*`app-name`*".

Option	Short option	Description
		If not specified, the application package name defaults to `com.example.`*`current-working-directory`*
`--build-app-permissions` *`permissions`*	`-ba`	Android application permissions. The list of permissions is provided as a comma separated list of `android.permission.*` identifiers. For more details, see Android permissions on page 2577.
`--build-app-version-code` *`version-code`*	`-bvc`	Application version code. For example: 100915 The value of this option must be an integer (do not use decimal numbers).
`--build-app-version-name` *`version-name`*	`-bvn`	Application version name. For example: 10.09.15 This will be the actual app version visible on devices.
`--build-apk-outputs` *`path`*	`-bo`	Defines the destination folder where the APK packages must be created.
`--build-distribution` *`path`*	`-bd`	Distribution folder path. Used to have a location to store the extracted scaffold folder, and be able to build GMA APKs if the project folder is an extension project. Default is `./gma/temp`, in the current working directory.
`--build-jarsigner-alias` *`alias`*	`-bja`	Jarsigner alias. This is the alias provided to the `keystore` utility to build the keystore file to sign the app. Used when APK artifacts are signed.
`--build-jarsigner-keypass` *`keypass`*	`-bjk`	Jarsigner keypass. Specifies the password used to protect the private key of the keystore entry addressed by the alias specified in the --build-jarsigner-alias option. The password is required when using jarsigner to sign a JAR file. Used when APK artifacts are signed.
`--build-jarsigner-keystore` *`path`*	`-bjks`	Jarsigner keystore path. This is the path to the keystore file generated by the `keystore` utility to sign the app. Used when APK artifacts are signed.
`--build-jarsigner-storepass` *`storepass`*	`-bjs`	Jarsigner storepass.

Option	Short option	Description
		Specifies the password that is required to access the keystore. Used when APK artifacts are signed.
`--build-mode {release|debug}`	`-bm`	Package build mode, to build a release version or a development/debug version. Default: `release`
`--build-output-apk-name` *name*	`-ban`	Defines the prefix for the APK packages names. By default, this prefix is "app". The file name of the APK package is formed from: **1.** the APK file name prefix defined by the `--build-output-apk-name` option (by default, "app"), **2.** the target type (`-arm` or `-x86`), **3.** if building a debug version, the `-debug` suffix, **4.** the `.apk` file extension. For example, if the APK file name prefix is `MyApp` and the target architecture is `arm` in debug mode, the resulting APK file name will be: `MyApp-arm-debug.apk`.
`--build-project` *path*	`-bp`	Defines the path to the directory containing the original (unzipped) GMA binary archive files, or the directory containing the Android Studio project, when building a customized GMA. **Note:** When using the original GMA binary archive, the zip file must be uncompressed before executing the build. Default is `./gma/project`, in the current working directory.
`--build-types {x86|arm|x86,arm}`	`-bt`	Target platform type. This option accepts a list of platform types separated by a comma (x86,arm) If not specified, both x86 and arm APKs will be generated.
`--clean`	`-c`	Clean the scaffold build directory before a rebuild. This option can be used with the `build` command, to cleanup the scaffold directories containing the application files, before a new build. To be used in case if the previous build was interrupted or has failed.
`--input-options` *path*	`-i`	Path to the file containing gmabuildtool options.

Option	Short option	Description
		Define all options in a file and pass the file to the gmabuildtool command with the `--input-options` argument. The options file must use the following format: *option-name option-value* ...
`--java-home` *path*	`-jh`	Java home path. Default is JAVA_HOME.
`--proxy-host` *host*	`-ph`	Defines the proxy host for the `updatesdk` command.
`--proxy-host` *port*	`-pp`	Defines the proxy port for the `updatesdk` command.
`--test-apk` *path*	`-ta`	Path to the APK file to deploy and launch with the test command.
`--help`	`-h`	Display the list of options.
`--update-sdk-project <arg>`	`-up`	Update SDK project folder path, used with the `updatesdk` command.
`--verbose-fine`	`-v`	Verbose mode (level 1)
`--verbose-finer`	`-vv`	Verbose mode (level 2)
`--verbose-finest`	`-vvv`	Verbose mode (level 3) - shows all possible logs.
`--version`	`-V`	Display GMA build tool version.

Deploying mobile apps on iOS devices

This section contains information to create a mobile application to be deployed on iOS devices.

Directory structure for GMI apps

Platform-specific rules need to be considered when deploying on iOS devices (GMI).

The application sandbox

On iOS devices, program interactions with the file system are limited to the directories inside the app's sandbox.

Directory structure for a GMI application

Inside its application sandbox, an iOS app uses the following directory structure:

```
appdir/
|-- main.42m                              --
|-- *.42m                                   |
|-- *.42f                                   |
|-- fglprofile                              |
|   ...                                     |
```

```
|-- *.42s                                    |
|-- de/                                      | Program files
|   |-- *.42s                                |
|-- fr/                                      |
|   |-- *.42s                                |
|-- zh/                                      |
|   |-- *.42s                                |
|-- ... other resource files/dirs ...        |
|   ...                                      |
|-- webcomponents                            |
|   |-- component-type                       |
|       |-- component-type.html              |
|       |-- other-web-comp-resource          |
|   ...                                    --

Documents/
|-- ... writable app files ...

tmpdir/
|-- ... temporary files ...
```

Program files directory (*appdir*)

Application program files (`.42m`, `.42f`, as well as other program resources) need to be deployed in the *appdir* directory.

> **Important:** On iOS, the application program directory is read-only. Only the "`Documents`" directory is writable.

The program files directory can be found in programs with the `base.Application.getProgramDir on page 1705` method.

The FGLAPPDIR environment variable is automatically set to the *appdir* directory.

Program name (MAIN)

When deploying on mobile devices, the name of the program file must be `main.42m` or `main.42r`.

> **Note:** When using the command-line app build scripts, the name of the program file must be main.42?. When using Genero Studio, the packaging script takes care of renaming this file, if you have not named it `main`.

As with other program files, the "MAIN" module must be located under the *appdir* application program directory.

Working directory

The current working directory for an iOS application is typically a writable "`Documents`" directory, in the private folder of the app. For example, the path to the working directory can be "`/private/var/mobile/.../Documents`".

The current working directory can be found in program with the `os.Path.pwd on page 2004` method.

> **Note:** Any file access without an absolute path will be relative to the current working directory.

Files that need to be writable (such as SQLite database files) must be created or copied from the program files directory into the working directory. Copy must be done by the app at first execution, by using `base.Application.getProgramDir on page 1705`, to find the program files directory, and `os.Path.pwd()`, to find the working directory.

Temporary directory (*tmpdir*)

A temporary directory is available for the application.

In order to find the temporary directory for the app, use the standard.feInfo front call, with the "dataDirectory" parameter.

To create a temporary file name, use the os.Path.makeTempName() method.

Language directories for localized strings

When the app starts, the appropriate .42s string files will be loaded from the directory corresponding to the current language settings of the mobile device. String files to be loaded can be defined in app's fglprofile, or you can use the main program name to avoid fglprofile settings.

For each language supported by your application, a directory must exist under *appdir*, with a name including the locale codes. Consider also providing default string files (in English for ex) directly under *appdir*, in case if the regional settings of the device do not match one of the locale directories of the app, otherwise the application will stop with error -8006.

For example:

```
appdir/mystrings.42s
appdir/fr/mystrings.42s
appdir/de/mystrings.42s
```

For more details, see Localized string files on mobile devices on page 333.

Deploying a custom fglprofile file

If you need to set fglprofile entries for your mobile application, create a file with the name fglprofile, and deploy it under the *appdir* directory, along with the other program files.

See Understanding FGLPROFILE on page 164 for more details about fglprofile settings.

Creating the initial database file

When a mobile application starts for the first time, it typically creates a new database, or copies a existing database template file from the *appdir* program file directory (base.Application.getProgramDir on page 1705) to the working directory (os.Path.pwd on page 2004).

For more details about database creation on mobile devices, see Creating a database from programs on page 414.

Building iOS apps with Genero

Genero provides a command-line tool to build applications for iOS devices.

Basics

Genero mobile apps for iOS are distributed as IPA packages like any other iOS app. Genero provides a command line tool to build the .ipa package for your mobile application, or the .app directory for simulators.

> **Note:** This documentation section assumes that you are familiar with iOS app programming concepts and requirements. In order to build your apps, you must have an Apple developer account, as well as certificates and provisioning profiles to deploy your apps. For more details, visit the Apple developer site at https://developer.apple.com.

Prerequisites

Before starting the command line tool to build or deploy the app, fulfill the following prerequisites:

- The Genero BDL development environment (FGLDIR) must be installed on the Mac computer to compile your program files.
- The GMI build tool must be installed and available (check that the `gmibuildtool` command is available).

  Note: The GMI build tool is provided as a ZIP archive (`fjs-fglgmi-*.zip`) that must be extracted directly into FGLDIR.

  Important: When re-installing a new GMI archive, remove all "`build`" directories created by the `gmibuildtool`.
- An Apple developer account, device identifiers (UDID) and corresponding identifiers to sign your iOS app (certificate, bundle id, provisioning profile).

  Important: The UDID is the identifier of your physical device, it can be found with the `instruments -s` command when the device is plugged to the Mac. When deploying on a physical device, make sure that the UDID of the device is listed in the Apple Developer account that is used to generate the provisioning profiles.
- XCode must be installed on your Mac OS X computer (utilities from XCode toolchain are required).

  Note: Make sure that the installed XCode version supports the iOS versions of your mobile devices. As a general rule, update the XCode and iOS to the latest versions.
- iOS app resources such as icons and launch images (in all required sizes).

Finding the UDID of the plugged device

In order to find the UDID of the device plugged to your Mac, execute the `instruments -s` command, and identify the line describing your physical device:

```
$ instruments -s
Known Devices:
fraise [55D6D6C1-DE87-52F0-865E-3C6DC79F13D7]
Fourjs2 iPod touch (9.1) [78b7452fa9462c98c3bc7047da344314fd032004]
iPad 2 (9.0) [19CDA827-CA55-46F1-9376-BF61E2ECFDBB]
iPad Air (9.0) [F55E1207-C42B-472E-BD76-5B5AE46DE77A]
iPad Air 2 (9.0) [A0E8C4CD-67CD-42CB-84DF-9C75AC773293]
...
Known Templates:
"Activity Monitor"
"Allocations"
...
```

In the above output, the UDID of the iPod is `78b7452fa9462c98c3bc7047da344314fd032004`.

Environment settings

Before starting the command-line build tool,

- Make sure that XCode tools are available (try `xcodebuild` from the command line)

Creating the GMI front-end for development purpose

Four Js is not allowed to provide a ready-to-use front-end component for iOS devices, because of iOS app limitations defined by Apple: An iOS app shipped on the App Store cannot listen to a TCP port to provide a GUI service. Therefore, you will have to create your own GMI front-end, with your own Apple certificate and provisioning profile. The generated GMI can then be deployed on your device or simulator for development purpose listening on the port 6400, to display applications running on a server (FGLSERVER).

In order to build your own GMI front-end:

1. Make sure that the `gmibuildtool` is available (if not done yet, extract the fjs-fglgmi*.zip archive into FGLDIR).

2. Go to the `FGLDIR/demo/MobileDemo/gmiclient` directory.
3. Delete the complete `build` directory if it exists (can be done with a `make clean` command).
4. Make the GMI app with `make` (program files like `main.42m` file must exist).
5. Build the GMI front-end:

   - In order to build only the GMI front-end (`GMI.app` directory) for the simulator, execute the `gmibuildtool` command without any parameter:

     ```
     $ gmibuildtool
     ```

   - In order to build and install the GMI front-end on the simulator, first make sure that the simulator is started (`open -a simulator` command), then execute the `gmibuildtool` command with following parameters:

     ```
     $ gmibuildtool --device booted
     ```

   - In order to build only the GMI front-end IPA for devices, get a development certificate and provisioning profile and execute the `gmibuildtool` command with following parameters:

     ```
     $ gmibuildtool \
         --device phone \
         --certificate HGRW8... \
         --provisioning "~/Library/MobileDevice/Provisioning Profiles/
     myapp.mobileprovision"
     ```

     The generated IPA file can be found in the `build` subdirectory. This IPA file can be installed on your devices by using iTunes.

   - In order to build and install the GMI front-end on the device plugged to your Mac, get a development certificate and provisioning profile, and the exact device name (with the `instruments -s` command) and execute the `gmibuildtool` command with following parameters::

     ```
     $ gmibuildtool \
         --device "Mike's iPhone 6 (9.0)"  \
         --certificate HGRW8... \
         --provisioning "~/Library/MobileDevice/Provisioning Profiles/
     myapp.mobileprovision"
     ```

Specifying the target to build and deploy the iOS app

The `gmibuildtool` command can build and install iOS apps for the simulator or for physical devices.

The build and/or install action is controlled by the `--device` option:

- By default, when not specifying the `--device` option, a `GMI.app` directory is created for the simulator.
- When specifying the `--device booted` option, the `GMI.app` directory is created and the app is installed on the booted simulator.
- When specifying the `--device phone` option, the `GMI.app` directory and `.ipa` file are created.
- When specifying the `--device` *`physical-device-name`* option (with a real physical device name plugged on your Mac), the `GMI.app` directory and `.ipa` file are created and the app is installed on the device.

By default, the generated `GMI.app` directory and `.ipa` archive can be found in `$PWD/build` sub-directories. However you can specify the destination IPA file with the `--output` option.

Elements used to build the iOS app

The `gmibuildtool` command builds the iOS app package from the following:

- The GMI binary archive, containing the GMI front end and the FGL runtime system library,

Note: These files are provided in the `fjs-fglgmi-*.zip` archive that must be extracted directly under FGLDIR.

Important: When re-installing a new GMI archive, remove all "`build`" directories created by the `gmibuildtool`.

- The compiled application program and resource files (`.42m`, `.42f`, etc),

  Note: The application program files must include a `main.42m` or `main.42r` module.

- The display name of the app (`--app-name` parameter),
- The version of the app (`--app-version` parameter),
- The debug or release mode (`--mode` parameter),
- The certificate (to sign the app) (`--certificate` parameter),
- The bundle Identifier (`--bundle-id` parameter),
- The app provisioning profile (`.mobileprovision` file) (`--provisioning` parameter),
- iOS app specific resources:
  - App icons (`--icons` parameter),
  - Launch images (`--launch-images` parameter) or launch storyboard file (`--storyboard` parameter).

For a complete description of command options, see gmibuildtool on page 2592.

Default build directory structure

For convenience, the build tool supports a default directory structure to find all files required to build the app:

```
top-dir
|
|-- main.42m and other program files, as described in Directory structure
 for GMI apps on page 2584
|
|-- gmi
|   |
|   |-- Info.plist
|   |
|   |-- LaunchScreen.storyboard
|   |
|   |-- Default@2x.png
|   |-- Default-568h@2x.png
|   |-- Default-Landscape.png
|   |-- Default-Landscape-667h@2x.png
|   |-- Default-Landscape-736h@3x.png
|   |-- Default-Landscape@2x.png
|   |-- Default-Portrait.png
|   |-- Default-Portrait-736h@3x.png
|   |-- Default-Portrait-667h@2x.png
|   |-- Default-Portrait@2x.png
|   |   ...
|   |-- icon_29x29.png
|   |-- icon_40x40.png
|   |-- icon_57x57.png
|   |-- icon_58x58.png
|   |-- icon_72x72.png
|   |-- icon_76x76.png
|   |-- icon_80x80.png
|   |-- icon_120x120.png
|   |-- icon_152x152.png
|   |   ...
```

In the above directory structure:

1. *top-dir* is the top directory of the default structure. It will typically hold your application program files. A different program files directory can be specified with the `--program-files` option.
2. *top-dir*`/gmi` is the default directory containing the app resource files such as icons:
   a. `Info.plist` is the Information Property List File that will be used to build the app. Some properties will be overwritten by `gmibuildtool` options like `--app-name` and `--app-version`.
   b. `LaunchScreen.storyboard` is the default storyboard file for the app launch screen. This file can be specified with the `gmibuildtool --storyboard` option.
   c. `Default-*.png` are the app launch image files. The directory to find launch images can be specified with the `gmibuildtool --launch-images` option.
   d. `icon_*.png` are the app icon files. The directory to find icons can be specified with the `gmibuildtool --icons` option.

Debug and release versions

iOS apps can be generated in a debug or release version. Release version are prepared for distribution on the App Store, while debug versions are used in development.

In debug mode, the app installed on the device can listen on the debug TCP port to allow `fgldb -m` connections, after enabling the debug port in the app settings.

Debug or release mode must be specified in the command line with the `--mode debug` or `--mode release` option. Additionnally, if you want to deploy on a physical device, you need to use a provisioning profile corresponding to the debug or release mode:

- In debug mode, the certificate must be a development certificate.
- In release mode, the certificate must be a distribution certificate.

Defining the app version and build number

Apple distinguishes the app version number of a bundle (visible to the end user), from the build version number of a bundle (called a release version number in Apple docs).

You specify the app version number with the `--app-version` option of the `gmibuildtool` command. This option sets the `CFBundleVersion` property of the `Info.plist` file), and must match the version specified in iTunes Connect.

In order to distinguish multiple builds (Apple's term is "releases") of the same app version number, define the build version number of your app with the `--build-number` option. This option sets the `CFBundleShortVersionString` property of the `Info.plist` file. For a given app version, you need to increase this build number, to be able to upload a new binary on iTunes Connect.

> **Note:** If you do not specify the `--build-number` option, the build version number defaults to the app version specified with the `--app-version` option.

Defining app properties in the ./gmi/Info.plist file

iOS app are created with a set of properties that are essential configuration information for a bundled executable. These properties are defined in the "Information Property List File", an XML formatted file, named `Info.plist` by convention.

Most important `Info.plist` properties are defined with `gmibuildtool` options such as `--app-name` and `--app-version`. However, you may need to define other properties that are out of the scope of the build tool. For example: background modes, device capabilities, screen orientations, permanent wifi, etc.

In order to define specific app properties, setup an `Info.plist` file in *top-dir*`/gmi` directory, before executing the `gmibuildtool`. Properties covered by the build tool will be overwritten, while any other property defined in the *top-dir*`/gmi/Info.plist` file will be left untouched.

For more details about the `Info.plist` file structure, see Apple developer site page about Information Property List File.

Building an iOS app with gmibuildtool

Follow the next steps to setup a GMI app build directory in order to create an iOS app, based on the default directory structure:

1. Create the root distribution directory (*`top-dir`*)
2. Copy compiled program files (`.42m`, `.42f`, `fglprofile`, application images, web component files, etc) under *`top-dir`*.
3. Copy the default English `.42s` compiled string resource file under *`top-dir`*.
4. Create non-English langage directories (fr, ge, ...) under *`top-dir`* and copie the corresponding `.42s` files.
5. Copy default application data files (database file for ex) under *`top-dir`*.
6. Create the *`top-dir`*`/gmi` directory.
7. Copy iOS app resources (icons, launch screen, storyboard) under *`top-dir`*`/gmi`.
8. If needed, create an *`top-dir`*`/gmi/Info.plist` file, to define specific iOS app properties.

Once the build directory is prepared, issue the following commands:

```
$ cd top-dir
$ gmibuildtool \
    --output myapp.ipa \
    --app-name "My App" \
    --app-version "v3.1.6" \
    --bundle-id "com.example.mycompany.myapp" \
    --mode release \
    --certificate HGRW8... \
    --provisioning "~/Library/MobileDevice/Provisioning Profiles/
myapp.mobileprovision" \
    --device phone
```

Building a GMI app with C extensions or custom front calls

In order to create an iOS app using C extensions written in Objective-C as in Implementing C-Extensions for GMI on page 1613, you need to setup a `Makefile` calling the `FGLDIR/lib/Makefile-gmi` generic makefile file.

In your Makefile, define the following variables to be passed to the generic makefile:

- `APPNAME`: Defines the display name of the app.
- `BUNDLE_IDENTIFIER`: Defines the Bundle Id (or App Id) of the app.
- `IDENTITY`: Defines the certificate to be used for this app.
- `PROVISIONING_PROFILE`: Defines the provisioning profile generated for this app.
- `USEREXTENSION`: Defines the lib name containing the C extensions.
- `TARGET`: Defines the device where the app must be installed (can be `phone` or `simulator`).

Custom Makefile example:

```
...

all: $(MODULES) $(FORMS) ...

run: all userextension.dylib
      fglrun -e userextension main

userextension.dylib: userextension.c
      fglmkext $?
...
```

```
GMI_OPTIONS = \
   APPNAME=MyApp \
   BUNDLE_IDENTIFIER=com.mycomany.myapp \
   IDENTITY=HGRW8... \
   PROVISIONING_PROFILE=~/Library/MobileDevice/Provisioning\ Profiles/
myapp.mobileprovision \
   USEREXTENSION=userextension.o \
   TARGET=phone

GMI_MAKE = make -f $(FGLDIR)/lib/Makefile-gmi $(GMI_OPTIONS)

gmi.all: all
      $(GMI_MMAKE) all

gmi.install: all
      $(GMI_MMAKE) install

gmi.uninstall:
      $(GMI_MMAKE) uninstall

gmi.info:
      $(GMI_MMAKE) info

gmi.clean:
      ~$(GMI_MMAKE) clean
```

The same technique can be used to build apps that must include custom front calls.

For complete examples, see `FGLDIR/demo/MobileDemo/userextension` and `FGLDIR/demo/MobileDemo/userfrontcall`

gmibuildtool
The gmibuildtool is a utility to create and test applications for an iOS devices.

Syntax

```
gmibuildtool [options]
```

1. *options* are described in Table 573: gmibuildtool options on page 2592.

Options

Table 573: gmibuildtool options

Option	Description
`--app-name` *application-name*	Display name of the mobile app. This option can be specified to define the display name of the app, it sets the `CFBundleDisplayName` property in the `Info.plist` file. If not specified, the name defaults to "Noname".
`--app-version` *application-version*	Defines app version visible to the users on the App Store. This option is mandatory and sets `CFBundleVersion` properties in the `Info.plist` file.

Option	Description
	Note: If the `--build-number` option is not used, `--app-version` will also set the both the `CFBundleShortVersionString` property. In iTunes Connect, you define the version of your app, that must match the `CFBundleVersion` property in the `Info.plist` file of the app. If these versions do not match, the app cannot be published. Once the app is visible on App Store, the version specified in iTunes Connect shows up in the "Version" section of the application page. The app version number should be a string comprised of three period-separated integers. For example: "1.4.2"
`--bundle-id` *`bundle-identifier`*	Defines the Bundle Identifier (a.k.a. App Id) for the app. This option is mandatory and sets the `CFBundleIdentifier` property in the `Info.plist` file. A bundle identifier is the unique identifier of your app, to let iOS recognize new app versions. When developing for the simulator, you can choose your own identifier. When creating an application for the App Store, the bundle identifier must be registered with Apple. If not specified, the name defaults to "noname" (for prototyping).
`--build-number` *`build-number`*	Defines the build number used to upload a new binary of the same app version. This option must be used to distinguish different builds for the same app version. It sets the `CFBundleShortVersionString` property in the `Info.plist` file. The build number needs to be incremented in order to upload a new binary version of the same app version in iTunes Connect. If this option is not used, the build number defaults to the version specified with the `--app-version` option. The build number is a string comprised of three period-separated integers. For example: "1.4.2"
`--certificate` *`identity`*	Name of a certificate to sign the app. This option is mandatory to build apps for a physical device or for the app store. The certificate can be found in the Keychain access program, in the "Common Name" field of the certificate panel. The command `security find-identity -v` can be used to list all available certificates.
`--device` *`device-name`*	Defines the name of a device or simulator. • By default, when not specifying the `--device` option, a `GMI.app` directory is created for the simulator.

Option	Description
	• When specifying the `--device booted` option, the `GMI.app` directory is created and the app is installed on the booted simulator. • When specifying the `--device phone` option, the `GMI.app` directory and `.ipa` file are created. • When specifying the `--device` *`physical-device-name`* option (with a real physical device name plugged on your Mac), the `GMI.app` directory and `.ipa` file are created and the app is installed on the device. **Note:** Use the `instruments -s` XCode command to find the list of available devices (simulators or connected devices).
`--help`	Display the help of the command tool.
`--icons` *`icons-dir`*	Provides the directory where the application icons are located. By default, the application icons directory is *`current-working-dir`*`/gmi`. The name of the app icon files must be: `icon_57x57.png, icon_72x72.png, icon_29x29.png, icon_40x40.png, icon_120x120.png, icon_152x152.png, icon_58x58.png, icon_76x76.png, icon_80x80.png`
`--launch-images` *`launch-images-dir`*	The directory where launch images are located. By default, the launch images directory is *`current-working-dir`*`/gmi`. **Note:** This option is ignored if the `--storyboard` option is provided. The name of the image files must be: `Default.png, Default@2x.png Default-568h@2x.png, Default-Portrait-667h@2x.png Default-Landscape-667h@2x.png, Default-Portrait-736h@3x.png, Default-Landscape-736h@3x.png, Default-Portrait.png, Default-Landscape.png Default-Portrait@2x.png, Default-Landscape@2x.png`. Each file name corresponds to a device type (you may not need to provide all files if you target only recent iOS devices), see Apple Developer documentation for more details about launch images.
`--mode {debug|release}`	Controls the debug or release mode for the app. By default, the mode is `debug`. Note that the provisioning profile must correspond: • `--mode debug`: Development provisioning profile. • `--mode release`: Distribution provisioning profile.
`--output` *`ipa-file-name`*	Path to output IPA and APP files to be generated.

Option	Description
	By default, a "`build`" directory is created, with subdirectories containing the .ipa and .app files. An IPA file is created when building an application for a physical device and the App Store. The IPA file is not needed and will not be created when building for the simulator.
`--program-files` *program-dir*	Path to Genero BDL program files (`.42m`, `.42f`, etc). By default, the program files directory is the current work directory. Following files are automaticlly excluded: `*.4gl`, `*.per`, `*.msg`, `*.str`, `*.sch`, `[Mm]akefile`, `*.42d`, `[Mm]akefile`, `*.[chdmo]`, `*.xib`, `build/` (the build directory), `gmi/` (this folder is the default location of LaunchScreens and AppIcons). If the file `gmiignore` exists, then this file contains additional files to be ignored.
`--provisioning` *provisioning-file*	Path to the provisioning profile (`.mobileprovision`). The provisioning profile is mandatory to build apps for a physical device or for the app store. Provisioning profiles can be found in `$HOME/Library/MobileDevice/Provisioning\ Profiles/`
`--storyboard` *storyboard-file*	Path to the storyboard file, to get a splash screen to be displayed when the app starts. This file is an alternative for Launch Screens (`--launch-images` option). This option is mandatory if you do not provide launch images with the `--launch-images` option. The default storyboard is an empty screen.

Running mobile apps on an application server

From the mobile device, programs can be started remotely on an application server, and displayed on the device.

Purpose of remote application execution for mobile devices

Remote applications displayed on a mobile device allow the use of the processor, memory, storage and software resources available on a server, for mobile users.

> **Note:** Executing remote/server applications for display on a mobile device requires a reliable and constant network connection. If the network connection fails, the application will stop, as with other client/server Genero front-ends.

Server applications can only be started through the Genero Application Server (GAS), by using the UA protocol available since version 3.00. You must set up and configure the GAS for the programs you want to start remotely. See the GAS documentation for more details.

> **Note:** Applications executed on the GAS server must use the UTF-8 encoding. Mobile front-ends will reject any attempt to display forms of an application using an encoding other than UTF-8.

Implementing the embedded mobile app

Create a small application to be deployed on the mobile device, which then starts the application(s) on an GAS server.

The server application is started from the embedded application through the `runOnServer` front call. The embedded mobile application can be a very simple `MAIN / END MAIN` program, only performing the "`runOnServer`" front call.

For example, this is the very minimal embedded application, starting a program on the GAS:

```
MAIN
    CALL ui.interface.frontcall("mobile","runOnServer",
                                ["http://myappserver:6394/ua/r/myapp"],[])
END MAIN
```

When the remote application starts, the graphical user interface displays on the mobile device.

The `runOnServer` front call returns when the called application ends, control goes back to the initial application executing on the mobile device.

> **Note:** In development context, it is possible to execute the parent starter app on a server, display on a mobile device with FGLSERVER on page 185 set properly, and use the `runOnServer` front call. Because starting remote GAS applications is done with a front call, this configuration mimics an embedded starter app running on the device.

Using the runOnServer front call

The application executed on the server-side is identified by the first parameter of the `runOnServer` front call. This application must be delivered by the Genero Application Server. The parameter must contain an "`ua/r`" URL syntax (the UA protocol introduced with the GAS 3.00).

For example: `http://myappserver:6394/ua/r/myapp`

The URL may contain a query string, with parameters for the application to be executed by the GAS.

If needed, you can add a second argument to define a timeout as a number of seconds. The embedded application will wait for the remote application to start, until the timeout expired. If no timeout parameter is specified, or when zero is passed, the timeout is infinite.

In case of failure (application not found, timeout expired), the front call raises the runtime error -6333 and the HTTP status code of the request can be found in the error message details. Use a `TRY/CATCH` block to check if the execution the server application was successful:

```
MAIN
  TRY
    CALL ui.interface.frontcall("mobile","runOnServer",
         ["http://myappserver:6394/ua/r/myapp"],[])
  CATCH
    ERROR err_get(STATUS)
  END TRY
END MAIN
```

Subsequent server-side application runs are allowed; the last active application will display on the device. However, it is not possible to navigate between started applications. Therefore, an application started with the `runOnServer` front call must only use the `RUN` instruction to start sub-programs. `RUN WITHOUT WAITING` is not supported.

Passing parameters to the server application

If needed, the embedded app can pass arguments to the server application by using parameter specification in the URL string, with the `?Arg=value1&Arg=value1&...` notation:

```
DEFINE params, base, complete_url STRING
LET params = "Arg=verbose&Arg=5677"
LET url = "http://myappserver:6394/ua/r/myapp"
LET complete_url = base || "?" || params
```

The remote program can retrieve the parameters with the `arg_val()` built-in function.

Note: It is not needed to URL-encode the string passed to the `runOnServer` front call.

See the GAS documentation (`AllowUrlParameters` attribute) about passing parameters in the application URL.

This is an example of an embedded application to be deployed on the mobile device, which passes parameters to a server-side application:

```
IMPORT util

MAIN
    DEFINE arr DYNAMIC ARRAY OF STRING, x INT
    MENU "test"
        COMMAND "runOnServer"
           CALL arr.clear()
           LET arr[1] = "first argument"
           LET arr[2] = "second argument"
           LET x = do_run("http://10.0.40.29:6394/ua/r/test1", 10, arr)
        COMMAND "exit"
           EXIT MENU
    END MENU
END MAIN

FUNCTION do_run(url,timeout,params)
  DEFINE url STRING,
         timeout SMALLINT,
         params DYNAMIC ARRAY OF STRING
  DEFINE i, r INTEGER, tmp STRING
  LET r = 0
  LET tmp = url
  FOR i=1 TO params.getLength()
      LET tmp = tmp || IIF(i==1,"?","&") || "Arg=" || params[i]
  END FOR
  TRY
    CALL ui.interface.frontcall("mobile","runOnServer",[tmp,timeout],[])
  CATCH
    ERROR err_get(STATUS)
    LET r = -1
  END TRY
  RETURN r
END FUNCTION
```

A sample server-side application:

```
MAIN
    MENU "Prog1"
        COMMAND "arg1" MESSAGE "Arg 1 = ", arg_val(1)
        COMMAND "arg2" MESSAGE "Arg 2 = ", arg_val(2)
        COMMAND "arg3" MESSAGE "Arg 3 = ", arg_val(3)
        COMMAND "Quit" EXIT MENU
    END MENU
```

```
END MAIN
```

Sharing files between embedded and server app

If files need to be shared between the embedded application and the server application, the application running on the GAS can only access the *data-directory* directory, in the sandbox of the embedded application that executes the "`runOnServer`" front call.

This matters when using file handling APIs such as `fgl_putfile()` and `fgl_getfile()` or front calls like `takePhoto` and `launchURL`.

The *data-directory* on the mobile device can be found with the `feInfo/dataDirectory` front call. In both the embedded app and the app running on the server, this front-call will return the same directory.

The following workflow can be used:

1. Before starting the server application with a `runOnServer` front call, the embedded app must copy files to the *data-directory*.
2. While executing, the server application can retrieve files from the *data-directory* with `fgl_getfile()`, and send its own files to the *data-directory*, with `fgl_putfile()`.
3. When the server application terminates, the embedded app can read files the server application left in the *data-directory*.

> **Note:** If several remote applications are started successively on the server with a `RUN` instruction, make sure to not overwrite files written by other server programs.

In order to write code for the embedded app, that can be executed in development mode (running on a server) and on the mobile device, you can adapt to the execution context: Make a simple file copy when executing on the mobile device, or do an `fgl_putfile()` call, when running on the development server. Check the execution context with the `base.Application.isMobile()` method.

This example, in the embedded app on the mobile device, copies a file from the device private directory to the *data-directory*:

```
IMPORT os
...
    CALL mobile_copy_to_data_dir("myfile.txt")
...
FUNCTION mobile_copy_to_data_dir(fn)
    DEFINE fn, dd, dst STRING, r INT
    CALL ui.interface.frontcall("standard","feInfo",["dataDirectory"],[dd])
    -- Always use / as path sep for Android/iOS dirs.
    LET dst = dd || "/" || os.Path.basename(fn)
    IF base.Application.isMobile() THEN
       -- Executing on device: make a simple copy to data-dir
       LET r = os.Path.copy(fn, dst)
       MESSAGE SFMT("COPY status = %1", r)
    ELSE
       -- Executing on dev server: make a file transfer to data-dir
       CALL fgl_putfile(fn, dst)
    END IF
END FUNCTION
```

> **Note:** We do not use the `os.Path.join()` method here because it would add the path separator according to the operating system where the application is executed. This would not be a problem when executing on the mobile device or Unix-like platforms. However, when running on a Windows platform, the `os.Path.join()` method would join the directory and the file name with a backslash, and the resulting path would not fit Android or iOS directory path specification for the *data-directory*.

In the server application, use the `fgl_getfile()` function, to transfer a file from the mobile device *data-directory* to the local server disk:

```
IMPORT os
...
  CALL server_get_from_data_dir("myfile.txt", "/tmp/server_file.txt")
...
FUNCTION server_get_from_data_dir(fn, dst)
    DEFINE fn, dst, dd, src STRING
    CALL ui.interface.frontcall("standard","feInfo",["dataDirectory"],[dd])
    -- Use / as path sep for Android/iOS dirs!
    LET src = dd || "/" || fn
    CALL fgl_getfile(src, dst)
END FUNCTION
```

Similarly, in the server application, use the `fgl_putfile()` function, to copy a file from the server application to the *data-directory* of the embedded app:

```
IMPORT os
...
  CALL server_put_to_data_dir("/tmp/server_file.txt", "myfile.txt")
...
FUNCTION server_put_to_data_dir(src, fn)
    DEFINE src, fn, dd, dst STRING
    CALL ui.interface.frontcall("standard","feInfo",["dataDirectory"],[dd])
    -- Use / as path sep for Android/iOS dirs!
    LET dst = dd || "/" || fn
    CALL fgl_putfile(src, dst)
END FUNCTION
```

Push notifications

This section describes how to implement push notification with Genero.

A push notification is a short message sent by a central server entity to an app installed on a mobile device. In order to be notified, the app+device must register itself to a push service (a global service such as Google Cloud Messaging), and register also to a push provider (part of the custom application). To indicate that fresh information is available, notifications are sent by push providers to the push service, which broadcasts notifications to registered devices. The apps can then get details about the notification and display a little hint to the end user. Enterprise mobile applications can use push notifications to produce urgent and important updates for users.

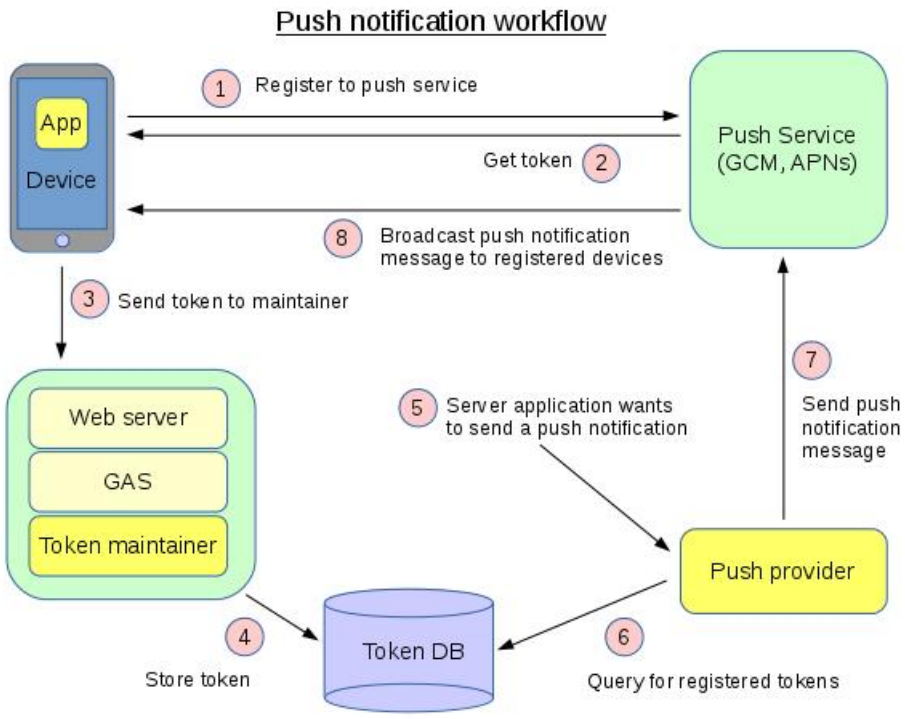


Figure 124: This figure describes the workflow for a push notification (items in yellow are the components that can be implemented with Genero BDL)

Workflow:

1. The app registers to the push service.
2. The push service generates a unique token to identify the device+app and returns this token to the app.
3. The app transmits the token to the token maintainer.
4. The token maintainer stores the new token in a database.
5. Some event occurs in the global application workflow that requires a push notification to warn all registered devices/apps.
6. The push provider reads the database for registered tokens.
7. The push provider sends push notification requests to the push service.
8. The push service broadcasts the notification messages to all registered devices.

There are several push notification mechanisms available. This chapter covers the Google Cloud Messaging (GCM) and Apple Push Notification services (APNs).

Common components can be implemented on the same code base for both GCM and APNs push notification mechanisms: The mobile app and the token maintainer.

Google Cloud Messaging (GCM)

Follow this procedure to implement push notification with GCM.

Introduction to GCM push notification

The push notification solution described in this section is based on the Google Cloud Messaging service. Familiarize yourself with GCM by visiting the https://developers.google.com/cloud-messaging web site.

Google Cloud Message services allow push servers to send notification message data to registered Android™ or iOS devices.

The system involves the following actors:

- The Google Cloud Message service (GCM):

GCM provides push server and client identification. It also handls all aspects of queuing of messages and delivery to the target application running on registered devices.

- The registration tokens maintainer:

  A Web Services server program maintaining the database of registration tokens with application user information. This program must listen to new device registration events and store them in a database. The push server program can then query this database to build the list of registration tokens to identify the devices to be notified.
- The push server program:

  Implemented by a third-party service or as a Genero BDL program using the Web Services API. This push server program will send notification messages to GCM with two connection servers (HTTP and XMPP).
- Devices running the Genero app registered to the push notification server:

  Registered devices use the push notification client API to register, get notification data and unregister from the service.

  > **Note:** The database used to store registration tokens must be a multi-user database (do not use SQLite for example), since two distinct programs will use the database.

Creating a GCM project

To initiate a push notification service dedicated to your applications, you must first create a Google Cloud Messaging project on the Google web site. Creating a GCM project will give you the *API Key* and the *Sender ID*. The API Key is the authentication key to access Google services. The Sender ID identifies your GCM project; this id will be used by your mobile app to indicate that it wants to get messages from this GCM project.

To get details about GCM project creation, visit: https://developers.google.com/cloud-messaging.

To create a GCM project and get the API Key and Sender ID, follow the steps at: https://developers.google.com/cloud-messaging/android/client#get-config.

Write down the API Key and the Sender ID generated for you, as these will be used later on.

Implementing the registration tokens maintainer

To handle device registrations on the server side of your application, the same code base can be used for GCM and other token-based frameworks.

For more details, see Implementing a token maintainer on page 2611.

Implementing the push server

The push server will produce application notification messages that will be transmitted to the GCM service. The GCM service will then spread them to all mobile devices registered to the service with the Sender ID.

> **Important:** The size of an GCM notification content cannot exceed 4 Kilobytes. If more information needs to be passed, after receiving the push message, apps must contact the server part to query for more information. However, this is only possible when network is available.

The push server will use RESTFul HTTP POST requests to send notifications through the GCM service to the following URL:

`"https://gcm-http.googleapis.com/gcm/send"`.

The HTTP POST header must contain the following attributes:

```
Content-Type:application/json
Authorization:key=API_Key
```

where *API_Key* is the API Key obtained during GCM project creation.

The push server program can be implemented with the Web Services API to make RESTFul requests as follows:

```
IMPORT com
IMPORT util

FUNCTION gcm_send_notif_http(api_key, notif_obj)
  DEFINE api_key STRING,
         notif_obj util.JSONObject

  DEFINE req com.HTTPRequest,
         resp com.HTTPResponse,
         req_msg STRING

  TRY
    LET req = com.HTTPRequest.Create("https://gcm-http.googleapis.com/gcm/
send")
    CALL req.setHeader("Content-Type", "application/json")
    CALL req.setHeader("Authorization", SFMT("key=%1", api_key))

    CALL req.setMethod("POST")
    LET req_msg = notif_obj.toString()
    IF req_msg.getLength() >= 4096 THEN
       LET res = "ERROR : GCM message cannot exceed 4 kilobytes"
       RETURN res
    END IF
    CALL req.doTextRequest(req_msg)
    LET resp = req.getResponse()
    IF resp.getStatusCode() != 200 THEN
      DISPLAY SFMT("HTTP Error (%1) %2",
                   resp.getStatusCode(),
                   resp.getStatusDescription())
    ELSE
      DISPLAY "Push notification sent!"
    END IF
  CATCH
    DISPLAY SFMT("ERROR : %1 (%2)", STATUS, SQLCA.SQLERRM)
  END TRY

END FUNCTION
```

The body of the HTTP POST request must be a JSON formatted record using a structure similar to the following example:

```
{
  "collapse_key":  "stock_update",
  "time_to_live": 108,
  "delay_while_idle": true,
  "data":
  {
    "stock_change":
    {
      "stock_id" : "STK-034" ,
      "timestamp" : "2015-02-24 15:10:34.18345",
      "item_count" : 15023
    },
  },
  "registration_ids" : [ "APA91b...", "Hun4MxP...", "5ego..." ]
}
```

> **Note:** This notification message uses the "registration_ids" attribute to provide a list of devices to be notified. If you want to notify a single device, use the "to" attribute instead of "registration_ids", and pass a single registration token instead of a JSON array.

For more details about the JSON request structure in a GCM HTTP POST, see https://developers.google.com/cloud-messaging/http.

By convention, if the "data" member of the JSON request defines a "genero_notification" member, the front-end will show graphical notification (popup hint) with the "title", "content" and the "icon" values.

> **Note:** With GMA, the icon should be packaged in the APK and should be accessible by name (as the gma_ic_genero.png in the drawable folders)

For example:

```
...
  "data":
  {
    "genero_notification":
    {
      "title":    "Stock has changed",
      "content":  "New stock information will be retrieved from the backend
 server...",
      "icon":     "stock_update"
    },
    ...
  },
  "registration_ids" : [ "APA91b...", "Hun4MxP...", "5ego..." ]
}
```

The next code example implements a function that creates the JSON object, which can be passed to the gcm_send_notif_http() function described above. The only purpose of this notification message is to test the "genero_notification" popup hint. The function takes an array of registration tokens as a parameter, which will be used to set the "registration_ids" attribute:

```
FUNCTION gcm_simple_popup_notif(reg_ids, notif_obj, popup_msg)
    DEFINE reg_ids DYNAMIC ARRAY OF STRING,
           notif_obj util.JSONObject,
           popup_msg STRING
    DEFINE data_obj, popup_obj util.JSONObject

    CALL notif_obj.put("registration_ids", reg_ids)

    LET data_obj = util.JSONObject.create()

    LET popup_obj = util.JSONObject.create()
    CALL popup_obj.put("title", "Notification message!")
    CALL popup_obj.put("content", popup_msg)
    CALL popup_obj.put("icon", "genero")

    CALL data_obj.put("genero_notification", popup_obj)
    CALL data_obj.put("other_info", "Additional data...")

    CALL notif_obj.put("data", data_obj)

END FUNCTION
```

The gcm_simple_popup_notif() and gcm_send_notif_http() functions can then be used as follows:

```
IMPORT com
```

```
IMPORT util

MAIN
    CONSTANT api_key = "xyz..."
    DEFINE reg_ids DYNAMIC ARRAY OF STRING,
           notif_obj util.JSONObject

    LET reg_ids[1] = "APA91bHun..."
    LET reg_ids[2] = "B4AA2q7xa..."

    LET notif_obj = util.JSONObject.create()
    CALL gcm_simple_popup_notif(reg_ids, notif_obj, "This is my message!")
    CALL gcm_send_notif_http(api_key, notif_obj)

END MAIN
```

In order to use the tokens database maintained by a token maintainer program, your GCM push server can collect registration tokens as shown in the following example:

```
FUNCTION gcm_collect_tokens(reg_ids)
    DEFINE reg_ids DYNAMIC ARRAY OF STRING
    DEFINE rec RECORD
               id INTEGER,
               sender_id VARCHAR(150),
               registration_token VARCHAR(250),
               badge_number INTEGER,
               app_user VARCHAR(50),
               reg_date DATETIME YEAR TO FRACTION(3)
           END RECORD
    DECLARE c1 CURSOR FOR
      SELECT * FROM tokens
       WHERE sender_id IS NOT NULL -- In case if APNs tokens remain in the
 db
    CALL reg_ids.clear()
    FOREACH c1 INTO rec.*
        CALL reg_ids.appendElement()
        LET reg_ids[reg_ids.getLength()] = rec.registration_token
    END FOREACH
END FUNCTION
```

The above function can then be used by another function to send the push message to all registered devices:

```
FUNCTION gcm_send_text(api_key, the_text)
    DEFINE api_key, the_text STRING
    DEFINE reg_ids DYNAMIC ARRAY OF STRING,
           notif_obj util.JSONObject,
           info_msg STRING
    CALL gcm_collect_tokens(reg_ids)
    IF reg_ids.getLength() == 0 THEN
       RETURN "No registered devices..."
    END IF
    LET notif_obj = util.JSONObject.create()
    CALL gcm_simple_popup_notif(reg_ids, notif_obj, the_text)
    LET info_msg = gcm_send_notif_http(api_key, notif_obj)
    RETURN info_msg
END FUNCTION
```

Handle push notifications in mobile apps

To handle push notifications in mobile apps, the same code base can be used for GCM and other token-based frameworks.

For more details see Handling notifications in the mobile app on page 2617.

Apple Push Notification Service (APNs)

Follow this procedure to implement push notification with APNs.

Introduction to APNs push notification

The push notification solution described in this section is based on the Apple Push Notification Service. Familiarize yourself with APNs by visiting the Apple Push Notification Service web site.

Apple Push Notification service allows push servers to send notification message data to registered iOS (and OS X) devices.

The APNs service transports and routes a remote notification from a given provider to a given device. A notification is a short message built from two pieces of data: the device token and the payload.

> **Note:** Each device needs to be identified by its device token, and the provider must send individual notification messages for each registered device.

The system involves the following actors:

- The Apple Push Notification Service (APNs):

  APNs provides push server and client identification. It also handles all aspects of message queuing and delivery to the target applications running on registered devices. The APNs system includes a feedback service that can be queried to check for devices that have unregistered and no longer need to be notified.
- The device tokens maintainer:

  A Web Services server program maintaining the database of device tokens, with application user information. This program must listen to new device registration events, store them in a database, and from time to time query the APNs feedback service to check for unregistrations.
- The push provider:

  This program will send notification messages to the APNs server by using the `com.APNS` class and TCP request API. The push provider program will query the device token database to know which devices need to be notified.
- Devices running the Genero app registered to the push notification server:

  Registered devices use the push notification client API to register, get notification data and unregister from the service.

  > **Note:** The database used to store device tokens must be a multi-user database (do not use SQLite for example), since two distinct programs will use the database.

APNS push notification security

iOS apps must be created with an Apple certificate for development or distribution, linked to an App ID (or Bundle ID) with push notification enabled. The provisioning profile used when building the IPA must be linked to the App ID with push enabled. Certificate, provisioning and bundle id must be specified to the GMI build tool.

To create the push provider linked to your app, usually you need to create two Apple Push Notification certificates linked to your App ID (you select the App ID when you create a push certificate in the Apple member center): One certification for development and another for distribution. For more details about the push provider certificates, see APNs SSL certificate on page 2095.

Check also Apple Push Notification documentation for more details about certificate requirements for push notifications.

Identifying target devices

Each APNs client device is identified by a *device token*. A device token is an opaque identifier of a device that APNs gives to the device when an app registers itself for push notification. It enables APNs to locate in a unique manner the device on which the client app is installed. The device shares the device token with the push provider. The push provider must produce notification messages for each device by including the device token in the message structure.

> **Important:** The mobile app obtains its device token by registering to the APNs service with the registerForRemoteNotifications on page 1934 front call. It is then in charge of sending its device token to the push provider; typically through a RESTFul request. The push provider must collect and store the device tokens, as they need to be specified in a push notification message send by the push provided.

Notification content (payload)

In a notification message, the *payload* is a JSON-defined property list that specifies how the user of an app on a device is to be alerted.

> **Important:** The size of an APNS notification payload cannot exceed 2 Kilobytes. Make sure that the resulting BYTE variable does not exceed this size limitation. If more information needs to be passed, after receiving the push message, apps must contact the server part to query for more information. However, this is only possible when network is available.

The payload must contain a list of "`aps`" records. Each "`aps`" record represents a notification message to be displayed as a hint on the device (for example, by adding a badge number to the app icon). The "`aps`" records can also contain custom data in a separate set of JSON attributes.

In the Genero mobile app, the notification messages are obtained by using the getRemoteNotifications on page 1930 front call, after a `notificationpushed` action was detected with an `ON ACTION` handler.

> **Important:** When an iOS app is in background, silent push notifications can occur, but notification message data (i.e. the payload) may not be available. In such case, GMI is able to detect that a notification arrived (i.e. when the app badge number is greater than zero) and raise the `notificationpushed` action, but the `getRemoteNotifications` front call will return no message data (*data* return param is `NULL`). If such case, implement a fallback mechanism (based on RESTFul web services for example), to contact the push notification provider and retrieve the message information.

Example of notification record list (JSON array) returned by the `getRemoteNotifications` front call:

```
[
 {
  "aps" :
  {
    "alert" : "My first push",
    "badge" : 1,
    "sound" : "default",
    "content-available" : 1
  }
 },
 {
  "aps" :
  {
    "alert" :
    {
      "title" : "Push",
      "body" : "My second push"
    }
    "badge" : 2,
    "sound" : "default",
    "content-available" : 1
```

```
  },
  "new_ids" : [ "XV234", "ZF452", "RT563" ],
  "updated_ids" : [ "AC634", "HJ153" ]
 }
]
```

Badge number handling

With APNs, badge number handling is in charge of the application code: The push provider sends a badge number in the payload records, the app can check the message content, and must communicate with a server component, to indicate that the notification message has been consumed. The server program can then maintain a badge number for each registered device, decrementing the badge number.

In order to set or query the badge number for your app, use the following front calls:

- setBadgeNumber (iOS) on page 1946
- getBadgeNumber (iOS) on page 1945

In this tutorial, badge numbers are stored on the server database. The token maintainer handlers requests from apps to sync the badge number for a given device token, and the push provider program reads the database to set the badge number in the notification payload. When the app consumes messages, it queries and resets the app badge number with the `getBadgeNumber`/`setBadgeNumber` front calls, and informs the token maintainer to sync the badge number in the central database.

Communication channels

A provider communicates with Apple Push Notification service over a binary network interface, using a streaming TCP socket design in conjunction with binary content:

- The binary interface of the APNs development environment is available through the URL `gateway.sandbox.push.apple.com` on port `2195`.
- The binary interface of the APNs production environment is available through the URL `gateway.push.apple.com` on port `2195`.
- The binary interface of the APNs feedback service is available through the URL `feedback.push.apple.com` on port `2196`.

For each interface, use TLS (or SSL) to establish a secured communication channel. The SSL certificate required for these connections is obtained from Apple's Member Center.

To establish a TLS session with APNs, an Entrust Secure CA root certificate must be installed on the provider’s server. If the server is running OS X, this root certificate is already in the keychain. On other systems the certificate might not be available.

Creating an APNs certificate for the app

The Apple Push Notification Certificate identifies the push notification service for a given mobile app. This certificate will be created from an App ID (a.k.a. Bundle ID) and is used by the APNs system to dispatch the notification message to the registered devices.

For more details, see APNs SSL certificate on page 2095.

Implementing the device tokens maintainer

To handle device registrations on the server side of your application, the same code base can be used for APNs and other token-based frameworks.

For more details, see Implementing a token maintainer on page 2611.

Implementing the push provider

The push provider will produce application notification messages that will be transmitted to the APNs service. The APNs service will then spread them to all registered mobile devices, identified by their device token.

To send notification messages, the push provider must build binary messages by using the com.APNS API, provided by the Web Services library, and send TCP message requests over SSL to the following URLs:

- `"tcps://gateway.sandbox.apple.com:2195"` (for development)
- `"tcps://gateway.push.apple.com:2195"` (for production)

> **Note:** In order to establish a secure connection to the APNs framework an SSL certificate needs to be defined in FGLPROFILE, as described in APNs SSL certificate on page 2095.

To send a notification message, the push provider must know the device tokens of the registered devices / applications.

> **Note:** A distinct notification message must be sent for each registered device.

The following example demonstrates how to implement a function to send an APNs notification message. The function takes a device token and a JSON object as parameters. First, build the binary data with the `com.APNS.EncodeMessage()` method, then POST the data with a `com.TCPRequest.doDataRequest()` method. In case of success, the TCP request timeout will occur (APNs service only responds immediately in case of error), then use the `com.TCPResponse.getDataResponse()` method, to get status information. See `com.APNS.EncodeMessage()` for more details about notification message creation.

```
IMPORT com
IMPORT security
IMPORT util

FUNCTION apns_send_notif_http(deviceTokenHexa, notif_obj)
    DEFINE deviceTokenHexa STRING,
           notif_obj util.JSONObject
    DEFINE req com.TCPRequest,
           resp com.TCPResponse,
           uuid STRING,
           ecode INTEGER,
           dt DATETIME YEAR TO SECOND,
           exp INTEGER,
           data, err BYTE,
           res STRING

    LOCATE data IN MEMORY
    LOCATE err IN MEMORY

    LET dt = CURRENT + INTERVAL(10) MINUTE TO MINUTE
    LET exp = util.Datetime.toSecondsSinceEpoch(dt)

    TRY
        LET req = com.TCPRequest.create( "tcps://
gateway.push.apple.com:2195" )
        CALL req.setKeepConnection(true)
        CALL req.setTimeout(2) # Wait 2 seconds for APNs to return error
 code
        LET uuid = security.RandomGenerator.createRandomString(4)
        CALL com.APNS.EncodeMessage(
                  data,
                  security.HexBinary.ToBase64(deviceTokenHexa),
                  notif_obj.toString(),
                  uuid,
```

```
                  exp,
                  10
             )
        IF LENGTH(data) > 2000 THEN
           LET res = "ERROR : APNS payload cannot exceed 2 kilobytes"
           RETURN res
        END IF
        CALL req.doDataRequest(data)
        TRY
            LET resp = req.getResponse()
            CALL resp.getDataResponse(err)
            CALL com.APNS.DecodeError(err) RETURNING uuid, ecode
            LET res = SFMT("APNS result: UUID: %1, Error code:
 %2",uuid,ecode)
        CATCH
            CASE STATUS
              WHEN -15553 LET res = "Timeout Push sent without error"
              WHEN -15566 LET res = "Operation failed :", SQLCA.SQLERRM
              WHEN -15564 LET res = "Server has shutdown"
              OTHERWISE   LET res = "ERROR :",STATUS
            END CASE
        END TRY
    CATCH
        LET res = SFMT("ERROR : %1 (%2)", STATUS, SQLCA.SQLERRM)
    END TRY
    RETURN res
END FUNCTION
```

The next code example implements a function that creates the JSON object defining notification content (payload). That object can be passed to the `apns_send_notif_http()` function described above:

```
FUNCTION apns_simple_popup_notif(notif_obj, msg_title, user_data,
 badge_number)
    DEFINE notif_obj util.JSONObject,
           msg_title, user_data STRING,
           badge_number INTEGER
    DEFINE aps_obj, data_obj util.JSONObject

    LET aps_obj = util.JSONObject.create()
    CALL aps_obj.put("alert", msg_title)
    CALL aps_obj.put("sound", "default")
    CALL aps_obj.put("badge", badge_number)
    CALL aps_obj.put("content-available", 1)
    CALL notif_obj.put("aps", aps_obj)

    LET data_obj = util.JSONObject.create()
    CALL data_obj.put("other_info", user_data)

    CALL notif_obj.put("custom_data", data_obj)

END FUNCTION
```

The `apns_simple_popup_notif()` and `apns_send_notif_http()` functions can then be used as follows:

```
IMPORT com
IMPORT util

MAIN
    DEFINE reg_ids DYNAMIC ARRAY OF STRING,
           notif_obj util.JSONObject,
           i INTEGER
```

```
    LET notif_obj = util.JSONObject.create()
    CALL gcm_simple_popup_notif(notif_obj, "This is my message!", 1)

    LET reg_ids[1] = "APA91bHun..."
    LET reg_ids[2] = "B4AA2q7xa..."
    ...
    FOR i=1 TO reg_ids.getLength()
        DISPLAY gcm_send_notif_http(reg_ids[i], notif_obj)
    END FOR

END MAIN
```

In order to use the tokens database maintained by a token maintainer program, your APNs push provider can collect device tokens as shown in the example below. Note that the dynamic array contains token ids and badge numbers:

```
FUNCTION apns_collect_tokens(reg_ids)
    DEFINE reg_ids DYNAMIC ARRAY OF RECORD
                       token STRING,
                       badge INTEGER
                   END RECORD
    DEFINE rec RECORD
               id INTEGER,
               sender_id VARCHAR(150),
               registration_token VARCHAR(250),
               badge_number INTEGER,
               app_user VARCHAR(50),
               reg_date DATETIME YEAR TO FRACTION(3)
           END RECORD,
           x INTEGER
    DECLARE c1 CURSOR FOR
      SELECT * FROM tokens
       WHERE sender_id IS NULL -- In case if GCM tokens remain in the db
    CALL reg_ids.clear()
    FOREACH c1 INTO rec.*
        LET x = reg_ids.getLength() + 1
        LET reg_ids[x].token = rec.registration_token
        LET reg_ids[x].badge = rec.badge_number
    END FOREACH
END FUNCTION
```

In order to handle badge numbers for each registered device, implement a function to update badge numbers in database:

```
FUNCTION save_badge_number(token, badge)
    DEFINE token STRING,
           badge INT
    UPDATE tokens SET
        badge_number = badge
    WHERE registration_token = token
END FUNCTION
```

The above functions can then be used to send a push message to all registered devices:

```
FUNCTION apns_send_message(msg_title, user_data)
    DEFINE msg_title, user_data STRING
    DEFINE reg_ids DYNAMIC ARRAY OF RECORD
                       token STRING,
                       badge INTEGER
                   END RECORD,
           notif_obj util.JSONObject,
```

```
            info_msg STRING,
            new_badge, i INTEGER
    CALL apns_collect_tokens(reg_ids)
    IF reg_ids.getLength() == 0 THEN
       RETURN "No registered devices..."
    END IF
    LET info_msg = "Send:"
    FOR i=1 TO reg_ids.getLength()
        LET new_badge = reg_ids[i].badge + 1
        CALL save_badge_number(reg_ids[i].token, new_badge)
        LET notif_obj = util.JSONObject.create()
        CALL apns_simple_popup_notif(notif_obj, msg_title, user_data,
 new_badge)
        LET info_msg = info_msg, "\n",
            apns_send_notif_http(reg_ids[i].token, notif_obj)
    END FOR
    RETURN info_msg
END FUNCTION
```

See also Provider Communication with Apple Push Notification Service.

Handle push notifications in mobile apps

To handle push notifications in mobile apps, the same code base can be used for APNs and other token-based frameworks.

For more details see Handling notifications in the mobile app on page 2617.

Implementing a token maintainer

The token maintainer is a BDL Web Services server program that handles push token registration from mobile apps.

Basics

In order to implement a push notification mechanism, you need to set up a server part (token maintainer and push notification server), in conjunction with a push notification framework such as Google Cloud Messaging (GCM) or Apple Push Notification service (APNs). In addition, you need to handle notification events in your mobile app. This section describes how to implement the token maintainer, the server program that maintains the list of registered devices (i.e. registration tokens for GCM or device tokens for APNs).

> **Note:** The max length of a push client token can vary according to the push framework provider. If you need to store registration tokens in a database, check the max size for a token and consider using a large column type such as `VARCHAR(250)`.

The same code base can be used for Android (using GCM) and iOS (using APNs) applications: The token maintainer will basically handle RESTful HTTP requests coming from the internet for token registration and token un-registration. For each of these requests, the program will insert a new record or delete an existing record in a dedicated database table.

> **Note:** The database used to store tokens must be created before starting the token maintainer program. By default, the program uses SQLite (dbmsqt) and the name of the database is "tokendb". To create this SQLite database, simply create an empty file with this name.

The push provider/server program can then query the tokens table to build the list of target devices for push notifications.

In the context of APNS, the token maintainer must also handle badge numbers for each registered device: When connsuming notification messages, the iOS app must inform the token maintainer that the badge number has changed. This function is implemented with the "`badge_number`" command.

The token maintainer is a Web Services server program which must be deployed behind a GAS to handle load balancing. You can, however, write code to test your program in development without a GAS.

The act of registering/unregistering push tokens is application specific: When registering tokens, you typically want to add application user information. Genero BDL allows you to implement a token maintainer in a simple way.

> **Note:** When executing this token maintainer program with APNs, you must pass the "APNS" command line argument to execute APNs feedback queries.

MAIN block and database creation

Start with the MAIN block, and the connection to a database. In this tutorial, we use SQLite as the database. The program will automatically create the database file and the tokens table if it does not yet exist.

```
...

MAIN
    CALL open_create_db()
    CALL handle_registrations()
END MAIN

FUNCTION open_create_db()
    DEFINE dbsrc VARCHAR(100),
           x INTEGER
    LET dbsrc = "tokendb+driver='dbmsqt'"
    CONNECT TO dbsrc
    WHENEVER ERROR CONTINUE
    SELECT COUNT(*) INTO x FROM tokens
    WHENEVER ERROR STOP
    IF SQLCA.SQLCODE<0 THEN
       CREATE TABLE tokens (
              id INTEGER NOT NULL PRIMARY KEY,
              sender_id VARCHAR(150),
              registration_token VARCHAR(250) NOT NULL UNIQUE,
              badge_number INTEGER NOT NULL,
              app_user VARCHAR(50) NOT NULL, -- UNIQUE
              reg_date DATETIME YEAR TO FRACTION(3) NOT NULL
       )
    END IF
END FUNCTION
```

Handling registration and unregistration requests

The next function is typical Web Service server code using the Web Services API to handle RESTful requests. Note that the TCP port is defined as a constant that is used to set FGLAPPSERVER automatically when not running behind the GAS:

```
IMPORT util
IMPORT com

CONSTANT DEFAULT_PORT = 9999

MAIN
    ...
    CALL handle_registrations()
END MAIN

FUNCTION handle_registrations()
    DEFINE req com.HTTPServiceRequest,
           url, method, version, content_type STRING,
```

```
           reg_data, reg_result STRING
   IF LENGTH(fgl_getenv("FGLAPPSERVER"))==0 THEN
      -- Normally, FGLAPPSERVER is set by the GAS
      DISPLAY SFMT("Setting FGLAPPSERVER to %1", DEFAULT_PORT)
      CALL fgl_setenv("FGLAPPSERVER", DEFAULT_PORT)
   END IF
   CALL com.WebServiceEngine.Start()
   WHILE TRUE
      TRY
         LET req = com.WebServiceEngine.getHTTPServiceRequest(20)
      CATCH
         IF STATUS==-15565 THEN
            CALL show_verb("TCP socket probably closed by GAS, stopping
process...")
            EXIT PROGRAM 0
         ELSE
            DISPLAY "Unexpected getHTTPServiceRequest() exception: ",
STATUS
            DISPLAY "Reason: ", SQLCA.SQLERRM
            EXIT PROGRAM 1
         END IF
      END TRY
      IF req IS NULL THEN -- timeout
         DISPLAY SFMT("HTTP request timeout...: %1", CURRENT YEAR TO
FRACTION)
         CALL check_apns_feedback()
         CALL show_tokens()
         CONTINUE WHILE
      END IF
      LET url = req.getURL()
      LET method = req.getMethod()
      IF method IS NULL OR method != "POST" THEN
         IF method == "GET" THEN
            CALL req.sendTextResponse(200,NULL,"Hello from token
maintainer...")
         ELSE
            DISPLAY SFMT("Unexpected HTTP request: %1", method)
            CALL req.sendTextResponse(400,NULL,"Only POST requests
supported")
         END IF
         CONTINUE WHILE
      END IF
      LET version = req.getRequestVersion()
      IF version IS NULL OR version != "1.1" THEN
         DISPLAY SFMT("Unexpected HTTP request version: %1", version)
         CONTINUE WHILE
      END IF
      LET content_type = req.getRequestHeader("Content-Type")
      IF content_type IS NULL
         OR content_type NOT MATCHES "application/json*" -- ;Charset=UTF-8
      THEN
         DISPLAY SFMT("Unexpected HTTP request header Content-Type: %1",
content_type)
         CALL req.sendTextResponse(400,NULL,"Bad request")
         CONTINUE WHILE
      END IF
      TRY
         CALL req.readTextRequest() RETURNING reg_data
      CATCH
         DISPLAY SFMT("Unexpected HTTP request read exception: %1", STATUS)
      END TRY
      LET reg_result = process_command(url, reg_data)
      CALL req.setResponseCharset("UTF-8")
      CALL req.setResponseHeader("Content-Type","application/json")
```

```
       CALL req.sendTextResponse(200,NULL,reg_result)
    END WHILE
END FUNCTION
```

Processing registration and unregistration commands

The next function is called when a RESTful request is to be processed. The URL will define the type of command to be executed by the server:

- If the URL contains "/token_maintainer/register", a new token must be inserted in the database.
- If the URL contains "/token_maintainer/unregister", an existing token must be deleted from the database.

```
FUNCTION process_command(url, data)
    DEFINE url, data STRING
    DEFINE data_rec RECORD
                sender_id VARCHAR(150),
                registration_token VARCHAR(250),
                app_user VARCHAR(50)
            END RECORD,
            p_id INTEGER,
            p_ts DATETIME YEAR TO FRACTION(3),
            result_rec RECORD
                status INTEGER,
                message STRING
            END RECORD,
            result STRING
    LET result_rec.status = 0
    TRY
       CASE
         WHEN url MATCHES "*token_maintainer/register"
           CALL util.JSON.parse( data, data_rec )
           SELECT id INTO p_id FROM tokens
                  WHERE registration_token = data_rec.registration_token
           IF p_id > 0 THEN
              LET result_rec.status = 1
              LET result_rec.message = SFMT("Token already registered:\n
 [%1]", data_rec.registration_token)
              GOTO pc_end
           END IF
           SELECT MAX(id) + 1 INTO p_id FROM tokens
           IF p_id IS NULL THEN LET p_id=1 END IF
           LET p_ts = util.Datetime.toUTC(CURRENT YEAR TO FRACTION(3))
           WHENEVER ERROR CONTINUE
           INSERT INTO tokens
               VALUES( p_id, data_rec.sender_id,
 data_rec.registration_token, data_rec.app_user, p_ts )
           WHENEVER ERROR STOP
           IF SQLCA.SQLCODE==0 THEN
              LET result_rec.message = SFMT("Token is now registered:\n
 [%1]", data_rec.registration_token)
           ELSE
              LET result_rec.status = -2
              LET result_rec.message = SFMT("Could not insert token in
 database:\n [%1]", data_rec.registration_token)
           END IF
         WHEN url MATCHES "*token_maintainer/unregister"
           CALL util.JSON.parse( data, data_rec )
           DELETE FROM tokens
                 WHERE registration_token = data_rec.registration_token
           IF SQLCA.SQLERRD[3]==1 THEN
```

```
            LET result_rec.message = SFMT("Token unregistered:\n [%1]",
 data_rec.registration_token)
         ELSE
            LET result_rec.status = -3
            LET result_rec.message = SFMT("Could not find token in
 database:\n [%1]", data_rec.registration_token)
         END IF
       WHEN url MATCHES "*token_maintainer/badge_number"
          CALL util.JSON.parse( data, data_rec )
          WHENEVER ERROR CONTINUE
            UPDATE tokens
               SET badge_number = data_rec.badge_number
             WHERE registration_token = data_rec.registration_token
          WHENEVER ERROR STOP
          IF SQLCA.SQLCODE==0 THEN
             LET result_rec.message = SFMT("Badge number update
 succeeded for Token:\n [%1]\n New value for badge number :[%2]\n",
 data_rec.registration_token, data_rec.badge_number)
          ELSE
             LET result_rec.status = -4
             LET result_rec.message = SFMT("Could not update badge number
 for token in database:\n [%1]", data_rec.registration_token)
          END IF
      END CASE
   CATCH
      LET result_rec.status = -1
      LET result_rec.message = SFMT("Failed to register token:\n [%1]",
 data_rec.registration_token)
   END TRY
LABEL pc_end:
   DISPLAY result_rec.message
   LET result = util.JSON.stringify(result_rec)
   RETURN result
END FUNCTION
```

Showing the current registered tokens

The following function is called after a WebServiceEngine timeout, when no request is to be processed. Its purpose is just to show the current list of registered tokens in a server log (stdout):

```
FUNCTION show_tokens()
   DEFINE rec RECORD
              id INTEGER,
              sender_id VARCHAR(150),
              registration_token VARCHAR(250),
              badge_number INTEGER,
              app_user VARCHAR(50),
              reg_date DATETIME YEAR TO FRACTION(3)
          END RECORD
   DECLARE c1 CURSOR FOR SELECT * FROM tokens ORDER BY id
   FOREACH c1 INTO rec.*
       IF rec.sender_id IS NULL THEN
          LET rec.sender_id = "(null)"
       END IF
       DISPLAY "   ", rec.id, ": ",
                     rec.app_user[1,10], " / ",
                     rec.sender_id[1,20],"... / ",
                     "(",rec.badge_number USING "<<<<&", ") ",
                     rec.registration_token[1,20],"..."
   END FOREACH
   IF rec.id == 0 THEN
      DISPLAY "No tokens registered yet..."
```

```
    END IF
END FUNCTION
```

APNs feedback checking

When using Apple Push Notification service, the device token maintainer can also handle device unregistration by querying the APNs feedback service. The APNs feedback service will provide the list of device tokens that are no longer valid because the app on the devices has unregistered.

> **Note:** When using the APNs feedback service, an SSL certificate needs to be defined in FGLPROFILE as described in APNs SSL certificate on page 2095.

To get the list of device tokens failed for remote notifications, send HTTP POST request to the following URL:

```
tcps://feedback.push.apple.com:2196
```

The token maintainer can use this service to clean up the token database.

The next function is called after a timeout when no request needs to be processed by the token maintainer:

```
FUNCTION check_apns_feedback()
    DEFINE req com.TCPRequest,
           resp com.TCPResponse,
           feedback DYNAMIC ARRAY OF RECORD
                       timestamp INTEGER,
                       deviceToken STRING
                    END RECORD,
           timestamp DATETIME YEAR TO FRACTION(3),
           token VARCHAR(250),
           i INTEGER,
           data BYTE

    IF arg_val(1)!="APNS" THEN RETURN END IF
    DISPLAY "Checking APNS feedback service..."

    LOCATE data IN MEMORY

    TRY
        LET req = com.TCPRequest.create( "tcps://
feedback.push.apple.com:2196" )
        CALL req.setKeepConnection(true)
        CALL req.setTimeout(2)
        CALL req.doRequest()
        LET resp = req.getResponse()
        CALL resp.getDataResponse(data)
        CALL com.APNS.DecodeFeedback(data,feedback)
        FOR i=1 TO feedback.getLength()
            LET timestamp =
 util.Datetime.fromSecondsSinceEpoch(feedback[i].timestamp)
            LET timestamp = util.Datetime.toUTC(timestamp)
            LET token = feedback[i].deviceToken
            DELETE FROM tokens
                  WHERE registration_token = token
                    AND reg_date < timestamp
        END FOR
    CATCH
        CASE STATUS
            WHEN -15553 DISPLAY "APNS feedback: Timeout: No feedback
 message"
            WHEN -15566 DISPLAY "APNS feedback: Operation failed :",
 SQLCA.SQLERRM
            WHEN -15564 DISPLAY "APNS feedback: Server has shutdown"
            OTHERWISE   DISPLAY "APNS feedback: ERROR :",STATUS
```

```
        END CASE
    END TRY
END FUNCTION
```

For more details about APNs feedback service, see The Feedback Service in Apple's APNs documentation.

Handling notifications in the mobile app

This topic describes how to handle push notification in the app running on mobile devices.

Basics

In order to implement a push notification mechanism, you need to set up a server part (token maintainer and push notification server), in conjunction with a push notification framework such as Google Cloud Messaging (GCM) or Apple Push Notification service (APNs). In Addition, you need to handle notification events in your mobile app. This section describes how to implement push notification in the app with the push notification API available in Genero BDL.

The same code base can be used to handle push notifications for Android (using GCM) and iOS (using APNs) devices. Only the content of the notification message will have to be processed with specific code, as the structure of the message differs according to standards defined by the push notification framework.

Genero API for push notifications

Genero BDL provides an API to handle push notification on mobile apps. Dedicated front calls are available to register to a push server, fetch push notification data, and unregister:

- registerForRemoteNotifications on page 1934
- getRemoteNotifications on page 1930
- unregisterFromRemoteNotifications on page 1939

To detect when a notification message arrives from the push server, a specific action called `notificationpushed` must be used by app code on a `ON ACTION` handler. This special action is referenced as a predefined action.

Android app permissions for GCM push notifications

Android apps using push notification services need specific permissions (Android manifest), such as:

- `android.permission.INTERNET`
- `android.permission.GET_ACCOUNTS`
- `android.permission.WAVE_LOCK`
- `com.google.android.c2dm.permission.RECEIVE`
- *`application-package-name`*`.permission.C2D_MESSAGE` where *application-package-name* is the Android package name of your app (for example, `com.mycompany.pushclient`)

Permissions will be automatically set when building the Android APK packages with the GMA build tool, according to the package name specified with the `--build-app-package-name` option.

See the GCM documentation for more details about required permissions for push notifications.

iOS app certificates for APNS push notifications

iOS apps must be created with an Apple certificate for development or distribution, linked to an App ID (or Bundle ID) with push notification enabled. The provisioning profile used when building the IPA must be linked to the App ID with push enabled. Certificate, provisioning and bundle id must be specified to the GMI build tool.

Handling push notification in the app

To handle push notifications in your mobile app, perform the following steps:

1. Register to the push service and get the registration token
2. Send the push notification token to your token maintainer
3. Handle notification events with the `notificationpushed` action
4. Eventually un-register from the push servers

1 - Registering to the push service and to the push provider

Register the app to the push notification service with the `"registerForRemoteNotifications"` front call.

- When using GCM, you must provide Sender ID to identify the GCM project.
- When using APNs, you can leave the Sender ID to NULL.

> **Note:** The app does not need to register for notification each time it is restarted: Even if the app is closed, the registration is still active until the `unregisterFromRemoteNotifications` front call is performed. At first execution, an app will typically ask if the user wants to get push notifications and register to the push service if needed. To disable push notification, apps usually implement an option that can be disabled (to unregister) and re-enabled (to register again) by the user. On Android, that the app must register for notification each time it is upgraded.
>
> **Important:**
>
> When an app restarts, if notifications are pending and the app has already registered for push notification in a previous execution, the `notificationpushed` action will be raised as soon as a dialog with the corresponding `ON ACTION` handler activates. The app should then perform a getRemoteNotifications on page 1930 front call as in the regular case, to get the pending notifications pushed to the device while the app was off.
>
> However, special consideration needs to be given to iOS devices. When push notification arrives for an iOS app that has not started, there is no mechanism to wake up the app and get the push data. Therefore, when the user starts the app from the springboard, there will never have any push data available. Depending on the context, implement the following programming patterns to solve this problem:
>
> 1. If the push notification contains a badge number, the app can verify if the badge is greater than 0 (with the `getBadgeNumber` front call) in order to perform a `getRemoteNotifications` front call. Even if there is no data available with the front call, the app should directly ask the server push provider to get last push data.
> 2. If the push notification does not contain badge numbers, the app should always perform a `getRemoteNotification` front call when it starts. If there is no push data available from the front call, the app should ask the server push provider if there is push data available. This is by the way also recommended when receiving a `notificationpushed` action during application life time.
> 3. If the user starts the app from the Notification Center, the app is launched with push data transmitted from the system, and the `notificationpushed` action is sent. The app should the perform, the `getRemoteNotifications` front call and get the push data.

The `registerForRemoveNotifications` front call will return a registration token for the app which will be used by the push server (a.k.a push provider).

- When using GCM, the returned identifier is the GCM "registration token".
- When using APNs, the returned identifier is the APNs "device token".

```
CONSTANT GCM_SENDER_ID = "<enter your GCM Sender ID ('' for APNs)>"

...
```

```
    LET rec.tm_host = "https://pushreg.example.orion"
    LET rec.tm__port = 4930
    LET rec.app_user = "mike"

    LET rec.registration_token = register(GCM_SENDER_ID, rec.app_user)

...

FUNCTION register(sender_id, app_user)
    DEFINE sender_id STRING,
           app_user STRING
    DEFINE registration_token STRING
    TRY
        CALL ui.Interface.frontCall(
                "mobile", "registerForRemoteNotifications",
                [ sender_id ], [ registration_token ] )
        IF tm_command( "register", sender_id, registration_token, app_user,
 0 ) < 0 THEN
           RETURN NULL
        END IF
    CATCH
        MESSAGE "Registration failed."
        RETURN NULL
    END TRY
    MESSAGE SFMT("Registration succeeded (token=%1)", registration_token)
    RETURN registration_token
END FUNCTION
```

2 - Sending a push notification token to your token maintainer

Once registered to the GCM or APNs service, the app must also register to the push server or push provider by sending the token obtained in step 1.

This is typically done by using a RESTFul HTTP POST, sending the token (along with additional application user information) to a dedicated server program that maintains the list of registered devices/ tokens.

The device token maintainer can be implemented in BDL as a Web Service program, as described in Implementing a token maintainer on page 2611.

In this tutorial, the `tm_command()` function implements token registration (as well as badge number handling for APNS):

```
IMPORT com
IMPORT util

...
    LET rec.tm_host = "https://pushreg.example.orion"
    LET rec.tm__port = 4930
...

FUNCTION tm_command( command, sender_id, registration_token, app_user,
 badge_number )
    DEFINE command STRING,
           sender_id STRING,
           registration_token STRING,
           app_user STRING,
           badge_number INTEGER
    DEFINE url STRING,
           json_obj util.JSONObject,
           req com.HTTPRequest,
           resp com.HTTPResponse,
           json_result STRING,
```

```
            result_rec RECORD
                           status INTEGER,
                           message STRING
                       END RECORD
    TRY
        LET url = SFMT( "http://%1:%2/token_maintainer/%3",
                        rec.tm_host, rec.tm_port, command )
        LET req = com.HTTPRequest.create(url)
        CALL req.setHeader("Content-Type", "application/json")
        CALL req.setMethod("POST")
        CALL req.setConnectionTimeOut(5)
        CALL req.setTimeOut(5)
        LET json_obj = util.JSONObject.create()
        CALL json_obj.put("sender_id", sender_id)
        CALL json_obj.put("registration_token", registration_token)
        CALL json_obj.put("app_user", app_user)
        CALL json_obj.put("badge_number", badge_number)
        CALL req.doTextRequest(json_obj.toString())
        LET resp = req.getResponse()
        IF resp.getStatusCode() != 200 THEN
           MESSAGE SFMT("HTTP Error (%1) %2",
                       resp.getStatusCode(),
                       resp.getStatusDescription())
           RETURN -2
        ELSE
           LET json_result = resp.getTextResponse()
           CALL util.JSON.parse(json_result, result_rec)
           IF result_rec.status >= 0 THEN
              RETURN 0
           ELSE
              MESSAGE SFMT("Notification maintainer message:\n %1",
 result_rec.message)
              RETURN -3
           END IF
        END IF
    CATCH
        MESSAGE SFMT("Failed to post token registration command: %1",
 STATUS)
        RETURN -1
    END TRY
END FUNCTION
```

When the app is declared as push notification client to the push server, continue with the normal program flow.

3 - Handling push notification events

To get and handle notification events, the current active dialog must implement the `notificationpushed` special action.

In the `ON ACTION` block for this action, query for notification messages by using the `"getRemoteNotifications"` front call, (passing the Sender ID as parameter when using GCM, for APNs the Sender ID must be NULL). This front call returns a JSON string containing a list of notification messages to be processed:

```
...
   DIALOG ...
       ...
       ON ACTION notificationpushed
          CALL handle_notification(sender_id)
       ...
   END DIALOG
...
```

```
FUNCTION handle_notification(sender_id)
    DEFINE sender_id STRING
    DEFINE notif_list STRING,
           notif_array util.JSONArray,
           notif_item util.JSONObject,
           notif_data util.JSONObject,
           notif_fld util.JSONObject,
           gcm_data, info STRING,
           i INTEGER
    CALL ui.Interface.frontCall(
             "mobile", "getRemoteNotifications",
             [ sender_id ], [ notif_list ] )
    TRY
        LET notif_array = util.JSONArray.parse(notif_list)
        IF notif_array.getLength() > 0 THEN
           CALL setup_badge_number(notif_array.getLength())
        END IF
        FOR i=1 TO notif_array.getLength()
            LET info = NULL
            LET notif_item = notif_array.get(i)
            -- Try APNs msg format
            LET notif_data = notif_item.get("custom_data")
            IF notif_data IS NULL THEN
               -- Try GCM msg format
               LET gcm_data = notif_item.get("data")
               IF gcm_data IS NOT NULL THEN
                  LET notif_data = util.JSONObject.parse(gcm_data)
               END IF
            END IF
            IF notif_data IS NOT NULL THEN
               LET info = notif_data.get("other_info")
            END IF
            IF info IS NULL THEN
               LET info = "Unexpected message format"
            END IF
            MESSAGE CURRENT HOUR TO SECOND, ": ", info
            SLEEP 1
        END FOR
    CATCH
        ERROR "Could not extract notification info"
    END TRY
END FUNCTION
```

When using APNS, the app must handle the badge numbers attached to the device token. The app must:

1. Query the current badge number with the `getBadgeNumber` front call.
2. Compute the new badge number according to the number of notifications consumed.
3. Reset the badge number with the setBadgeNumber front call.
4. Inform the token maintainer to sync the badge number in the central database.

The following function handles badge numbers for the app:

```
FUNCTION setup_badge_number(consumed)
    DEFINE consumed INTEGER
    DEFINE badge_number INTEGER
    TRY -- If the front call fails, we are not on iOS...
        CALL ui.Interface.frontCall("ios", "getBadgeNumber", [],
 [badge_number])
    CATCH
        RETURN
    END TRY
    IF badge_number>0 THEN
```

```
       LET badge_number = badge_number - consumed
    END IF
    CALL ui.Interface.frontCall("ios", "setBadgeNumber", [badge_number], [])
    IF tm_command( "badge_number",
                   rec.sender_id, rec.registration_token,
                   rec.user_name, badge_number) < 0 THEN
       ERROR "Could not send new badge number to token maintainer."
       RETURN
    END IF
END FUNCTION
```

4 - Unregistering the app from push notification

If the app no longer wants to get push notifications, unregister from the push provider (using a RESTful POST, in the `regunreg_token()` function), and unregister from the push service by using the `"unregisterFromRemoteNotifications"` front call.

- When using GCM, you must pass the GCM Sender ID as parameter.
- When using APNs, the parameter must be NULL.

```
...

    LET rec.tm_host = "https://pushreg.example.orion"
    LET rec.tm__port = 4930

    CALL unregister(GCM_SENDER_ID, rec.registration_token, rec.app_user)

...

FUNCTION unregister(sender_id, registration_token, app_user)
    DEFINE sender_id STRING,
           registration_token STRING,
           app_user STRING
    IF tm_command( "unregister", sender_id, registration_token, app_user,
 0 ) < 0 THEN
       RETURN
    END IF
    TRY
        CALL ui.Interface.frontCall(
                "mobile", "unregisterFromRemoteNotifications",
                [ sender_id ], [ ] )
    CATCH
        MESSAGE "Un-registration failed (broacast service)."
        RETURN
    END TRY
    MESSAGE "Un-registration succeeded"
END FUNCTION
```