FOUR Js
The Power of Simplicity

Genero BDL Tutorial

Contents

Genero BDL Tutorial Summary

If you are a developer new to Genero and the Genero Business Development Language (BDL), this tutorial is designed for you.

This tutorial explains concepts and provides code examples for common business-related tasks. The only prerequisite knowledge is familiarity with relational databases and SQL.

The chapters contain a series of programs that range in complexity from displaying a database row to more advanced topics, such as handling arrays and master/detail relationships. Each chapter has a general discussion of the features and programming techniques used in the example programs, with annotated code samples. The examples in later chapters build on concepts and functions explained in earlier chapters.

These programs have the BDL keywords in uppercase letters; this is a convention only. The line numbers in the programs are for reference only; they are not a part of the BDL code.

To run the example programs or try out the programming techniques described in this tutorial, see Testing the Example Programs.

For an overview of Genero BDL, refer to the section *Overview of Genero BDL* in the *Genero Business Development Language User Guide*.

Testing the Example Programs

The program examples used in this tutorial are packaged with Genero Studio. To run the programs you will need a complete install of the Genero product suite.

The Genero product suite includes:

- Genero: The Genero Business Development Language with its compiler and virtual machine
- Genero Studio: The integrated Development environment for the Genero product suite
- Genero Report Writer: The enterprise graphical reporting tool

A Genero Project (`4pw`) manages the source files and properties for building and executing the program examples in the tutorial. You can follow the steps below to work with the examples in the BDLTutorial project in a convenient and visual way using Genero Studio.

Genero installs with a preconfigured SQLite database that you can use for tutorial examples that require database access. An `fglprofile` configuration file with the entries needed to connect to the SQLite database is also provided.

Perform these steps to open the BDLTutorial project in Genero Studio and explore the project structure.

1. Launch Genero Studio from the taskbar or Start Menu.
2. From the **Welcome Page**, **Tutorials & Samples** tab, select the **BDLTutorial** project. This opens the BDLTutorial project file in the Projects view.

Figure 1: Selecting the BDLTutorial project in Genero Studio

3. In the Project view, expand the nodes of the project tree to view group nodes associated with each chapter of the tutorial.
4. Expand the **chap02** group node and you will find application nodes for three programs: `connectdb`, `debugit`, and `simple`. Application nodes contain the application source files (modules).
5. Expand the **simple** application node to see the single BDL source module for the application: `simple.4gl`.
6. Double-click `simple.4gl` to view the source code in Code Editor, a programming-oriented editor included with Genero Studio. `simple.4gl` will be the first example analyzed in chapter two, where you will find instructions on compiling and executing an application in Genero Studio as well as from the Command line.

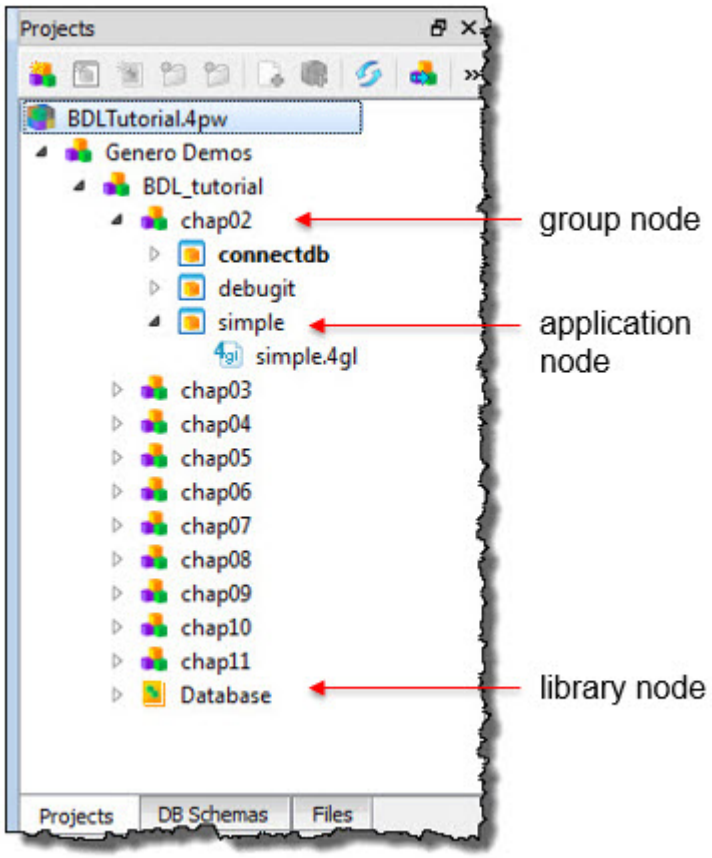

Figure 2: Viewing the BDLTutorial project structure in the Project view.

7. Close the `simple.4gl` program by selecting the close symbol

   or leave the file open in Code Editor in preparation for Chapter 2.

You can learn more about projects and the Project view in the *Project Manager* section of the *Genero Studio User Guide*.

For more information about Code Editor, see the *Code Editor* section of the *Genero Studio User Guide*.

Tutorial Chapters

Each chapter illustrates Genero Business Development Language (BDL) concepts with program examples.

Table 1: Tutorial chapters

Chapter	Description
Tutorial Chapter 1: Overview on page 12	This chapter provides an overview of the Tutorial and a description of the database schema and sample data used for the example programs.
Tutorial Chapter 2: Using BDL on page 17	The topics in this chapter illustrate the structure of a BDL program and some of the BDL statements that perform some common tasks - display a text message to the screen, connect to a database and retrieve data, define variables, and pass variables between functions.

Chapter	Description
Tutorial Chapter 3: Displaying Data (Windows/ Forms) on page 24	This chapter illustrates opening a window that contains a form to display information to the user. An SQL statement is used to retrieve the data from a database table. A form specification file is defined to display the values retrieved. The actions that are available to the user are defined in the source code, tied to buttons that display on the form.
Tutorial Chapter 4: Query by Example on page 35	The program in this chapter allows the user to search a database by entering criteria in a form. The search criteria is used to build an SQL `SELECT` statement to retrieve the desired database rows. A cursor is defined in the program, to allow the user to scroll back and forth between the rows of the result set. Testing the success of the SQL statements and handling errors is illustrated.
Tutorial Chapter 5: Enhancing the Form on page 52	Program forms can be displayed in a variety of ways. This chapter illustrates adding a toolbar or a topmenu (pull-down menu) by modifying the form specification file, changing the window's appearance, and disabling/enabling actions. The example programs in this chapter use some of the action defaults defined by Genero BDL to standardize the presentation of common actions to the user.
Tutorial Chapter 6: Add, Update and Delete on page 62	This program allows the user to insert/ update/delete rows in the customer database table. Embedded SQL statements (`UPDATE`/`INSERT`/`DELETE`) are used to update the table, based on the values stored in the program record. SQL transactions, concurrency, and consistency are discussed. A dialog window is displayed to prompt the user to verify the deletion of a row.
Tutorial Chapter 7: Array Display on page 74	The example in this chapter displays multiple `customer` records at once. The `disparray` program defines a program array to hold the records, and displays the records in a form containing a table and a screen array. The example program is then modified to dynamically fill the array as needed. This program illustrates a library function - the example is written so it can be used in multiple programs, maximizing code reuse.
Tutorial Chapter 8: Array Input on page 85	The program in this chapter allows the user to view and change a list of records displayed on a form. As each record in the program array is added, updated, or deleted, the program logic makes corresponding changes in the rows of the corresponding database table.
Tutorial Chapter 9: Reports on page 96	This program generates a simple report of the data in the `customer` database table. The two parts of a report, the report driver logic and the report

Chapter	Description
	definition are illustrated. A technique to allow a user to interrupt a long-running report is shown.
Tutorial Chapter 10: Localization on page 106	Localization support and localized strings allow you to internationalize your application using different languages, and to customize it for specific industry markets in your user population. This chapter illustrates the use of localized strings in your programs.
Tutorial Chapter 11: Master/Detail on page 115	The form used by the program in this chapter contains fields from both the `orders` and `items` tables in the `custdemo` database, illustrating a master-detail relationship. Since there are multiple items associated with a single order, the rows from the `items` table are displayed in a table on the form. This chapter focuses on the master/detail form and the unique features of the corresponding program.
Tutorial Chapter 12: Changing the User Interface Dynamically on page 135	This chapter focuses on using the classes and methods in the `ui` package of built-in classes to modify the user interface at runtime. Among the techniques illustrated are hiding or disabling form items; changing the text, style or image associated with a form item; loading a combobox from a database table; and adding toolbars and topmenus dynamically.
Tutorial Chapter 13: Master/Detail using Multiple Dialogs on page 149	This chapter shows how to implement order and items input in a unique `DIALOG` statement. In chapter 11 the order input is detached from the items input. The code example in chapter 13 makes both order and item input fields active at the same time, which is more natural in GUI applications.

Tutorial Chapter 1: Overview

This chapter provides an overview of the Tutorial and a description of the database schema and sample data used for the example programs.

This chapter covers concepts from the section *Genero BDL concepts* in the *Genero Business Development Language User Guide*.

Overview

Especially well-suited for large-scale, database-intensive business applications, Genero Business Development Language (BDL) is a reliable, easy-to-learn high-level programming language.

BDL allows application developers to:

- express business logic in a clear yet powerful syntax
- use SQL statements for database access to any of the supported databases
- localize your application to follow a specific language or cultural rules
- define user interfaces in an abstract, platform-independent manner
- define Presentation Styles to customize and standardize the appearance of the interface
- manipulate the user interface at runtime, as a tree of objects

The separation of business logic, user interface, and deployment provides maximum flexibility.

- The business logic is written in text files (`.4gl` source code modules) that interact with separate form files defining the user interface.
- Actions defined in the business logic are tied to action views (buttons, menu items, toolbar icons) in the form definition files, and respond to user interaction statements in the source code.
- Compiling a form definition file translates it into XML, which is used to display the user interface to various Genero clients running on different platforms.

You can write once, deploy anywhere - one production release supports all major versions of UNIX™, Linux™, Windows™, and Mac OS X.

The BDL Language

Genero Business Development Language (BDL) is a program language designed to write an interactive database application, as a set of programs that handle the interaction between a user and a database.

The Genero Business Development Language includes:

- Program flow control
- Conditional logic
- SQL statement support
- Connection management
- Error handling
- Localized strings

Dynamic SQL management allows you to execute any SQL statement that is valid for your database version, in addition to those that are included as part of the language. The statement can be hard coded or created at runtime, with or without SQL parameters, returning or not returning a result set.

High-level BDL user interaction statements substitute for the many lines of code necessary to implement common business tasks, mediating between the user and the user interface in order to:

- Provide a selection of actions to the user (`MENU`)
- Allow the user to enter database search criteria on a form (`CONSTRUCT`)
- Display information from database tables (`DISPLAY`, `DISPLAY ARRAY`)
- Allow the user to modify the contents of database tables (`INPUT`, `INPUT ARRAY`)

Multiple dialogs allow a Genero program to handle interactive statements in parallel.

In addition, built-in classes and methods, and built-in functions are provided to assist you in your program development.

The BDL Tutorial

The chapters in this tutorial describe the basic functionality of Genero BDL.

Annotated code examples in each chapter guide you through the steps to implement the features discussed. In addition, complete source code programs of the examples are available for download, contact your support channel to get the links. See Tutorial Chapters on page 9 for a description of each chapter.

The example programs interact with a demo database, the `custdemo` database, containing store and order information for a fictional retail chain.

If you wish to test the example programs on your own system, see Testing the Programs for information about the software and sample data that must be installed and configured.

The Example Database (custdemo)

The following SQL statements create the tables for the `custdemo` database.

These statements are in the file `custdemo.sql` in the Tutorial subdirectory of the documentation.

```
create table customer(
    store_num     integer not null,
    store_name    char(20) not null,
    addr          char(20),
    addr2         char(20),
    city          char(15),
    state         char(2),
    zip-code       char(5),
    contact_name  char(30),
    phone         char(18),
    primary key (store_num)
);
create table orders(
    order_num     integer not null,
    order_date    date not null,
    store_num     integer not null,
    fac_code      char(3),
    ship_instr    char(10),
    promo         char(1) not null,
    primary key (order_num)
);
create table factory(
    fac_code      char(3) not null,
```

```
    fac_name      char(15) not null,
    primary key (fac_code)
);
create table stock(
    stock_num     integer not null,
    fac_code      char(3) not null,
    description   char(15) not null,
    reg_price     decimal(8,2) not null,
    promo_price   decimal(8,2),
    price_updated date,
    unit          char(4) not null,
    primary key (stock_num)
);
create table items(
    order_num     integer not null,
    stock_num     integer not null,
    quantity      smallint not null,
    price         decimal(8,2) not null,
    primary key (order_num, stock_num)
);
create table state(
    state_code char(2) not null,
    state_name char(15) not null,
    primary key (state_code)
);
```

The Sample Data

The `custdemo` database contains the following sample data.

Customer table

```
101|Bandy's Hardware|110 Main| |Chicago|IL|60068|Bob Bandy|
630-221-9055|
102|The FIX-IT Shop|65W Elm Street Sqr.| |Madison|WI|65454| |
630-34343434|
103|Hill's Hobby Shop|553 Central Parkway| |Eau Claire|WI|54354|Janice
 Hilstrom|
666-4564564|
104|Illinois Hardware|123 Main Street| |Peoria|IL|63434|Ramon Aguirra|
630-3434334|
105|Tools and Stuff|645W Center Street| |Dubuque|IA|54654|Lavonne Robinson|
630-4533456|
106|TrueTest Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael
 Mazukelli|
640-3453456|
202|Fourth Ill Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael
 Mazukelli|
640-3453456|
203|2nd Hobby Shop|553 Central Parkway| |Eau Claire|WI|54354|Janice
 Hilstrom|
666-4564564|
204|2nd Hardware|123 Main Street| |Peoria|IL|63434|Ramon Aguirra|
630-3434334|
205|2nd Stuff|645W Center Street| |Dubuque|IA|54654|Lavonne Robinson|
630-4533456|
206|2ndTest Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael
 Mazukelli|
640-3453456|
302|Third FIX-IT Shop|65W Elm Street Sqr.| |Madison|WI|65454| |
630-34343434|
```

```
303|Third Hobby Shop|553 Central Parkway| |Eau Claire|WI|54354|Janice
 Hilstrom|
666-4564564|
304|Third IL Hardware|123 Main Street| |Peoria|IL|63434|Ramon Aguirra|
630-3434334|
305|Third and Stuff|645W Center Street| |Dubuque|IA|54654|Lavonne Robinson|
630-4533456|
306|Third Hardware|6123 N. Michigan Ave| |Chicago|IL|60104|Michael
 Mazukelli|
640-3453456|
```

Orders table

```
1|04/04/2003|101|ASC|FEDEX|N|
2|06/06/2006|102|ASC|FEDEX|Y|
3|06/10/2006|103|PHL|FEDEX|Y|
4|06/10/2006|104|ASC|FEDEX|Y|
5|07/06/2006|101|ASC|FEDEX|Y|
6|07/16/2006|105|ASC|FEDEX|Y|
7|08/04/2006|104|PHL|FEDEX|Y|
8|08/16/2006|101|ASC|FEDEX|Y|
9|08/23/2006|101|ASC|FEDEX|Y|
10|09/06/2006|106|PHL|FEDEX|Y|
```

Items table

```
1|456|10|5.55|
1|310|5|12.85|
1|744|60|250.95|
2|456|15|5.55|
2|310|2|12.85|
3|323|2|0.95|
4|744|60|250.95|
4|456|15|5.55|
5|456|12|5.55|
5|310|15|12.85|
5|744|6|250.95|
6|456|15|5.55|
6|310|2|12.85|
7|323|10|0.95|
8|456|10|5.55|
8|310|15|12.85|
9|744|20|250.95|
10|323|200|0.95|
```

Stock table

```
456|ASC|lightbulbs|5.55|5.0|01/16/2006|ctn|
310|ASC|sink stoppers|12.85|11.57|06/16/2006|grss|
323|PHL|bolts|0.95|0.86|01/16/2006|20/b|
744|ASC|faucets|250.95|225.86|01/16/2006|6/bx|
```

Factory table

```
ASC|Assoc. Std. Co.|
PHL|Phelps Lighting|
```

State table

```
IL|Illinois|
IA|Iowa|
WI|Wisconsin|
```

Tutorial Chapter 2: Using BDL

The topics in this chapter illustrate the structure of a BDL program and some of the BDL statements that perform some common tasks - display a text message to the screen, connect to a database and retrieve data, define variables, and pass variables between functions.

A simple BDL program

This simple example displays a text message to the screen, illustrating the structure of a BDL program.

Genero BDL source code is written as text in a source module (a file with an extension of `.4gl`). Because Genero BDL is a structured programming language as well as a 4th generation language, executable statements can appear only within logical sections of the source code called program blocks. This can be the `MAIN` statement, a `FUNCTION` statement, or a `REPORT` statement. (Reports are discussed in Chapter 9.)

Execution of any program begins with the special, required program block `MAIN`, delimited by the keywords `MAIN` and `END MAIN`. The source module that contains `MAIN` is called the main module.

The `FUNCTION` statement is a unit of executable code, delimited by `FUNCTION` and `END FUNCTION`, that can be called by name. In a small program, you can write all the functions used in the program in a single file. As programs grow larger, you will usually want to group related functions into separate files, or source modules. Functions are available on a global basis. In other words, you can reference any function in any source module of your program.

Although the language keywords in this example and throughout the tutorial are in all-capitals, this is just a convention used in these documents. You may write keywords in any combination of capitals and lowercase you prefer.

You can begin a comment that terminates at the end of the current line with a pair of minus signs (--) or #. Curly braces {} can be used to delimit comments that occupy multiple lines.

The following example is a small but complete Genero BDL program named `simple.4gl`.

```
01 -- simple.4gl
02
03 MAIN
04     CALL sayIt()
05 END MAIN
06
07 FUNCTION sayIt()
08    DISPLAY "Hello, world!"
09 END FUNCTION
```

Note:

- Line `01` simply lists the filename as a comment , which will be ignored by BDL.
- Line `03` indicates the start of the `MAIN` program block.
- Line `04` Within the `MAIN` program block, the `CALL` statement is used to invoke the function named `sayIt`. Although no arguments are passed to the function `sayIt`, the empty parentheses are required. Nothing is returned by the function.

- Line `05` defines the end of the `MAIN` program block. When all the statements within the program block have been executed the program will terminate automatically.
- Line `07` indicates the start of the function `sayIt` .
- Line `08` uses the `DISPLAY` statement to display a text message, enclosed within double quotes, to the user. Because the program has not opened a window or form, the message is displayed on the command line.
- Line `09` indicates the end of the function. After the message is displayed, control in the program is returned to the `MAIN` function, to line `05`, the line immediately following the statement invoking the function. As there are no additional statements to be executed (`END MAIN` has been reached), the program terminates.

Compiling and Executing the Program

BDL programs are made up of a single module, or modules, containing the program functions. You can compile and execute programs in Genero Studio or use command line tools if you prefer.

From Genero Studio

The **Execute** option in the Genero Studio Project view will compile and link files in the specified application node if necessary before executing the application. You can also compile individual modules or build an application (compile and link files) as independent steps.

To compile and execute the **simple** program in Genero Studio:

1. In the Project view, expand the **BDLTutorial** project and find the **chap02** group.
2. Expand the **chap02** group, right-click on the **simple** application node and select **Execute**.

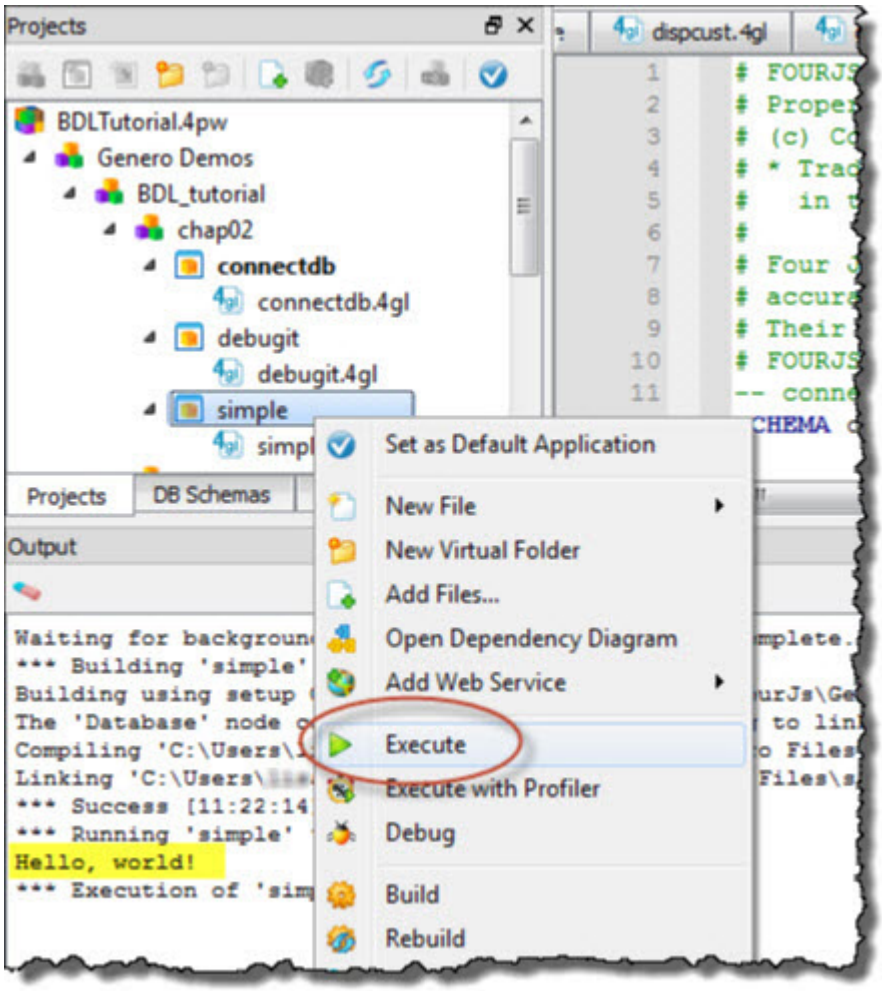

Figure 3: Using the Execute option to compile and execute the simple program

From the command line

The following tools can be used to compile and execute the `simple` program from the command line.

1. Compile the single module program:

```
fglcomp simple.4gl
```

2. Execute the program:

```
fglrun simple.42m
```

Tip:

1. You can compile and run a program without specifying the file extensions:

```
fglcomp simple
fglrun simple
```

You can do this in one command line, adding the `-M` option for errors:

```
fglcomp -M simple && fglrun simple
```

Debugging a BDL Program

You can use the Genero graphical debugger or the command line debugger to search for programming errors.

The command line debugger is integrated in the runtime system. You typically start a program in debug mode by passing the `-d` option to `fglrun` .

The following lines illustrate a debug session with the `simple` program:

```
fglrun -d simple
(fgldb) break main
Breakpoint 1 at 0x00000000: file simple.4gl, line 2.
(fgldb) run
Breakpoint 1, main() at simple.4gl:2
2         CALL sayIt()
(fgldb) step
sayit() at simple.4gl:6
6         DISPLAY "Hello, world!"
(fgldb) next
Hello, world!
7     END FUNCTION -- sayIt (fgldb) continue
Program existed normally.
(fgldb) quit
```

This chapter covers concepts from the section *The debugger* in the *Genero Business Development Language User Guide*.

The "Connect to database" Program

This program illustrates connecting to a database and retrieving data, defining variables, and passing variables between functions.

A row from the `customer` table of the `custdemo` example database is retrieved by an SQL statement and displayed to the user.

Connecting to the database

To connect to a database server, most database engines require a name to identify the server, a name to identify the database entity, a user name and a password.

Connecting through the Open Database Interface, the database can be specified directly, and the specification will be used as the data source. Or, you can define the database connection parameters indirectly in the `fglprofile` configuration file, and the database specification will be used as a key to read the connection information from the file. This technique is flexible; for example, you can develop your application with the database name "custdemo" and connect to the real database "custdemo1" in a production environment.

The `CONNECT` instruction opens a session in multi-session mode, allowing you to open other connections with subsequent `CONNECT` instructions (to other databases, for example). The `DISCONNECT` instruction can be used to disconnect from specific sessions, or from all sessions. The end of a program disconnects all sessions automatically.

The username and password can be specified in the `CONNECT` instruction, or defaults can be defined in the `fglprofile` file. Otherwise, the user name and password provided to your operating system will generally be used for authentication.

```
CONNECT TO "custdemo"
```

Variable definition

A Variable contains volatile information of a specific BDL data type. Variables must be declared before you use them in your program, using the `DEFINE` statement. After definition, variables have default values based on the data type.

```
DEFINE cont_ok INTEGER
```

You can use the `LIKE` keyword to declare a variable that has the same data type as a specified column in a database schema. 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, not at runtime. Make sure that your schema files correspond exactly to the production database.

```
DEFINE store_name LIKE customer.store_name
```

Genero BDL allows you to define structured variables as records or arrays. Examples of this are included in later chapters.

Variable scope

Variables defined in a `FUNCTION` , `REPORT` or `MAIN` program block have local scope (are known only within the program block). `DEFINE` must precede any executable statements within the same program block. A variable with local scope can have its value set and can be used only within the function in which it is defined.

A Variable defined with module scope can have its value set and can be used in any function within a single source-code module. The `DEFINE` statement must appear at the top of the module, before any program blocks.

A Variable defined with global scope can have its value set and can be used in any function within any modules of the same program.

For a well-structured program and ease of maintenance, we recommend that you use module variables instead of global when you need persistent data storage. You can include get/set functions in the module to make the value of the variable accessible to functions in other modules.

A compile-time error occurs if you declare the same name for two variables that have the same scope.

Passing variables

Functions can be invoked explicitly using the `CALL` statement. Variables can be passed as arguments to a function when it is invoked. The parameters can be variables, literals, constants, or any valid expressions. Arguments are separated by a comma. If the function returns any values, the `RETURNING` clause of the `CALL` statement assigns the returned values to variables in the calling routine. The number of input and output parameters is static.

The function that is invoked must have a `RETURN` instruction to transfer the control back to the calling function and pass the return 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. If the function returns only one unique value, it can be used as a scalar function in an expression.

```
CALL myfunc()
CALL newfunc(var1) RETURNING var2, var3
LET var2 = anotherfunc(var1)
IF testfunc1(var1) == testfunc2(var1) THEN ...
```

Retrieving data from a database

Using Static SQL, an embedded SQL `SELECT` statement can be used to retrieve data from a database table into program variables. If the `SELECT` statement returns only one row of data, you can write it directly as a procedural instruction, using the `INTO` clause to provide the list of variables where the column values

will be fetched. If the SELECT statement returns more than one row of data, you must declare a database cursor to process the result set.

Example: connectdb.4gl

This program connects to the custdemo database, selects the store name from the customer table and displays it to the user.

> **Note:** The line numbers shown in the examples in this tutorial are not part of the BDL code; they are used here so specific lines can be easily referenced. The BDL keywords are shown in uppercase, as a convention only.

Program **connectdb.4gl**:

```
01 -- connectdb.4gl
02 SCHEMA custdemo
03
04 MAIN
05   DEFINE
06    m_store_name LIKE customer.store_name
07
08   CONNECT TO "custdemo"
09
10   CALL select_name(101)
11       RETURNING m_store_name
12   DISPLAY m_store_name
13
14   DISCONNECT CURRENT
15
16 END MAIN
17
18 FUNCTION select_name(f_store_num)
19   DEFINE
20    f_store_num  LIKE customer.store_num,
21    f_store_name LIKE customer.store_name
22
23   SELECT store_name INTO f_store_name
24      FROM customer
25      WHERE store_num = f_store_num
26
27   RETURN f_store_name
28
29 END FUNCTION  -- select_name
```

Note:

- Line 02 The SCHEMA statement is used to define the database schema files to be used as custdemo. The LIKE syntax has been used to define variables in the module.
- Lines 05 and 06 Using DEFINE the local variable m_store_name is declared as being LIKE the store_name column; that is, it has the same data type definition as the column in the customer table of the custdemo database.
- Line 08 A connection in multi-session mode is opened to the custdemo database, with connection parameters defined in the fglprofile configuration file. Once connected to the database server, a current database session is started. Any subsequent SQL statement is executed in the context of the current database session.
- Line10 The select_name function is called, passing the literal value 101 as an argument. The function returns a value to be stored in the local variable m_store_name.
- Line 12 The value of m_store_name is displayed to the user on the standard output.
- Line 14 The DISCONNECT instruction disconnects you from the current session. As there are no additional lines in the program block, the program terminates.

- Line 18 Beginning of the definition of the function `select_name`. The value "101" that is passed to the function will be stored in the local variable `f_store_num`.
- Lines 19 thru 21 Defines multiple local variables used in the function, separating the variables listed with a comma. Notice that a variable must be declared with the same name and data type as the parameter listed within the parenthesis in the function statement, to accept the passed value.
- Lines 23 thru 25Contains the embedded `SELECT ... INTO` SQL statement to retrieve the store name for store number 101. The store name that is retrieved will be stored in the `f_store_name` local variable. Since the store number is unique, the `WHERE` clause ensures that only a single row will be returned.
- Line 27 The `RETURN` statement causes the function to terminate, returning the value of the local variable `f_store_name`. The number of variables returned matches the number declared in the `RETURNING` clause of the `CALL` statement invoking the function. Execution of the program continues with line 12.

The database schema file

This program requires a database schema file because of the use of the `LIKE` keyword when defining the variable `m_store_name`. The database schema contains the definition of the database tables and columns and is used to centralize column data types to define program variables. The schema file for the BDLTutorial has already been extracted from the `custdemo` database and is used at compile time.

To learn more about database schema files see *Database schema* in the *Genero Business Development Language User Guide*.

Compiling and executing the program

You can compile and execute the connectdb application using the `Execute` option in the Project view of Genero Studio or use the command line options.

1. Compile the single module program:

```
fglcomp connectdb.4gl
```

2. Execute the program:

```
fglrun connectdb.42m
```

Tutorial Chapter 3: Displaying Data (Windows/Forms)

This chapter illustrates opening a window that contains a form to display information to the user. An SQL statement is used to retrieve the data from a database table. A form specification file is defined to display the values retrieved. The actions that are available to the user are defined in the source code, tied to buttons that display on the form.

Application Overview

This example program opens a window containing a form to display information to the user.

The appearance of the form is defined in a separate form definition file. The program logic to display information on the form is written in the `.4gl` program module. The same form file can be used with different applications.

The options to retrieve data or exit are defined as actions in a `MENU` statement in the `.4gl` file. By default, push buttons are displayed on the form corresponding to the actions listed in the `MENU` statement. When the user presses the **query** button, the code listed for the action statement is executed - in this case, an SQL `SELECT` statement retrieves a single row from the `customer` table and displays it on the form.

A `FORM` can contain form fields for entering and displaying data; explanatory text (labels); and other form objects such as buttons, topmenus (dropdown menus), toolbar icons, folders, tables, and checkboxes. Form objects that are associated with an action are called action views. Messages providing information to the user can be displayed on the form.

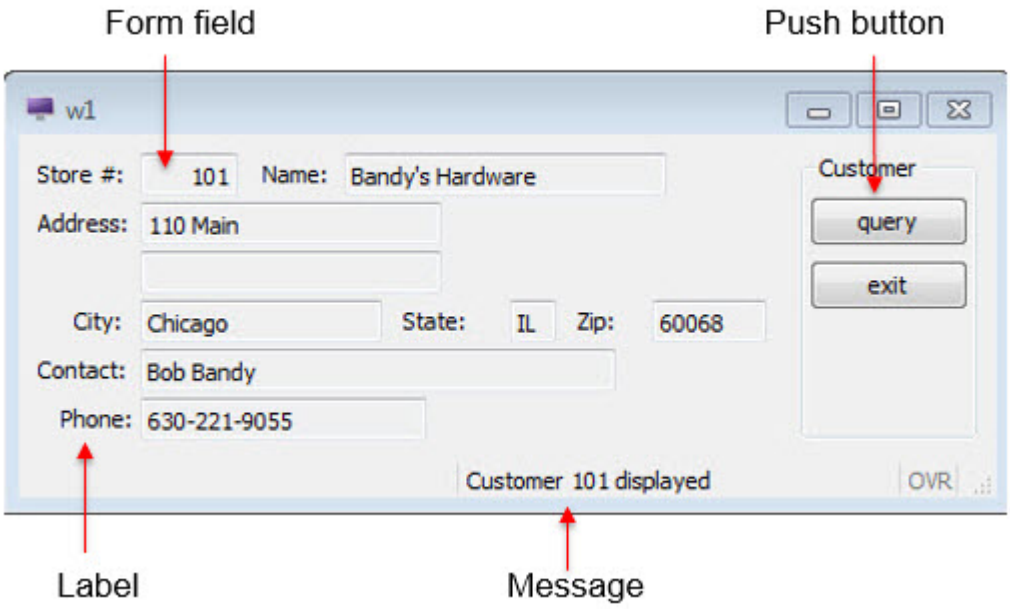

Figure 4: Interface as displayed on Windows™ platforms

The .4gl File - Opening Windows and Forms

A program creates a window with the `OPEN WINDOW` instruction, and destroys a window with the `CLOSE WINDOW` instruction.

The `OPEN WINDOW ... WITH FORM` instruction can be used to automatically open a window containing a specified form:

```
OPEN WINDOW custwin WITH FORM "custform"
```

When you are using a graphical front end, windows are created as independent resizable windows. By default windows are displayed as normal application windows, but you can specify a Presentation Style. The standard window styles are defined in the default Presentation Style file (`FGLDIR/lib/default.4st`):

If the `WITH FORM` option is used in opening a window, the `CLOSE WINDOW` statement closes both the window and the form.

```
CLOSE WINDOW custwin
```

When the runtime system starts a program, it creates a default window named `SCREEN`. This default window can be used as another window, but it can be closed if not needed.

```
CLOSE WINDOW SCREEN
```

> **Note:** The appropriate Genero Front-end Client must be running for the program to display the window and form.

The .4gl File - Interacting with the User

Your form can display options to the user using action views - buttons, dropdown menus (topmenus), toolbars, and other items on the window.

Defining Actions - the MENU statement

An action defined in the `.4gl` module, which identifies the program routine to be executed, can be associated with each action view shown on the form. You define the program logic to be executed for each action in the `.4gl` module.

- In this BDL program, the `MENU` statement supplies the list of actions and the statements to be executed for each action. The actions are specified with `ON ACTION` clauses:

```
ON ACTION query
  CALL query_cust()
```

- The `ON ACTION` clause defines the action name and the statements to be executed for the action. The presentation attributes - title, font, comment, etc. - for the graphical object that serves as the action view are defined in a separate action defaults file, or in the `ACTION DEFAULTS` section of the form file. This allows you to standardize the appearance of the views for common actions. Action Defaults are illustrated in Tutorial Chapter 5: Enhancing the Form on page 52.

  You can also use `ON ACTION` clauses with some other interactive BDL statements, such as `INPUT`, `INPUT ARRAY`, `DIALOG`, and `DISPLAY ARRAY`.
- When the `MENU` statement in your program is executed, the action views for the actions (**query**, in the example) that are listed in the interactive `MENU` statement are enabled. Only the action views for the actions in the specific `MENU` statement are enabled, so you must be sure to include a means of exiting the `MENU` statement. If there is no action view defined in your form specification file for a listed action, a simple push button action view is automatically displayed in the window. Control is turned over to the user, and the program waits until the user responds by selecting one of enabled action views or exiting the form. Once an action view is selected, the corresponding program routine (action) is executed.

See *Ring menus (MENU)* in the *Genero Business Development Language User Guide* for a complete discussion of the statement and all its options.

Displaying Messages and Errors

The `MESSAGE` and `ERROR` statements are used to display text containing a message to the user. The text is displayed in a specific area, depending on the front end configuration and window style. The `MESSAGE` text is displayed until it is replaced by another `MESSAGE` statement or field comment. You can specify any combination of variables and strings for the text. BDL generates the message to display by replacing any variables with their values and concatenating the strings:

```
MESSAGE "Customer " || l_custrec.store_num , || " retrieved."
```

The Localized Strings feature can be used to customize the messages for specific user communities. This is discussed in Tutorial Chapter 10: Localization on page 106.

Example: dispcust.4gl

This portion of the `dispcust.4gl` program connects to a database, opens a window and displays a form and a menu.

Program `dispcust.4gl`:

```
01 -- dispcust.4gl
02 SCHEMA custdemo
03
```

```
04 MAIN
05
06   CONNECT TO "custdemo"
07
08   CLOSE WINDOW SCREEN
09   OPEN WINDOW custwin WITH FORM "custform"
10   MESSAGE "Program retrieves customer 101"
11
12   MENU "Customer"
13     ON ACTION query
14       CALL query_cust()
15     ON ACTION exit
16       EXIT MENU
17   END MENU
18
19   CLOSE WINDOW custwin
20
21   DISCONNECT CURRENT
22
23 END MAIN
```

Note:

- Line `02` The `SCHEMA` statement is required since variables are defined as `LIKE` a database table in the function `query_cust`.
- Line `06`opens the connection to the `custdemo` database.
- Line `08` closes the default window named `SCREEN`, which is opened each time the runtime system starts a program containing interactive statements
- Line `09` uses the `WITH FORM` syntax to open a window having the identifier `custwin` containing the form identified as `custform`. The window name must be unique among all windows defined in the program. Its scope is the entire program. You can use the window's name to reference any open window in other modules with other statements. Although there can be multiple open windows, only one window may be current at a given time. By default, the window that opens will be a normal application window. The form identifier is the name of the compiled `.42f` file (`custform.42f`). The form identifier must be unique among form names in the program. Its scope of reference is the entire program.
- Line `10` displays a string as a `MESSAGE` to the user. The message will be displayed until it is replaced by a different string.
- Lines `12` through `17` contain the interactive `MENU` statement. By default, the menu options **query** and **exit** are displayed as buttons in the window, with `Customer` as the menu title. When the `MENU` statement is executed, the buttons are enabled, and control is turned over to the user. If the user selects the **query** button, the function `query_cust` will be executed. Following execution of the function, the action views (buttons in this case) are re-enabled and the program waits for the user to select an action again. If the user selects the **exit** button, the `MENU` statement is terminated, and the program continues with line 19.
- Line `19` The window `custwin` is closed which automatically closes the form, removing both objects from the application's memory.
- Line `21` The program disconnects from the database; as there are no more statements in `MAIN`, the program terminates.

The .4gl File - Retrieving and Displaying Data

The example demonstrates how to define a record so you can treat variables as a group. Static SQL instructions retrieve rows from the database which are displayed to the form using the `DISPLAY BY NAME` statement.

Defining a Record

In addition to defining individual variables, the `DEFINE` statement can define a record, a collection of variables each having its own data type and name. You put the variables in a record so you can treat them as a group. Then, you can access any member of a record by writing the name of the record, a dot (known as dot notation), and the name of the member.

```
DEFINE custrec RECORD
          store_num  LIKE customer.store_num
          store_name LIKE customer.store_name
       END RECORD
DISPLAY custrec.store_num
```

Your record can contain variables for the columns of a database table. At its simplest, you write `RECORD LIKE tablename.*` to define a record that includes members that match in data type all the columns in a database table. However, if your database schema changes often, it's best to list each member individually, so that a change in the structure of the database table won't break your code. Your record can also contain members that are not defined in terms of a database table.

Using SQL to Retrieve the Data

A subset of SQL, known as Static SQL, is provided as part of the BDL language and can be embedded in the program. At runtime, these SQL statements are automatically prepared and executed by the runtime System.

```
SELECT store_num, store_name INTO custrec.* FROM customer
```

Only a limited number of SQL instructions are supported this way. However, Dynamic SQL Management allows you to execute any kind of SQL statement.

Displaying a Record: DISPLAY BY NAME

A common technique is to use the names of database columns as the names of both the members of a program record and the fields in a form. Then, the `DISPLAY BY NAME` statement can be used to display the program variables. By default, a screen record consisting of the form fields associated with each database table column is automatically created. BDL will match the name to the name of the form field, ignoring any record name prefix:

```
DISPLAY BY NAME custrec.*
```

The program variables serve as the intermediary between the database and the form that is displayed to the user. Values from a row in the database table are retrieved into the program variables by an SQL `SELECT` statement, and are then displayed on the form. In Tutorial Chapter 6: Add, Update and Delete on page 62 you will see how the user can change the values in the form, resulting in changes to the program variables, which could then be used in SQL statements to modify the data in the database.

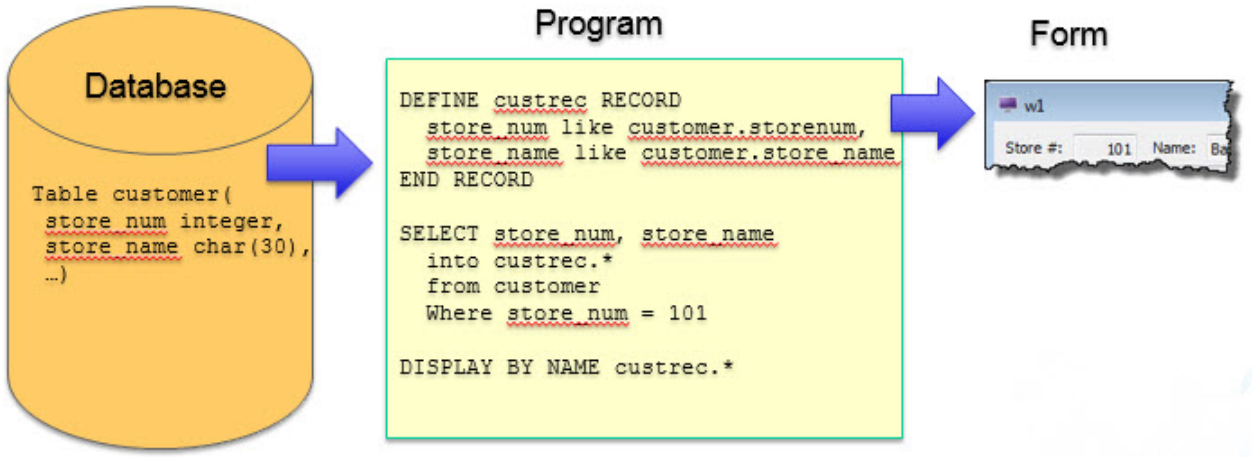

Figure 5: Flow between the database, program, and form

Example: dispcust.4gl (function query_cust)

This function retrieves a row from the `customer` table and displays it in a form.

Function `query_cust`:

```
01 FUNCTION query_cust()      -- displays one row
02   DEFINE l_custrec RECORD
03      store_num    LIKE customer.store_num,
04      store_name   LIKE customer.store_name,
05      addr         LIKE customer.addr,
06      addr2        LIKE customer.addr2,
07      city         LIKE customer.city,
08      state        LIKE customer.state,
09      zip-code      LIKE customer.zip-code,
10      contact_name LIKE customer.contact_name,
11      phone        LIKE customer.phone
12     END RECORD
13
14   SELECT store_num,
15          store_name,
16          addr,
17          addr2,
18          city,
19          state,
20          zip-code,
21          contact_name,
22          phone
23     INTO l_custrec.*
24     FROM customer
25     WHERE store_num = 101
26
27   DISPLAY BY NAME l_custrec.*
28   MESSAGE "Customer " || l_custrec.store_num ||
29           " displayed."
30 END FUNCTION
```

Note:

- Line `01` is the beginning of the function `query_cust`. No variables are passed to the function.

- Lines 02 thru 12 DEFINE a record l_custrec as LIKE columns in the customer database table, listing each variable separately.
- Line 14 thru 25 SELECT ... INTO can be used, since the statement will retrieve only one row from the database. The SELECT statement lists each column name to be retrieved, rather than using SELECT *. This allows for the possibility that additional columns might be added to a table at a future date. Since the SELECT list retrieves values for all the variables in the program record, in the order listed in the DEFINE statement, the shorthand INTO l_custrec.* can be used.
- Line 27 The names in the program record l_custrec match the names of screen fields on the form, so DISPLAY BY NAME can be used. l_custrec.* indicates that all of the members of the program record are to be displayed.
- Lines 28 and 29 A string for the MESSAGE statement is concatenated together using the double pipe (||) operator and displayed. The message consists of the string "Customer", the value of l_custrec.store_num, and the string "displayed".

There are no additional statements in the function, so the program returns to the MENU statement, awaiting the user's next action.

The Form Specification File

You can specify the layout of a form in a form specification file, which is compiled separately from your program. The form specification file defines the initial settings for the form, which can be changed programmatically at runtime.

Overview

Form specification files created in Genero Studio's Form Designer have a file extension of .4fd. Text-based form specification files have a file extension of .per. The structure of the form is independent of the use of the form. For example, one function can use a form to display a database row, another can let the user enter a new database row, and still another can let the user enter criteria for selecting database rows.

A Form can contain the following types of items:

- Container - groups other form items. Every form item must be in a container. GRID is the basic container, frequently used to display a single row of database data. TABLE containers can provide record-list presentation in columns and rows. Other containers, such as a FOLDER or GROUP, provide additional options for organizing the data that is displayed.
- FormField - defines an area where the user can view and edit data. The data is stored in variables defined in the .4gl source code file. The EDIT formfield provides a simple line-edit field. Other form items, such as a COMBOBOX or RADIOGROUP, provide a user-friendly interface to the data stored in the underlying formfield. The data type of a formfield can be defined by a database table column, or it can be FORMONLY - defined specifically in the form.
- Action view - allows the user to trigger actions specified in the .4gl file. An Action view can be a BUTTON, toolbar icon, or topmenu option, for example.
- Other - items that enhance the display or provide read-only information (an IMAGE or LABEL, for example).

Each form and form item has attributes that control its appearance and behavior. See the documentation for *Form specification files*, and *Form item attributes* in the *Genero Business Development Language User Guide* for additional information about form items.

Styles from a Presentation Styles file can be applied to the form and form items.

A basic form specification consists of the following sections:

The SCHEMA section (optional)

This specifies the database schema file to be used when the form is compiled. It is required if any form items are defined as data types based on a column of a database table.

```
SCHEMA custdemo
```

The ACTION DEFAULTS, TOPMENU, and TOOLBAR sections (optional)

These sections are provided to allow you to define the decoration for action views (action defaults), as well as to define topmenus and toolbars for the form. In this case, the definitions are specific to the form. If your definitions are in external XML files instead, they can be applied to any form.

This is discussed in chapter 5.

The LAYOUT section

This section defines the appearance of a form using a layout tree.

Of containers, which can hold other containers or can define a screen area. Some of the available containers are `GRID`, `VBOX`, `HBOX`, `GROUP`, `FOLDER`, and `PAGE`.

The simplest layout tree could have only a `GRID` container defining the dimensions and the position of the logical elements of a screen:

```
LAYOUT
  GRID
    grid-area
  END
END
```

The `END` keyword is mandatory to define the end of a container block.

The grid-area is delimited by curly braces. Within this area, you can specify the position of form items or interactive objects such as `BUTTON`, `COMBOBOX`, `CHECKBOX`, `RADIOGROUP`, `PROGRESSBAR`, and so on.

Simple form fields, delimited by square brackets (`[]`), are form items used to display data and take input. Generally, the number of characters in the space between the brackets defines the width of the region to be used by the item. For example, in the grid-area, the following field could be defined:

```
[f01           ]
```

This form field has an item tag of `f01`, which will be used to link the field to its definition in the `ATTRIBUTES` section of the form specification.

Interactive form items, such as `COMBOBOX`, `CHECKBOX`, and `RADIOGROUP`, can be used instead of simple form fields to represent the values in the underlying formfield. Special width calculations are done for some of these form items, such as `COMBOBOX`, `BUTTONEDIT`, and `DATEEDIT`. 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.

Text in the grid-area that is outside brackets is display-only text, as in the word `Company`:

```
Company [f01           ]
```

The TABLES section (optional)

If a database table or database view is referenced elsewhere in the form specification file, in the `ATTRIBUTES`

Section for example, the table or view must be listed in the TABLES section:

```
TABLES
  customer
END
```

A default screen record is automatically created for the form fields associated with each table listed in this section.

The ATTRIBUTES section

The ATTRIBUTES section defines properties of the items used in the form.

Form Fields

For form fields (items that can be used to display data or take input) the definition is:

```
<item-type> <item-tag> = <item-name>, <attribute-list> ;
```

- The *item-type* defines the kind of graphical object which must be used to display the form element.
- The *item-tag* identifies the form item in the display area.
- The *item-name* provides the name of the form item.
- The optional *attribute-list* defines the aspect and behavior of the form item.

Examples

```
EDIT f01 = customer.cust_num,REQUIRED;
COMBOBOX f03 = customer.state;
CHECKBOX f04 = formonly.propcheck;
```

The most commonly used item-type, EDIT, defines a simple line edit box for data input or display. This example uses an EDIT item-type for the form field f01. The COMBOBOX and CHECKBOX item types present the data contained in the form fields f03 and f04 in a user-friendly way.

The item-name must specify a database column as the name of the display field, or must be FORMONLY (fields defined as FORMONLY are discussed in chapter 11) Fields are associated with database columns only during the compilation of the form specification file, to identify the data type for the form field based on the database schema. After the form compiler identifies the data types, the association between fields and database columns is broken, and the item-name is associated with the screen record.

Form field and 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; you can insure that a value is entered in the field during the input of a new row (REQUIRED); columns in a table can be specified as sortable or non-sortable; numbers and dates can be formatted for display; data entry patterns can be defined and input data can be upshifted or downshifted.

A form field can be an EDIT, BUTTONEDIT, CHECKBOX, COMBOBOX, DATEEDIT, IMAGE, LABEL, PROGRESSBAR, RADIOGROUP, or TEXTEDIT.

Other form items

For form items that are not form fields (BUTTON, CANVAS, GROUP, static IMAGE, static LABEL, SCROLLGRID, and TABLE) the definition is:

```
<item-type> <item-tag>: <item-name> , <attribute-list> ;
```

Examples:

```
BUTTON btn1: print, TEXT = "Print Report";
```

```
LABEL lab1: label1, TEXT ="Customer";
```

The INSTRUCTIONS section (optional)

The `INSTRUCTIONS` section is used to define explicit screen records or screen arrays. This is discussed in Chapter 7

Example: Form Specification File custform.per

This form specification file is used with the `dispcust.4gl` program to display program variables to the user. This form uses a layout with a simple `GRID` to define the display area.

File `custform.per`:

```
01 SCHEMA custdemo
02
03 LAYOUT
04   GRID
05   {
06    Store #:[f01  ] Name:[f02                ]
07    Address:[f03                ]
08            [f04                ]
09       City:[f05             ]State:[f6]Zip:[f07   ]
10    Contact:[f08                          ]
11      Phone:[f09               ]
12
13   }
14   END   --grid
15 END  -- layout
16
17 TABLES
18     customer
19 END
20
21 ATTRIBUTES
22 EDIT f01 = customer.store_num, REQUIRED;
23 EDIT f02 = customer.store_name, COMMENT="Customer name";
24 EDIT f03 = customer.addr;
25 EDIT f04 = customer.addr2;
26 EDIT f05 = customer.city;
27 EDIT f6  = customer.state;
28 EDIT f07 = customer.zip-code;
29 EDIT f08 = customer.contact_name;
30 EDIT f09 = customer.phone;
31 END
```

Note:

- Line `01` lists the database schema file from which the form field data types will be obtained.
- Lines `03` through `15` delimit the `LAYOUT` section of the form.
- Lines `04` thru `14` delimit the `GRID` area, indicating what will be displayed to the user between the curly brackets on lines `05` and `13`.
- Line `17` The `TABLES` statement is required since the field descriptions reference the columns of the database table customer.
- Within the grid area, the form fields have item tags linking them to descriptions in the `ATTRIBUTES` section, in lines `20` thru `28`. As an example, `f01` is the display area for a program variable having the same data type definition as the `store_num` column in the `customer` table of the `custdemo` database.

- Line 22 All of the item-tags in the form layout section are listed in the `ATTRIBUTES` section. For example, the item-tag `f01` is listed as having an item-type of `EDIT`. This field will be used for display only in this program, but the same form will be used for input in a later program. An additional attribute, `REQUIRED`, indicates that when this form is used for input, an entry in the field `f01` must be made. This prevents the user from trying to add a row with a NULL `store_num` to the `customer` table, which would result in an error message from the database.
- Line 23 The second field is defined with the attribute `COMMENT`, which specifies text to be displayed when this field gets the focus, or as a tooltip when the mouse goes over the field.

Compiling the Program and Form

When this form is compiled (translated) using the **Compile** menu option in the Project view or the `fglform` tool, an XML file is generated that has a file extension of `.42f`. The runtime system uses this file along with your programs to define the Abstract User Interface.

To compile the form with `fglform`:

```
fglform custform.per
```

Compile the single module program:

```
fglcomp dispcust.4gl
```

Execute the program:

```
fglrun dispcust.42m
```

Tutorial Chapter 4: Query by Example

The program in this chapter allows the user to search a database by entering criteria in a form. The search criteria is used to build an SQL `SELECT` statement to retrieve the desired database rows. A cursor is defined in the program, to allow the user to scroll back and forth between the rows of the result set. Testing the success of the SQL statements and handling errors is illustrated.

Implementing Query-by-Example

This program implements query-by-example, using the `CONSTRUCT` statement to allow the user to enter search criteria in a form. The criteria is used to build an SQL `SELECT` statement which will retrieve rows from the `customer` database table.

A `SCROLL CURSOR` is defined in the program, to allow the user to scroll back and forth between the rows of the result set. The `SQLCA.SQLCODE` is used to test the success of the SQL statements. Handling errors, and allowing the user to cancel the query, is illustrated.

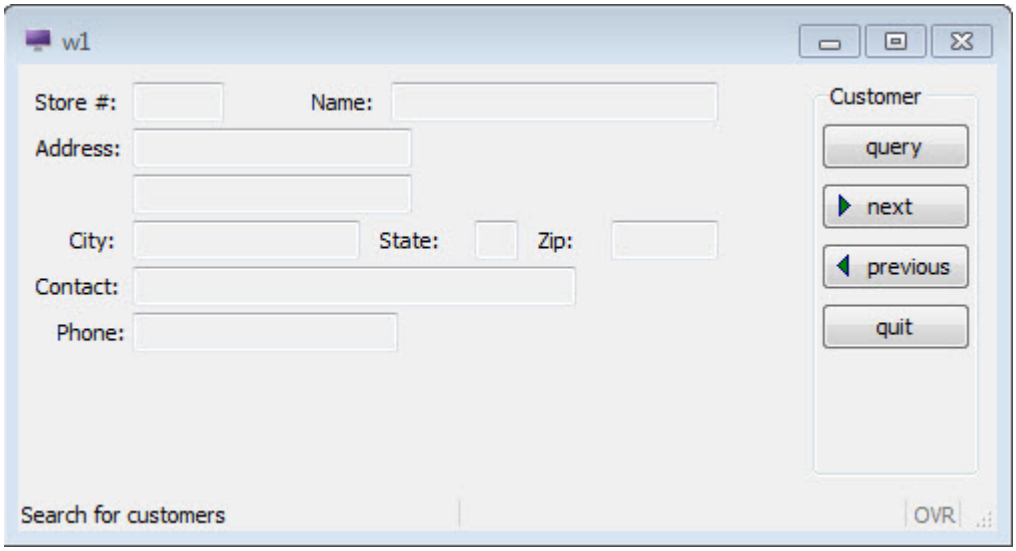

Figure 6: Display of the custform form used for query-by-example in Chapter 4.

Steps for implementing Query-by-Example

This topic describes the steps involved to implement query-by-example using the `CONSTRUCT` statement.

1. Define fields linked to database columns in a form specification file.
2. Define a `STRING` variable in your program to hold the query criteria.
3. Open a window and display the form.
4. Activate the form with the interactive dialog statement `CONSTRUCT`, for entry of the query criteria. Control is turned over to the user to enter his criteria.
5. The user enters his criteria in the fields specified in the `CONSTRUCT` statement.

The CONSTRUCT statement accepts logical operators in any of the fields to indicate ranges, comparisons, sets, and partial matches. Using the form in this program, for example, the user can enter a specific value, such as "IL" in the state field, to retrieve all the rows from the customer table where the state column = IL. Or he can enter relational tests, such as "> 103", in the Store # field, to retrieve only those rows where the `store_num` column is greater than 103.

6. After entering his criteria, the user selects **OK**, to instruct your program to continue with the query, or **Cancel** to terminate the dialog.

   In this program, the action views for accept (**OK**) and cancel are displayed as buttons on the screen.

7. If the user accepts the dialog, the CONSTRUCT statement creates a Boolean expression by generating a logical expression for each field with a value and then applying unions (and relations) to the field statements.

   This expression is stored in the character string that you specified in the CONSTRUCT statement.

8. You can then use the Boolean expression to create a STRING variable containing a complete SELECT statement.

   You must supply the WHERE keyword to convert the Boolean expression into a WHERE clause. Make sure that you supply the spaces required to separate the constructed Boolean expression from the other parts of the SELECT statement.

9. Execute the statement to retrieve the row(s) from the database table, after preparing it or declaring a cursor for SELECT statements that might retrieve more than one row.

Using CONSTRUCT and STRING variables

The CONSTRUCT statement temporarily binds the specified form fields to database columns. It allows you to identify database columns for which the user can enter search criteria.

A basic CONSTRUCT statement has the following format:

```
CONSTRUCT <variable-name> ON <column-list> FROM <field-list
```

Each field and CONSTRUCT corresponding column must be the same or compatible data types. You can use the BY NAME clause when the fields on the screen form have the same names as the corresponding columns in the ON clause. The user can query only the screen fields implied in the BY NAME clause.

```
CONSTRUCT BY NAME <variable-name> ON <column-list>
```

The runtime system converts the entered criteria into a Boolean SQL condition that can appear in the WHERE clause of a SELECT statement. The variable to hold the query condition can be defined as a STRING data type. Strings are a variable length, dynamically allocated character string data type, without a size limitation. The STRING variable can be concatenated, using the double pipe operator (||), with the text required to form a complete SQL SELECT statement. The LET statement can be used to assign a value to the variable. For example:

```
DEFINE where_clause, sqltext STRING
CONSTRUCT BY NAME where_clause ON customer.*
LET sql_text = "SELECT COUNT(*) FROM customer WHERE " || where_clause
```

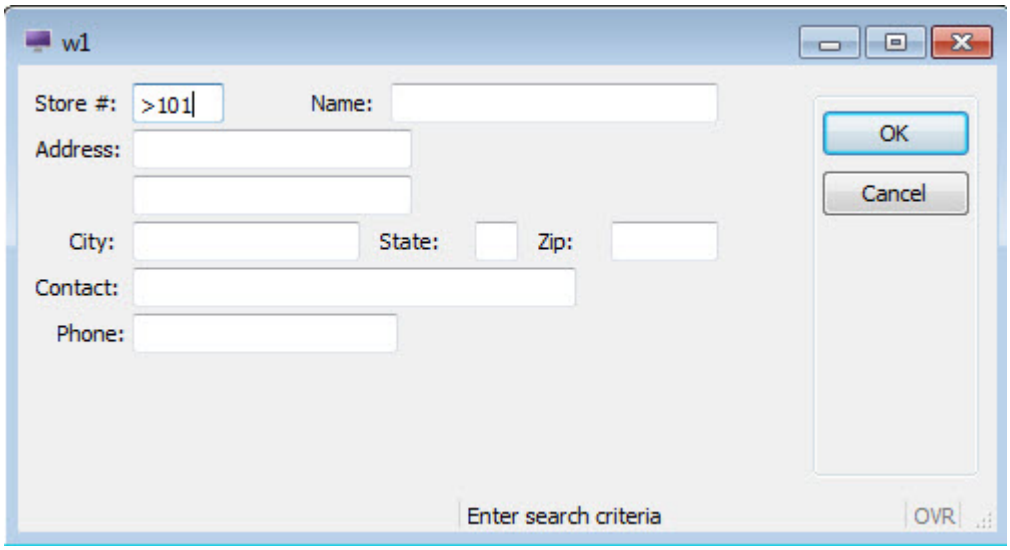

Figure 7: Display of user select criteria on Windows™ Platform

In this example the user has entered the criteria "> 101" in the `store_num` field. The `where_clause` value would be generated as

```
"store_num > 101"
```

And the complete sql text would be

```
"SELECT COUNT(*) FROM customer WHERE store_num > 101"
```

Preparing the SQL Statement

The `STRING` created in the query-by-example is not valid for execution. The `PREPARE` instruction sends the text of the string to the database server for parsing, validation, and to generate the execution plan.

The scope of a prepared SQL statement is the module in which it is declared.

```
PREPARE cust_cnt_stmt FROM sql_text
```

A prepared SQL statement can be executed with the `EXECUTE` instruction.

```
EXECUTE cust_cnt_stmt INTO cust_cnt
```

Since the SQL statement will only return one row (containing the count) the `INTO` syntax of the `EXECUTE` instruction can be used to store the count in the local variable `cust_cnt`. (The function `cust_select` illustrates the use of database cursors with SQL `SELECT` statements.)

When a prepared statement is no longer needed, the `FREE` instruction will release the resources associated with the statement.

```
FREE cust_cnt_stmt
```

Allowing the User to Cancel the Query Operation

The query-by-example application demonstrates methods used to allow users to cancel an interactive dialog statement using predefined actions and conditional logic.

You can handle user Cancel actions or interrupts gracefully during interactive dialog instructions such as `CONSTRUCT`, using the built-in global integer variable `INT_FLAG` and the `DEFER INTERRUPT` statement. Use conditional logic statements to test for user cancel or interrupt actions and specify statement blocks to execute conditionally based on the results.

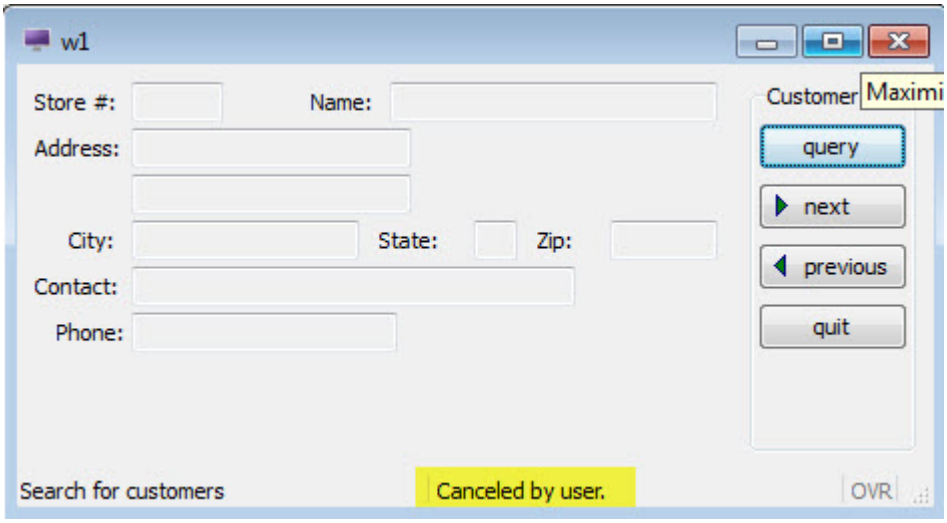

Figure 8: User Cancels query and exits back to the main menu

Predefined Actions (accept/cancel)

The language predefines some actions and associated names for common operations, such as `accept` or `cancel`, used during interactive dialogs such as `CONSTRUCT`.

You do not have to define predefined actions in the interactive instruction block, the runtime system interprets predefined actions. For example, when the `accept` action is caught, the dialog is validated.

You can define action views (such as buttons, toolbar icons, menu items) in your form using these predefined names; the corresponding action will automatically be attached to the view. If you do not define any action views for the actions, default buttons for these actions will be displayed on the form as appropriate when interactive dialog statements are executed.

When the `CONSTRUCT` statement executes, buttons representing accept and cancel actions (**OK**/**Cancel**) will be displayed by default, allowing the user to validate or cancel the interactive dialog statement.

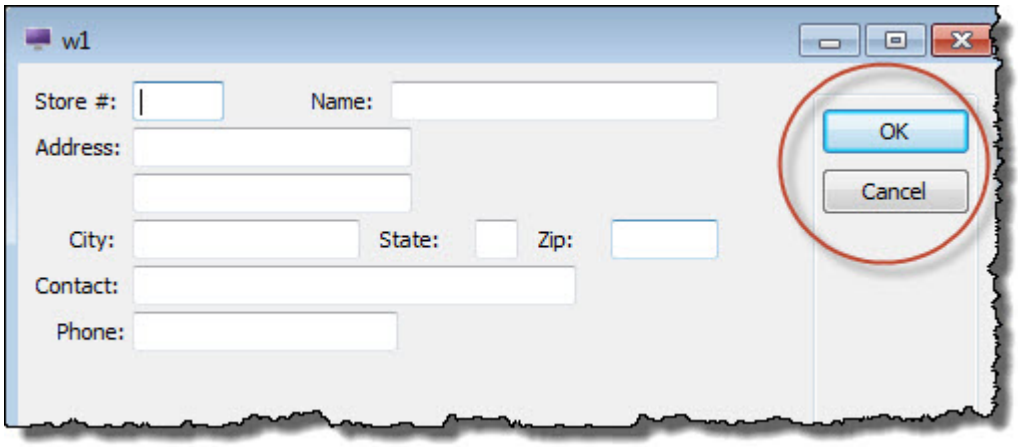

Figure 9: Default buttons for predefined actions accept and cancel.

DEFER INTERRUPT and the INT_FLAG

If the user selects **Cancel** during the `CONSTRUCT`, the built-in global integer variable `INT_FLAG` is automatically set to `TRUE`.

Once `INT_FLAG` is set to `TRUE`, your program must reset it to `FALSE` to detect a new cancellation. You typically set `INT_FLAG` to `FALSE` before you start a dialog instruction, and you test it just after (or in the `AFTER CONSTRUCT` / `AFTER INPUT` block) to detect if the dialog was canceled:

```
LET INT_FLAG = FALSE
CONSTRUCT BY NAME where_part
  ...
END CONSTRUCT
IF INT_FLAG = TRUE THEN
  ...
END IF
```

The statement `DEFER INTERRUPT` in your `MAIN` program block will prevent your program from terminating abruptly if a `SIGINT` signal is received. When using a GUI interface, the user can generate an interrupt signal if you have an action view named 'interrupt' (the predefined interrupt action). If an interrupt event is received, `TRUE` is assigned to `INT_FLAG`.

It is up to the programmer to manage the interruption event (stop or continue with the program), by testing the value of `INT_FLAG` variable.

Interruption handling is discussed in the report example, in Tutorial Chapter 9: Reports on page 96.

Conditional Logic

Once the `CONSTRUCT` statement is completed, you must test whether the `INT_FLAG` was set to `TRUE` (whether the user canceled the dialog). Genero BDL provides the conditional logic statements `IF` or `CASE` to test a set of conditions.

The IF statement

The `IF` instruction executes a group of statements conditionally.

```
IF <condition> THEN
  ...
ELSE
  ...
END IF
```

`IF` statements can be nested. The `ELSE` clause may be omitted.

If condition is TRUE, the runtime system executes the block of statements following THEN, until it reaches either the ELSE keyword or the END IF keywords. Your program resumes execution after END IF. If condition is FALSE, the runtime system executes the block of statements between ELSE and END IF.

```
IF (INT_FLAG = TRUE) THEN
  LET INT_FLAG = FALSE
  LET cont_ok = FALSE
ELSE
  LET cont_ok = TRUE
END IF
```

The CASE statement

The CASE statement specifies statement blocks to be executed conditionally, depending on the value of an expression.

Unlike IF statements, CASE does not restrict the logical flow of control to only two branches. Particularly if you have a series of nested IF statements, the CASE statement may be more readable. In the previous example, the CASE statement could have been substituted for the IF statement:

```
CASE
WHEN (INT_FLAG = TRUE)
  LET INT_FLAG = FALSE
  LET cont_ok = FALSE
OTHERWISE
  LET cont_ok = TRUE
END CASE
```

Usually, there would be several conditions to check. The following statement uses an alternative syntax, since all the conditions check the value of var1:

```
CASE var1
WHEN 100
  CALL routine_100()
WHEN 200
  CALL routine_200()
OTHERWISE
  CALL error_routine()
END CASE
```

The first WHEN condition in the CASE statement will be evaluated. If the condition is true (var1=100), the statement block is executed and the CASE statement is exited. If the condition is not true, the next WHEN condition will be evaluated, and so on through subsequent WHEN statements until a condition is found to be true, or OTHERWISE or END CASE is encountered. The OTHERWISE clause of the CASE statement can be used as a catchall for unanticipated cases.

See Flow Control for other examples of IF and CASE syntax and the additional conditional statement WHILE.

The Query program

The Query program consists of two modules. The custmain.4gl module must be linked with the custquery.4gl module in order for the program to be run.

The line numbers shown in the code are for reference only, and are not a part of the code.

Example: Module custmain.4gl
This module contains the `MAIN` program block for the query program, and the `MENU` that drives the query actions.

Module `custmain.4gl`:

```
01 MAIN
02
03   DEFER INTERRUPT
04
05   CONNECT TO "custdemo"
06   CLOSE WINDOW SCREEN
07   OPEN WINDOW w1 WITH FORM "custform"
08
09   MENU "Customer"
10     ON ACTION query
11       CALL query_cust()
12     ON ACTION next
13       CALL fetch_rel_cust(1)
14     ON ACTION previous
15       CALL fetch_rel_cust(-1)
16     ON ACTION exit
17       EXIT MENU
18   END MENU
19
20   CLOSE WINDOW w1
21
22   DISCONNECT CURRENT
23
24 END MAIN
```

Note:

- Line `01` Beginning of the `MAIN` block. The `SCHEMA` statement is not needed since this module does not define any program variables in terms of a database table.
- Line `03` uses the `DEFER INTERRUPT` statement to prevent the user from terminating the program prematurely by pressing the **INTERRUPT** key.
- Line `07` opens a window with the same form that was used in the Chapter 3 example.
- Lines `09` thru `18` contains the `MENU` for the query program. Four actions - `query`, `next`, `previous`, and `quit` - will be displayed as buttons on the form. The predefined actions `accept` (OK button) and `cancel` will automatically be displayed as buttons when the `CONSTRUCT` statement is executed.
- Line `11` calls the function `query_cust` in the `cust_query.4gl` module.
- Line `13` calls the function `fetch_rel_cust` in the `cust.query.4gl` module. The literal value `1` is passed to the function, indicating that the cursor should move forward to the next row.
- Line `15` calls the function `fetch_rel_cust` also, but passes the literal value -1, indicating that the cursor should move backwards to retrieve the previous row in the results set.
- Line `17` exits the `MENU`statement.
- Line `20` closes the window that was opened.
- Line `22` disconnects from the database.

There are no further statements so the Query program terminates.

Example: Module custquery.4gl
This module of the Query program contains the logic for querying the database and displaying the data retrieved.

The function `query_cust` is called by the "query" option of the `MENU` in `custmain.4gl`.

Module `custquery.4gl` (and function `query_cust`):

```
01 -- custquery.4gl
02
03 SCHEMA custdemo
04
05 DEFINE  mr_custrec RECORD
06  store_num    LIKE customer.store_num,
07  store_name   LIKE customer.store_name,
08  addr         LIKE customer.addr,
09  addr2        LIKE customer.addr2,
10  city         LIKE customer.city,
11  state        LIKE customer.state,
12  zip-code     LIKE customer.zip-code,
13  contact_name LIKE customer.contact_name,
14  phone        LIKE customer.phone
15 END RECORD
16
17 FUNCTION query_cust()
18   DEFINE cont_ok     SMALLINT,
19          cust_cnt    SMALLINT,
20          where_clause STRING
21   MESSAGE "Enter search criteria"
22   LET cont_ok = FALSE
23
24   LET INT_FLAG = FALSE
25   CONSTRUCT BY NAME where_clause
26      ON customer.store_num,
27         customer.store_name,
28         customer.city,
29         customer.state,
30         customer.zip-code,
31         customer.contact_name,
32         customer.phone
33
34   IF (INT_FLAG = TRUE) THEN
35     LET INT_FLAG = FALSE
36     CLEAR FORM
37     LET cont_ok = FALSE
38     MESSAGE "Canceled by user."
39   ELSE
40     CALL get_cust_cnt(where_clause)
41       RETURNING cust_cnt
42     IF (cust_cnt > 0) THEN
43       MESSAGE cust_cnt USING "<<<<",
44                  " rows found."
45     CALL cust_select(where_clause)
46       RETURNING cont_ok
47     ELSE
48       MESSAGE "No rows found."
49       LET cont_ok = FALSE
50     END IF
51   END IF
52
53   IF (cont_ok = TRUE) THEN
54     CALL display_cust()
55   END IF
56
57 END FUNCTION
```

Note:

- Line `03` is required to identify the database schema file to be used when compiling the module.

- Lines 05 thru 15 define a RECORD, mr_custrec, that is modular in scope, since it is at the top of the module and outside any function. The values of this record will be available to, and can be set by, any function in this module.
- Line 17: Function query_cust. This is the beginning of the function query_cust.
- Line 18 defines cont_ok, a local variable of data type SMALLINT, to be used as a flag to indicate whether the query should be continued. The keywords TRUE and FALSE are used to set the value of the variable (0=FALSE, <> 0=TRUE).
- Line 19 defines another local SMALLINT variable, cust_cnt, to hold the number of rows returned by the SELECT statement.
- Line 20 defines where_clause as a local STRING variable to hold the boolean condition resulting from the CONSTRUCT statement.
- Line 21 displays a message to the user that will remain until it is replaced by another MESSAGE statement.
- Line 22 sets cont_ok to FALSE, prior to executing the statements of the function.
- Line 24 sets INT_FLAG to FALSE. It is common to set this global flag to FALSE immediately prior to the execution of an interactive dialog, so your program can test whether the user attempted to cancel the dialog.
- Lines 25 thru 32: The CONSTRUCT statement lists the database columns for which the user may enter search criteria. The program does not permit the user to enter search criteria for the address columns. The BY NAME syntax matches the database columns to form fields having the same name.
- Line 34 is the beginning of an IF statement testing the value of INT_FLAG. This test appears immediately after the CONSTRUCT statement, to test whether the user terminated the CONSTRUCT statement (INT_FLAG would be set by the runtime system to TRUE).
- Lines 35 thru 38 are executed only if the value of INT_FLAG is TRUE. The INT_FLAG is immediately reset to FALSE, since it is a global variable which other parts of your program will test. The form is cleared of any criteria that the user has entered, the cont_ok flag is set to FALSE, and a message is displayed to the user. The program will continue with the statements after the END IF on line 49.
- Lines 40 thru 50: contain the logic to be executed if INT_FLAG was not set to TRUE (the user did not cancel the query).
    - In lines 40 and 41, the get_cust_cnt function is called, to retrieve the number of rows that would be returned by the query criteria. The where_clause variable is passed to the function, and the value returned will be stored in the cust_cnt variable.
    - Lines 42 is the beginning of a nested IF statement, testing the value of cust_cnt.
    - Lines 43 thru 46 are executed if the value of cust_cnt is greater than zero; a message with the number of rows returned is displayed to the user, and the function cust_select is called. The where_clause is passed to this function, and the returned value is stored in cont_ok. Execution continues with the statement after the END IF on line 51.
    - Lines 48 and 49 are executed if the value is zero (no rows found); a message is displayed to the user, and cont_ok is set to FALSE. Execution continues after the END IF on line 51.
- Line 49 is the end of the IF statement beginning on line 33.
- Lines 53 thru 55 test the value of cont_ok, which will have been set during the preceding IF statements and in the function cust_select. If cont_ok is TRUE, the function display_cust is called.
- Line 57 is the end of the query_cust function.

Example: custquery.4gl (Function get_cust_cnt)
This function is called by the function `query_cust` to return the count of rows that would be retrieved by the `SELECT` statement. The criteria previously entered by the user and stored in the variable `where_clause` is used.

Function `get_cust_cnt`:

```
01 FUNCTION get_cust_cnt(p_where_clause)
02   DEFINE p_where_clause STRING,
03          sql_text STRING,
04          cust_cnt SMALLINT
05
06   LET sql_text =
07    "SELECT COUNT(*) FROM customer" ||
08    " WHERE " || p_where_clause
09
10   PREPARE cust_cnt_stmt FROM sql_text
11   EXECUTE cust_cnt_stmt INTO cust_cnt
12   FREE cust_cnt_stmt
13
14   RETURN cust_cnt
15
16 END FUNCTION
```

Note:

- Line `01` The function accepts as a parameter the value of `where_clause`, stored in the local variable `p_where_clause` defined on Line `60`.
- Line `02` defines a local string variable, `sql_txt`, to hold the complete text of the SQL `SELECT` statement.
- Line `04` defines a local variable `cust_cnt` to hold the count returned by the `SELECT` statement.
- Lines `06` thru `08` create the string containing the complete SQL `SELECT` statement, concatenating `p_where_clause` at the end using the || operator. Notice that the word `WHERE` must be provided in the string.
- Line `10` uses the `PREPARE` statement to convert the string into an executable SQL statement, parsing the statement and storing it in memory. The prepared statement is modular in scope. The prepared statement has the identifier `cust_cnt_stmt`, which does not have to be defined.
- Line `11` executes the SQL `SELECT` statement contained in `cust_cnt_stmt`, using the `EXECUTE ... INTO` syntax to store the value returned by the statement in the variable `cust_cnt`. This syntax can be used if the SQL statement returns a single row of values.
- Line `12` The `FREE` statement releases the memory associated with the `PREPARE`d statement, since this statement is no longer needed.
- Line `14` returns the value of `cust_cnt` to the calling function, `query_cust`.
- Line `16` is the end of the `get_cust_cnt` function.

Retrieving data from the Database

When an SQL `SELECT` statement in your application will retrieve more than one row, a cursor must be used to pass the selected data to the program one row at a time.

Using Cursors

The cursor is a data structure that represents a specific location within the active set of rows that the `SELECT` statement retrieved.

- Sequential cursor - reads through the active set only once each time it is opened, by moving the cursor forward one row each time a row is requested.

- Scroll cursor - fetches the rows of the active set in any sequence. To implement a scroll cursor, the database server creates a temporary table to hold the active set.

The scope of a cursor is the module in which it is declared. Cursor names must be unique within a module.

The general sequence of program statements when using a SELECT cursor for Query-by-Example is:

- DECLARE - the program declares a cursor for the STRING that contains the SQL SELECT statement. This allocates storage to hold the cursor. The string does not have to be prepared using the PREPARE statement.
- OPEN - the program opens the cursor. The active set associated with the cursor is identified, and the cursor is positioned before the first row of the set.
- FETCH - the program fetches a row of data into host variables and processes it. The syntax FETCH NEXT <*cursor-identifier*> INTO <*variable-names*> can be used with a SCROLL CURSOR to fetch the next row relative to the current position of the cursor in the SQL result set. Using FETCH PREVIOUS ... moves the cursor back one row in the SQL result set.
- CLOSE - the program closes the cursor after the last row desired is fetched. This releases the active result set associated with the cursor. The cursor can be reopened.
- FREE - when the cursor is no longer needed, the program frees the cursor to release the storage area holding the cursor. Once a cursor has been freed, it must be declared again before it can be reopened.

The cursor program statements must appear physically within the module in the order listed.

The SQLCA.SQLCODE

The SQLCA name stands for "SQL Communication Area". The SQLCA variable is a predefined record containing information on the execution of an SQL statement.

The SQLCA record is filled after any SQL statement execution. The SQLCODE member of this record contains the SQL execution code:

Table 2: SQL execution codes

Execution Code	Description
0	SQL statement executed successfully.
100	No rows were found.
<0	An SQL error occurred.

The NOTFOUND constant is a predefined integer value that evaluates to "100". This constant is typically used to test the execution status of an SQL statement returning a result set, to check if rows have been found.

Example custquery.4gl (function cust_select)

This function is called by the function query_cust, if the row count returned by the function get_cust_cnt indicates that the criteria previously entered by the user and stored in the variable where_clause would produce an SQL SELECT result set.

Function cust_select:

```
01 FUNCTION cust_select(p_where_clause)
02   DEFINE p_where_clause STRING,
03          sql_text STRING,
04          fetch_ok SMALLINT
05
06   LET sql_text = "SELECT store_num, " ||
07      " store_name, addr, addr2, city, " ||
08      " state, zip-code, contact_name, phone " ||
09      " FROM customer WHERE " || p_where_clause ||
```

```
10      " ORDER BY store_num"
11
12   DECLARE cust_curs SCROLL CURSOR FROM sql_text
13   OPEN cust_curs
14   CALL fetch_cust(1)   -- fetch the first row
15     RETURNING fetch_ok
16   IF NOT (fetch_ok) THEN
17     MESSAGE "no rows in table."
18   END IF
19
20   RETURN fetch_ok
21
22 END FUNCTION
```

Note:

- Line 01 The function cust_select accepts as a parameter the where_clause, storing it in the local variable p_where_clause.
- Lines 06 thru 10 concatenate the entire text of the SQL statement into the local STRING variable sql_txt.
- Line 12 declares a SCROLL CURSOR with the identifier cust_curs, for the STRING variable sql_text.
- Line 13 opens the cursor, positioning before the first row of the result set. These statements are physically in the correct order within the module.
- Lines 14 and 15 call the function fetch_cust, passing as a parameter the literal value 1, and returning a value stored in the local variable fetch_ok. Passing the value 1 to fetch_cust will result in the NEXT row of the result set being fetched (see the logic in the function fetch_cust), which is this case would be the first row.
- Line 16 Since fetch_ok is defined as a SMALLINT, it can be used as a flag containing the values TRUE or FALSE. The value returned from the function fetch_cust indicates whether the fetch was successful.
- Line 17 displays a message to the user if the FETCH was not successful. Since this is the fetch of the first row in the result set, another user must have deleted the rows after the program selected the count.
- Line 20 returns the value of fetch_ok to the calling function. This determines whether the function display_cust is called.
- Line 22 is the end of the function cust_select.

Example: custquery.4gl (function fetch_cust)

This function is designed so that it can be reused each time a row is to be fetched from the customer database table; a variable is passed to indicate whether the cursor should move forward one row or backward one row.

Function fetch_cust:

```
01 FUNCTION fetch_cust(p_fetch_flag)
02   DEFINE p_fetch_flag SMALLINT,
03             fetch_ok SMALLINT
04
05   LET fetch_ok = FALSE
06   IF (p_fetch_flag = 1) THEN
07     FETCH NEXT cust_curs
08       INTO mr_custrec.*
09   ELSE
10     FETCH PREVIOUS cust_curs
11       INTO mr_custrec.*
12   END IF
13
14   IF (SQLCA.SQLCODE = NOTFOUND) THEN
```

```
15     LET fetch_ok = FALSE
16   ELSE
17     LET fetch_ok = TRUE
18   END IF
19
20   RETURN fetch_ok
21
22 END FUNCTION
```

Note:

- Line 01 The function fetch_cust accepts a parameter and stores it in the local variable p_fetch_flag.
- Line 03 defines a variable, fetch_ok, to serve as an indicator whether the FETCH was successful.
- Lines 06 thru 12 tests the value of p_fetch_flag, moving the cursor forward with FETCH NEXT if the value is 1, and backward with FETCH PREVIOUS if the value is -1. The values of the row in the customer database table are fetched into the program variables of the mr_custrec record. The INTO mr_custrec.* syntax requires that the program variables in the record mr_custrec are in the same order as the columns are listed in the SELECT statement.
- Lines 14 thru 15 tests SQLCA.SQLCODE and sets the value of fetch_ok to FALSE if the fetch did not return a row. If the FETCH was successful, fetch_ok is set to TRUE.
- Line 20 returns the value of fetch_ok to the calling function.
- Line 22 is the end of the function fetch_cust.

Example: custquery.4gl (function fetch_rel_cust)

This function is called by the MENU options **next** and **previous** in the custmain.4gl module.

Function fetch_rel_cust:

```
01 FUNCTION fetch_rel_cust(p_fetch_flag)
02   DEFINE p_fetch_flag SMALLINT,
03          fetch_ok SMALLINT
04
05   MESSAGE " "
06   CALL fetch_cust(p_fetch_flag)
07     RETURNING fetch_ok
08
09   IF (fetch_ok) THEN
10     CALL display_cust()
11   ELSE
12     IF (p_fetch_flag = 1) THEN
13       MESSAGE "End of list"
14     ELSE
15       MESSAGE "Beginning of list"
16     END IF
17   END IF
18
19 END FUNCTION
```

Note:

- Line 01 The parameter passed to p_fetch_flag will be 1 or -1, depending on the direction in which the SCROLL CURSOR is to move.
- Line 05 resets the MESSAGE display to blanks.
- Line 06 calls the function fetch_cust, passing it the value of p_fetch_flag. The function fetch_cust uses the SCROLL CURSOR to retrieve the next row in the direction indicated, returning FALSE if there was no row found.

- Lines09 and 10 If a row was found (the `fetch_cust` function returned `TRUE`) the `display_cust` function is called to display the row in the form.
- Line 13 If no rows were found and the direction is forward, indicated by `p_fetch_flag` of `1`, the cursor is past the end of the result set.
- Line 15 If no rows were found and the direction is backward, indicated by `p_fetch_flag` of `-1`, the cursor is prior to the beginning of the result set.
- Line 19 is the end of the function `fetch_rel_cust`.

Example: custquery.4gl (function display_cust)

This function displays the contents of the `mr_custrec` record in the form. It is called by the functions `query_cust` and `fetch_rel_cust`.

Function `display_cust`:

```
01 FUNCTION display_cust()
02   DISPLAY BY NAME mr_custrec.*
03 END FUNCTION
```

Note:

- Line `02` uses the `DISPLAY BY NAME` syntax to display the contents of the program record `mr_custrec` to the form fields having the same name.

Compiling and Linking the Program

The two example modules must be compiled and then linked into a single program. You can select the **Build** option in the Genero Studio Project view to perform these tasks or use command line tools.

From the command line:

```
fglcomp custmain.4gl
fglcomp custquery.4gl
```

This produces the object modules `custmain.42m` and `custquery.42m`, which must be linked to produce the program `cust.42r`:

```
fgllink -o cust.42r custmain.42m custquery.42m
```

Or, compile both modules and link at the same time:

```
fgl2p -o cust.42r custmain.4gl custquery.4gl
```

Modifying the Program to Handle Errors

Topics in this section describe methods to detect and handle errors encountered during program execution.

The WHENEVER ERROR statement

Since program statements that access the database may be expected to fail occasionally (the row is locked, etc.) the `WHENEVER ERROR` statement can be used to handle this type of error.

By default, when a runtime error occurs the program will stop. To prevent this happening when SQL statements that access the database fail, surround the SQL statement with `WHENEVER ERROR` statements, as in this example based on the `fetch_cust` function in the `custquery.4gl` program module:

```
01 IF (p_fetch_flag = 1) THEN
```

```
02   WHENEVER ERROR CONTINUE
03   FETCH NEXT cust_curs
04     INTO mr_custrec.*
05   WHENEVER ERROR STOP
06   ...
```

`WHENEVER ERROR` statements are modular in scope, and generate additional code for exception handling when the module is compiled. This exception handling is valid until the end of the module or until a new `WHENEVER ERROR` instruction is encountered by the compiler.

When the example code is compiled, `WHENEVER ERROR CONTINUE` will generate code to prevent the program from stopping if the `FETCH` statement fails. Immediately after the `FETCH` statement, the `WHENEVER ERROR STOP` instruction will generate the code to reset the default behavior for the rest of the module.

You can write your own error function to handle SQL errors, and use the `WHENEVER ERROR CALL` <*function-name*> syntax to activate it. Runtime errors may be logged to an error log.

Negative SQLCA.SQLCODE

The database server returns an execution code whenever an SQL statement is executed, available in `SQLCA.SQLCODE`. If the code is a negative number, an SQL error has occurred.

Just as we checked the `SQLCA.SQLCODE` for the `NOTFOUND` condition, we can also check the code for database errors (negative `SQLCODE`). The `SQLCA.SQLCODE` should be checked immediately after each SQL statement that may fail, including `DECLARE`, `OPEN`, `FETCH`, etc. For simplicity of the examples, the error handling in these programs is minimal.

SQLERRMESSAGE

If an SQL error occurs, the `SQLERRMESSAGE` operator returns the error message associated with the error code. This is a character string that can be displayed to the user with the `ERROR` instruction.

```
ERROR SQLERRMESSAGE
```

Changes to function `fetch_cust` (`custquery.4gl`)

```
01 FUNCTION fetch_cust (p_fetch_flag)
02    DEFINE p_fetch_flag SMALLINT,
03               fetch_ok SMALLINT
04
05   LET fetch_ok = FALSE
06   IF (p_fetch_flag = 1) THEN
07      WHENEVER ERROR CONTINUE
08      FETCH NEXT cust_curs
09         INTO mr_custrec.*
10      WHENEVER ERROR STOP
11   ELSE
12       WHENEVER ERROR CONTINUE
13       FETCH PREVIOUS cust_curs
14         INTO mr_custrec.*
15       WHENEVER ERROR STOP
16   END IF
17
18   CASE
19   WHEN (SQLCA.SQLCODE = 0)
20     LET fetch_ok = TRUE
21   WHEN (SQLCA.SQLCODE = NOTFOUND)
22     LET fetch_ok = FALSE
23   WHEN (SQLCA.SQLCODE < 0)
24     LET fetch_ok = FALSE
25     ERROR SQLERRMESSAGE
26   END CASE
```

```
27
28  RETURN fetch_ok
29
30 END FUNCTION
```

Note:

- Lines 08, 09, 13,14 The SQL statements are surrounded by WHENEVER ERROR statements. If an error occurs during the SQL statements, the program will continue. The error handling is reset to the default (STOP) immediately after each SQL statement so that failures of other program statements will not be ignored.
- Lines 18 to 26 Immediately after the WHENEVER ERROR STOP statement, the SQLCA.SQLCODE is checked, to see whether the SQL statement succeeded. A CASE statement is used, since there are more than two conditions to be checked.

Close and Free the Cursor

Closing and freeing the cursor when you no longer need it is good practice, especially if the modules are part of a larger program.

This function must be placed in the same module as the DECLARE/OPEN/FETCH statements and in sequence, so this is the last function in the query_cust module. However, the function can be called from cust_main, as a final "cleanup" routine.

Function cleanup (custquery.4gl)

```
01 FUNCTION cleanup()
02   WHENEVER ERROR CONTINUE
03   CLOSE cust_curs
04   FREE cust_curs
05   WHENEVER ERROR STOP
06 END FUNCTION
```

Note:

- Line 03 Closes the cursor used to retrieve the database rows.
- Line 04 Frees the memory associated with the cursor.
- Lines 02 and 05 The WHENEVER ERROR statements prevent a program error if the user exited the program without querying, and the cursor was never created.

Error if Cursor is not Open

In the example program in this chapter, if the user selects the Next or Previous action from the MENU before he has queried, the program returns an error ("Program stopped at line .... Fetch attempted on unopened cursor").

One way to prevent this error would be to add a variable to the program to indicate whether the user has queried for a result set, and to prevent him from executing the actions associated with Next or Previous until he has done so.

Changes to function query_cust (custquery.4gl):

```
01 FUNCTION query_cust()
02   DEFINE cont_ok SMALLINT,
03          cust_cnt SMALLINT,
04          where_clause STRING
05   MESSAGE "Enter search criteria"
06   LET cont_ok = FALSE
07
...
08
09   IF (cont_ok = TRUE) THEN
```

```
10      CALL display_cust()
11   END IF
12
13   RETURN cont_ok
14
15 END FUNCTION
```

Note:

- Line 13 A single line is added to the `query_cust` function to return the value of `cont_ok`, which indicates whether the query was successful, to the calling function in `custmain.4gl`.

Changes to module `custmain.4gl`:

```
01 MAIN
02   DEFINE query_ok SMALLINT
03
04   DEFER INTERRUPT
05
06   CONNECT TO "custdemo"
07   CLOSE WINDOW SCREEN
08   OPEN WINDOW w1 WITH FORM "custform"
09   LET query_ok = FALSE
10
11   MENU "Customer"
12     ON ACTION query
13       CALL query_cust() RETURNING query_ok
14     ON ACTION next
15       IF (query_ok) THEN
16         CALL fetch_rel_cust(1)
17       ELSE
18         MESSAGE "You must query first."
19       END IF
20     ON ACTION previous
21       IF (query_ok) THEN
22         CALL fetch_rel_cust(-1)
23       ELSE
24         MESSAGE "You must query first."
25       END IF
26     ON ACTION quit
27       EXIT MENU
28   END MENU
29
30   CLOSE WINDOW w1
31   CALL cleanup()
32   DISCONNECT CURRENT
33
34 END MAIN
```

Note:

- Line 03 defines the variable `query_ok`, which will be used to indicate whether the user has queried.
- Line 09 sets the initial value of `query_ok` to FALSE.
- Line 13 the function `query_cust` now returns a value for `query_ok`.
- Lines 15 thru 19 and Lines 21 thru 25: these sections test the value of `query_ok` when **Next** or **Previous** has been selected. If `query_ok` is TRUE, the function `fetch_rel_cust` is called; otherwise, a message is displayed to the user.
- Line 31 calls the cleanup function to close the cursor used to fetch the database rows.

Tutorial Chapter 5: Enhancing the Form

Program forms can be displayed in a variety of ways. This chapter illustrates adding a toolbar or a topmenu (pull-down menu) by modifying the form specification file, changing the window's appearance, and disabling/enabling actions. The example programs in this chapter use some of the action defaults defined by Genero BDL to standardize the presentation of common actions to the user.

You can change the way that program options are displayed in a form in a variety of ways. This example program illustrates some of the simple changes that can be made:

- By changing the form specification file, you can provide the user with a valid list of abbreviations for the state field and add a toolbar or pull-down menu (topmenu). The program business logic in the BDL program need not change. Once you recompile the form file, it can be used by the program with no additional changes required.
- You can change the appearance of the application window, adding a custom title and icon.
- You can disable and enable actions dynamically to control the options available to the user.

The program also illustrates some of the Genero BDL action defaults that standardize the presentation of common actions.

Adding a Toolbar

A toolbar presents buttons on the form associated with actions defined by the current interactive BDL instruction in your program.

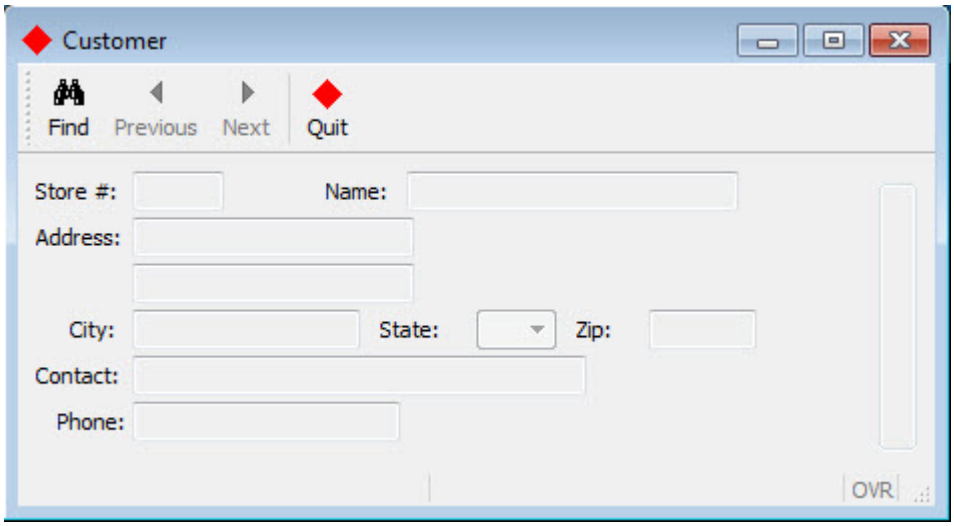

Figure 10: A form with a toolbar displayed on Windows™ platforms

The `TOOLBAR` section of a form specification file defines a toolbar with buttons that are bound to actions. A toolbar definition can contain the following elements:

- an `ITEM` - specifies the action that is bound to the toolbar button
- a `SEPARATOR` - a vertical line

Values can be assigned to `TEXT`, `COMMENT`, and `IMAGE` attributes for each item in the toolbar.

The toolbar commands are enabled by actions defined by the current interactive BDL instruction, which in our example is the `MENU` statement in the `custquery.4gl` module. When a toolbar button is selected by the user, the program triggers the action to which the toolbar button is bound.

Example: (in custform.per)

The toolbar in this example will display buttons for `find`, `next`, `previous`, and `quit` actions.

Form `custform.per`:

```
01 SCHEMA custdemo
02
03 TOOLBAR
04   ITEM find
05   ITEM previous
06   ITEM next
07   SEPARATOR
08   ITEM quit (TEXT="Quit", COMMENT="Exit the program", IMAGE="exit")
09 END
10
...
```

Note:

- Line `04` The `ITEM` command-identifier `find` will be bound to the `MENU` statement action `find` on line `14` in the `custmain.4gl` file. The word find must be identical in both the `TOOLBAR` `ITEM` and the `MENU` statement action, and must always be in lowercase. The other command-identifiers are similarly bound.

- Line 08 Although attributes such as TEXT or COMMENT are defined for the ITEM quit, the items find, previous, and next do not have any attributes defined in the form specification file. These actions are common actions that have default attributes defined in the action defaults file.

Adding a Topmenu

A topmenu presents a pull-down menu on a form, composed of actions defined by the current interactive BDL instruction in your program.

The same options that were displayed to the user as a toolbar can also be defined as buttons on a pull-down menu (a topmenu). To change the presentation of the menu options to the user, simply modify and recompile the form specification file.

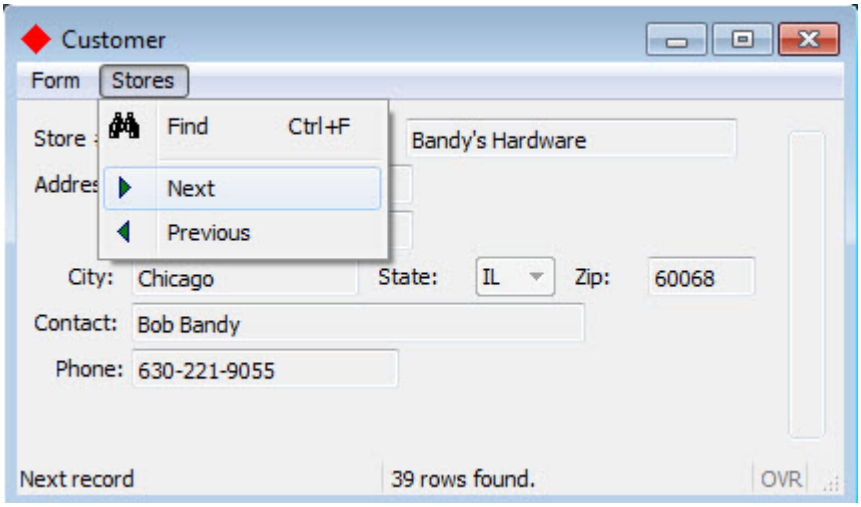

Figure 11: A form with a topmenu displayed on a Windows™ platform

The TOPMENU section of the form specification allows you to design the pull-down menu. The TOPMENU section must appear after SCHEMA, and must contain a tree of GROUP elements that define the pull-down menu. The GROUP TEXT value is the title for the pull-down menu group.

A GROUP can contain the following elements:

- a COMMAND - specifies the action the menu option must be bound to
- a SEPARATOR - a horizontal line
- GROUP children - a subgroup within a group.

Values can be assigned to attributes such as TEXT, COMMENT, and IMAGE for each item in the TOPMENU.

As in a toolbar, the TOPMENU commands are enabled by actions defined by the current interactive BDL instruction (dialog), which in our example is the MENU statement in the custquery.4gl module. When a TOPMENU option is selected by the user, the program triggers the action to which the TOPMENU command is bound.

Example (in custform.per)

The example shows a TOPMENU section in the form specification file (custform.per) for Chapter 5.

Form custform.per:

```
01 SCHEMA custdemo
02
03 TOPMENU
04   GROUP form (TEXT="Form")
05     COMMAND quit (TEXT="Quit", COMMENT="Exit the program", IMAGE="exit")
06   END
```

```
07  GROUP stores (TEXT="Stores")
08     COMMAND find
09     SEPARATOR
13     COMMAND next
14     COMMAND previous
15  END
16END
17
...
```

Note:

- Lines `04` and `07` This example `TOPMENU` will consist of two groups on the menu bar of the form. The `TEXT` displayed on the menu bar for the first group will be **Form**, and the second group will be **Stores**.
- Line `08` to `14` Under the menu bar item **Stores**, the command-identifier **find** on line`05` will be bound to the `MENU` statement action `find` on line `14` in the `custmain.4gl` file. The word `find` must be identical (including case) in both the `TOPMENU` command and the `MENU` statement action. The other command-identifiers are similarly bound.

The revised form specification file must be recompiled before it can be used in the program.

Adding a COMBOBOX form item

A combobox defines a dropdown box of values, allowing the user to select a value for the underlying formfield.

In this example application the only valid values for the state column of the database table customer are `IL`, `IA`, and `WI`. The form item used to display the state field can be changed to a `COMBOBOX` displaying a dropdown list of valid state values. The combobox is active during an `INPUT`, `INPUT ARRAY`, or `CONSTRUCT` statement, allowing the user to select a value for the state field.

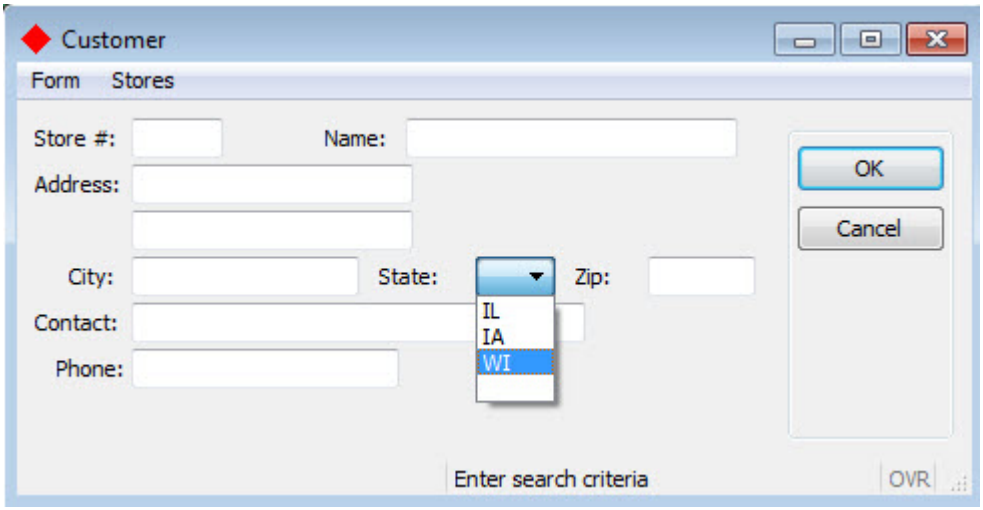

Figure 12: A form with a combobox

The values of the list are defined by the `ITEMS` attribute:

```
COMBOBOX f6=customer.state, ITEMS = ("IL", "IA", "WI");
```

In this example, the value displayed on the form and the real value (the value to be stored in the program variable corresponding to the form field) are the same. You can choose to define different display and real

values; in this example, the values `Paris`, `Madrid`, and `London` would be displayed to the user, but the value stored in the corresponding program variable would be `1`, `2`, or `3`:

```
COMBOBOX f9=formonly.cities, ITEMS=((1,"Paris"),(2,"Madrid"),(3,"London"));
```

Although the list of values for the combobox is contained in the form specification file in this example program, you could also set the `INITIALIZER` attribute to define a function that will provide the values. The initialization function would be invoked at runtime when the form is loaded, to fill the combobox item list dynamically with database records, for example.

Changing the Window Appearance

Genero provides attributes that can be used to customize the appearance of windows, forms, and form objects in your application. In addition, you can create Presentation Styles to standardize the appearance of window and form objects across applications.

Some of the simple changes that you can make are:

Title

The default title for a window is the name of the object in the `OPEN WINDOW` statement. For example, in the programs we've seen so far, the title of the window is `w1`:

```
OPEN WINDOW w1 WITH FORM "custform"
```

In the form specification file, the attribute `TEXT` of the `LAYOUT` section can be used to change the title of the parent window:

```
LAYOUT (TEXT="Customer")
```

Icon

The Genero runtime system provides built-in classes, or object templates, which contain methods, or functions, that you can call from your programs. The classes are grouped together into packages. One package, `ui`, contains the `Interface` class, allowing you to manipulate the user interface. For example, the `setImage` method can be used to set the default icon for the windows of your program. You may simply call the method, prefixing it with the package name and class name; you do not need to create an Interface object.

```
CALL ui.Interface.setImage("imagename")
```

Window Style

By default windows are displayed as normal application windows, but you can choose a specific style using the `WINDOWSTYLE` attribute of the `LAYOUT` section of the form file. The default window styles are defined as a set of attributes in an external file (`default.4st`).

```
LAYOUT (WINDOWSTYLE="dialog")
```

Example: (in custform.per)

This example shows how to define a combobox in a text-based form specification file (`custform.per`).

Form `custform.per`:

```
...
18 LAYOUT (TEXT="Customer")
19  GRID
20  {
21   Store #:[f01  ]       Name:[f02                ]
22   Address:[f03                 ]
23           [f04                 ]
24      City:[f05             ]State:[f6  ]Zip:[f07   ]
25   Contact:[f08                          ]
26     Phone:[f09                ]
27  }
28  END
29 END
30 TABLES
31   customer
32 END
33 ATTRIBUTES
34 EDIT f01=customer.store_num,
35   REQUIRED, COMMENT="This is the co-op store number";
36 EDIT f02=customer.store_name;
37 EDIT f03=customer.addr;
38 EDIT f04=customer.addr2;
39 EDIT f05=customer.city;
40 COMBOBOX f6=customer.state,
41   REQUIRED, ITEMS = ("IL", "IA", "WI");
41 EDIT f07=customer.zip-code;
42 EDIT f08=customer.contact_name;
43 EDIT f09=customer.phone;
43 END
```

Note:

- Line `18`, the title of the window is set to `Customer`. Since this is a normal application window, the default window style is used.
- Line `40`, a `COMBOBOX` is substituted for a simple `EDIT` form field.
- Line `35` and `41` The `REQUIRED` attribute forces the user to enter or select a value for this field when a new record is being added. See the attributes list for a complete list of the attributes that can be defined for a form field.

Example: (in custmain.4gl)

This example shows how to use the built-in class method `ui.Interface.setImage` to change the icon for the application windows.

Module `custmain.4gl`:

```
...
04 MAIN
05   DEFINE query_ok SMALLINT
06
07   DEFER INTERRUPT
08
09   CONNECT TO "custdemo"
10   CLOSE WINDOW SCREEN
```

```
11   CALL ui.Interface.setImage("smiley")
12   OPEN WINDOW w1 WITH FORM "custform"
13
...
```

Note:

- Line 11 For convenience, the image used is the smiley image from the `pics` directory, which is the default image directory of the Genero Desktop Client.

Managing Actions

Disable/Enable Actions

In the example in the previous lesson, if the user clicks the **Next** or **Previous** buttons on the application form without first querying successfully, a message displays and no action is taken. You can disable and enable the actions instead, providing visual cues to the user when the actions are not available.

The `ui.Dialog` built-in class provides an interface to the BDL interactive dialog statements, such as `CONSTRUCT` and `MENU`. The method `setActionActive` enables and disables actions. To call a method of this class, use the predefined `DIALOG` object within the interactive instruction block.

For example:

```
MENU
  ...
  BEFORE MENU
    CALL DIALOG.setActionActive("actionname" , state)
    ...

END MENU
```

where *actionname* is the name of the action, *state* is an integer, `0` (disable) or `1` (enable).

You must be within an interactive instruction in order to use the `DIALOG` object in your program, but you can pass the object to a function. Using this technique, you could create a function that enables/disables an action, and call the function from the `MENU` statement, for example. See *The Dialog class* in the *Genero Business Development Language User Guide* for further information.

The Close Action

In Genero applications, when the user clicks the

button in the upper-right corner of the application window, a predefined `close` action is sent to the program. What happens next depends on the interactive dialog statement.

- When the program is in a `MENU` dialog statement, the `close` action is converted to an `INTERRUPT` key press. If there is a `COMMAND KEY INTERRUPT` block in the `MENU` statement, the statements in that control block are executed. Otherwise, no action is taken.
- When the program is in an `INPUT`, `INPUT ARRAY`, `CONSTRUCT` or `DISPLAY ARRAY` statement, the `close` action cancels the dialog, and the `INT_FLAG` is set to `TRUE`. Your program can check the value of `INT_FLAG` and take appropriate action.

You can change this default behavior by overwriting the `close` action within the interactive statement. For example, to exit the `MENU` statement when the user clicks this button:

```
MENU
  ...
  ON ACTION close
    EXIT MENU
```

```
END MENU
```

By default the action view for the `close` action is hidden and does not display on the form.

Example: (custmain.4gl)

Calls to the `setActionActive` method from the `ui.Dialog` class have been added to `custmain.4gl` to disable and enable menu actions appropriately to give the user visual cues. An additional `ON ACTION` statement exits the menu if the user selects

.

Module `custmain.4gl`:

```
01
02 MAIN
03 DEFINE query_ok SMALLINT
04
05 DEFER INTERRUPT
06 CONNECT TO "custdemo"
07 CLOSE WINDOW SCREEN
08 CALL ui.Interface.setImage("smiley")
09 OPEN WINDOW w1 WITH FORM "custform"
10
11 LET query_ok = FALSE
12
13 MENU
14   BEFORE MENU
15     CALL DIALOG.setActionActive("next",0)
16     CALL DIALOG.setActionActive("previous",0)
17   ON ACTION find
18     CALL DIALOG.setActionActive("next",0)
19     CALL DIALOG.setActionActive("previous",0)
20     CALL query_cust() RETURNING query_ok
21     IF (query_ok) THEN
22       CALL DIALOG.setActionActive("next",1)
23       CALL DIALOG.setActionActive("previous",1)
24     END IF
25   ON ACTION next
26      CALL fetch_rel_cust(1)
27   ON ACTION previous
28     CALL fetch_rel_cust(-1)
29   ON ACTION quit
30     EXIT MENU
31   ON ACTION close
32     EXIT MENU
33 END MENU
34
35 CLOSE WINDOW w1
36
37 DISCONNECT CURRENT
38
39 END MAIN
```

Note:

- Line `08` The icon for the application windows is set to the "exit" image.
- Lines `15`, `16` Before the menu is first displayed, the `next` and `previous` actions are disabled.
- Lines `18`, `19` Before the `query_cust` function is executed the `next` and `previous` actions are disabled

- Lines 21 thru 24 If the query was successful the `next` and `previous` actions are enabled.
- Line 31 The `close` action is included in the menu, although an action view won't display on the form. If the user clicks the  in the top right of the window, the action on line 32, `EXIT MENU`, will be taken.

Action Defaults

The Genero BDL runtime system includes an XML file, `default.4ad`, in the `lib` subdirectory of the installation directory *FGLDIR*, that defines presentation attributes for some commonly used actions.

If you match the action names used in this file exactly when you define your action views (toolbar or topmenu items, buttons, etc.) in the form specification file, the presentation attributes defined for this action will be used. All action names must be in lowercase.

For example, the following line in the `default.4ad` file:

```
<ActionDefault name="find" text="Find"
               image="find" comment="Search" />
```

defines presentation attributes for a `find` action - the text to be displayed on the action view `find` defined in the form, the image file to be used as the icon for the action view, and the comment to be associated with the action view. The attribute values are case-sensitive, so the action name in the form specification file must be "find", not "Find".

The following line in the `default.4ad` file defines presentation attributes for the predefined action `cancel`. An accelerator key is assigned as an alternate means of invoking the action:

```
<ActionDefault name="cancel" text="Cancel"
               acceleratorName="Escape" />
```

You can override a default presentation attribute in your program. For example, by specifying a `TEXT` attribute for the action `find` in the form specification file, the default `TEXT` value of "Find" will be replaced with the value "Looking".

```
03 TOPMENU
04
...
07   GROUP stores (TEXT="Stores")
08    COMMAND find (TEXT="Looking")
```

You can create your own `.4ad` file to standardize the presentation attributes for all the common actions used by your application. See *Action defaults files* in the *Genero Business Development Language User Guide* for additional details.

MENU/Action Defaults Interaction

Attributes defined in the form specification file override attributes defined in the `.4ad` file.

The attributes of the action views for the `MENU` actions in the `custmain.4gl` example will be determined as shown in Table 3: custmain.4gl example actions on page 61.

Table 3: `custmain.4gl` example actions

Action	From the form specification file	From the default.4ad file	From the MENU statement in the .4gl file
find	No attributes listed	TEXT="Find" IMAGE="find" COMMENT="Search"	Overridden by default.4ad
next	No attributes listed	TEXT="Next" IMAGE="goforw" COMMENT="Next record"	Overridden by default.4ad
previous	No attributes listed	TEXT="Previous" IMAGE="gorev" COMMENT="Previous record"	Overridden by default.4ad
close	Not listed in the form file	attributes are listed in default.4ad but the action view is not displayed on form by default	Overridden by default.4ad (predefined action)
quit	For both TOPMENU and TOOLBAR, the action view has the attributes TEXT="Quit", COMMENT="Exit the program", IMAGE="exit".	Action is not listed in the file	Overridden by the form specification file.
accept	Not listed in the form file.	TEXT="OK" AcceleratorName="Return" AcceleratorName2="Enter"	This action is not defined in a MENU instruction (predefined action.)
cancel	Not listed in the form file.	TEXT="Cancel" AcceleratorName="Escape"	This action is not defined in a MENU instruction (predefined action.)

Note: The predefined actions `accept` and `cancel` do not have action views defined in the form specification file; by default, they appear on this form as buttons in the right-hand section of the form when the `CONSTRUCT` statement is active. Their attributes are taken from the `default.4ad` file.

Images

The image files specified in these definitions are among the files provided with the Genero Desktop Client, in the `pics` subdirectory.

Tutorial Chapter 6: Add, Update and Delete

This program allows the user to insert/update/delete rows in the customer database table. Embedded SQL statements (`UPDATE`/`INSERT`/`DELETE`) are used to update the table, based on the values stored in the program record. SQL transactions, concurrency, and consistency are discussed. A dialog window is displayed to prompt the user to verify the deletion of a row.

Entering data on a form: INPUT statement

The `INPUT` statement allows the user to enter or change the values in a program record, which can then be used as the data for new rows in a database table, or to update existing rows.

In the `INPUT` statement you list:

- The program variables that are to receive data from the form
- The corresponding form fields that the user will use to supply the data

```
INPUT <program-variables> FROM <form-fields>
```

The `FROM` clause explicitly binds the fields in the screen record to the program variables, so the `INPUT` instruction can manipulate values that the user enters in the screen record. The number of record members must equal the number of fields listed in the `FROM` clause. Each variable must be of the same (or a compatible) data type 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.

When invoked, the `INPUT` statement enables the specified fields of the form in the current BDL window, and waits for the user to supply data for the fields. The user moves the cursor from field to field and types new values. Each time the cursor leaves a field, the value typed into that field is deposited into the corresponding program variable. You can write blocks of code as clauses in the `INPUT` statement that will be called automatically during input, so that you can monitor and control the actions of your user within this statement.

The `INPUT` statement ends when the user selects the **accept** or **cancel** actions.

`INPUT` supports the same shortcuts for naming records as the `DISPLAY` statement. You can ask for input to all members of a record, from all fields of a screen record, and you can ask for input `BY NAME` from fields that have the same names as the program variables.

```
INPUT BY NAME <program record>.*
```

UNBUFFERED attribute

By default, field values are buffered. The `UNBUFFERED` attribute makes the `INPUT` dialog "sensitive", allowing you to easily change some form field values programmatically during `INPUT` execution.

When you assign a value to a program variable, the runtime system will automatically display that value in the form; when you input values in a form field, the runtime system will automatically store that value in the corresponding program variable. Using the `UNBUFFERED` attribute is strongly recommended.

WITHOUT DEFAULTS attribute

An `INPUT` with the `WITHOUT DEFAULTS` attribute can be used to allow the user to make changes to an existing program record representing a row in the database.

The same `INPUT` statement can be used, with the `WITHOUT DEFAULTS` attribute, to allow the user to make changes to an existing program record representing a row in the database. This attribute prevents BDL from automatically displaying any default values that have been defined for the form fields when `INPUT` is invoked, allowing you to display the existing database values on the screen before the user begins editing the data. In this case, when the `INPUT` statement is used to allow the user to add a new row, any existing values in the program record must first be nulled out. Note however that the `REQUIRED` attribute is ignored when `WITHOUT DEFAULTS` is `TRUE`. If you want to use `REQUIRED`, for example to force the end user to visit all required fields and fire the `AFTER FIELD` trigger to validate the entered data, you can turn off or on the `WITHOUT DEFAULTS` attribute according to the need, by using a Boolean expression.

Updating Database Tables

The values of the program variables that have been input through the form can be used in SQL statements that update tables in a database.

SQL transactions

The embedded SQL statements `INSERT`, `UPDATE`, and `DELETE` can be used to make changes to the contents of a database table.

If your database has transaction logging, you can use the `BEGIN WORK` and `COMMIT WORK` commands to delimit a transaction block, usually consisting of multiple SQL statements. 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. At runtime, the Genero database driver generates the appropriate SQL commands to be used with the target database server.

To eliminate concurrency problems, keep transactions as short as possible.

Concurrency and Consistency

While your program is modifying data, another program may also be reading or modifying the same data. To prevent errors, database servers use a system of locks.

When another program requests the data, the database server either makes the program wait or turns it back with an error. BDL provides a combination of statements to control the effect that locks have on your data access:

```
SET LOCK MODE TO {WAIT [n]| NOT WAIT }
```

This defines the timeout for lock acquisition for the current connection. The timeout period can be specified in seconds (*n*). If no period is specified, the timeout is infinite. If the `LOCK MODE` is set to `NOT WAIT`, an exception is returned immediately if a lock cannot be acquired.

> **Important:** This feature is not supported by all databases. When possible, the database driver sets the corresponding connection parameter to define the timeout. If the database server does not support setting the lock timeout parameter, the runtime system generates an exception.

```
SET ISOLATION LEVEL TO { DIRTY READ
                       | COMMITTED READ
                       | CURSOR STABILITY
                       | REPEATABLE READ }
```

This defines the `ISOLATION LEVEL` for the current connection. When possible, the database driver executes the native SQL statement that corresponds to the specified isolation level.

For portable database programming, the following is recommended:

- Transactions must be enabled in your database.
- The `ISOLATION LEVEL` must be at least `COMMITTED READ`. On most database servers, this is usually the default isolation level and need not be changed.
- The `LOCK MODE` must be set to `WAIT` or `WAIT` *`time period`*, if this is supported by your database server.

See *Database transactions* in the *Genero Business Development Language User Guide* for a more complete discussion.

The *SQL adaptation guides* provide detailed information about the behavior of specific database servers.

Adding a new row

The `INPUT` statement provides control blocks to allow your program to initialize field contents and validate user input when adding a new row.

INPUT Statement Control blocks

Control blocks `BEFORE FIELD` and `ON CHANGE` are called automatically during an `INPUT` as the user moves the cursor through the fields of a form.

For example:

- `BEFORE FIELD` control blocks are executed immediately prior to the focus moving to the specified field. The example program uses this control block to prevent the user from changing the store number during an Update, by immediately moving the focus to the store name field (the `NEXT FIELD` instruction).
- An `ON CHANGE` is used to verify the uniqueness of the store number that was entered, and to make sure that the store name is not left blank. The user receives notification of a problem with the value of a field as soon as the field is exited. Validating these values as they are completed is less disruptive than notifying the user of several problems after the entire record has been entered.

See *INPUT control blocks* in the *Genero Business Development Language User Guide* for a complete list of `INPUT` control blocks.

Example: add a new row to the customer table

New functions are added to the `custmain.4gl` and `custquery.4gl` modules to allow users to add rows to the customer table.

Module custmain.4gl

The `MENU` statement in the module `custmain.4gl` is modified to call functions for adding, updating, and deleting the rows in the customer table.

The `MAIN` block (`custmain.4gl`)

```
01 -- custmain.4gl
02
03 MAIN
04   DEFINE query_ok INTEGER
05
06   DEFER INTERRUPT
07   CONNECT TO "custdemo"
08   SET LOCK MODE TO WAIT 6
09   CLOSE WINDOW SCREEN
10   OPEN WINDOW w1 WITH FORM "custform"
11
12   MENU
13    ON ACTION find
```

```
14      LET query_ok = query_cust()
15    ON ACTION next
16      IF query_ok THEN
17        CALL fetch_rel_cust(1)
18      ELSE
19        MESSAGE"You must query first."
20      END IF
21    ON ACTION previous
22      IF query_ok THEN
23        CALL fetch_rel_cust(-1)
24      ELSE
25        MESSAGE "You must query first."
26      END IF
27    COMMAND "Add"
28      IF inpupd_cust("A") THEN
29        CALL insert_cust()
30      END IF
31    COMMAND "Delete"
32      IF delete_check() THEN
33        CALL delete_cust()
34      END IF
35    COMMAND "Modify"
36      IF inpupd_cust("U") THEN
37        CALL update_cust()
38      END IF
39    ON ACTION quit
40      EXIT MENU
41   END MENU
42
43   CLOSE WINDOW w1
44
45   DISCONNECT CURRENT
46
47 END MAIN
```

Note:

- Line `08` sets the lock timeout period to 6 seconds.
- Lines `12` thru `41` define the main menu of the program.
- Lines `27` thru `30` The `MENU` option **Add** now calls an `inpupd_cust` function. Since this same function will also be used for updates, the value "A", indicating an Add of a new row, is passed. If `inpupd_cust` returns `TRUE`, the `insert_cust` function is called.
- Lines `31` thru `34` The `MENU` option **Delete** now calls a `delete_check` function. If `delete_check` returns `TRUE`, the `delete_cust` function is called.
- Lines `35` thru `38` are added to the `MENU` statement for the **Modify** option, calling the `inpud_cust` function. The value "U", for an Update of a new row, is passed as a parameter. If `inpupd_cust` returns `TRUE`, the `update_cust` function is called.

Module custquery.4gl (function inpupd_cust)

A new function, `inpupd_cust`, is added to the `custquery.4gl` module, allowing the user to insert values for a new customer row into the form.

Function `inpupd_cust` (`custquery.4gl`):

```
01 FUNCTION inpupd_cust(au_flag)
02   DEFINE au_flag CHAR(1),
03          cont_ok SMALLINT
04
05   LET cont_ok = TRUE
06
07
```

```
08   IF (au_flag = "A") THEN
09     MESSAGE "Add a new customer"
10     INITIALIZE mr_custrec.* TO NULL
12   END IF
13
14   LET INT_FLAG = FALSE
15
16   INPUT BY NAME mr_custrec.*
17         WITHOUT DEFAULTS ATTRIBUTES(UNBUFFERED)
18
19   ON CHANGE store_num
20     IF (au_flag = "A") THEN
21       SELECT store_name,
22              addr,
23              addr2,
24              city,
25              state,
26              zip-code,
27              contact_name,
28              phone
29         INTO mr_custrec.*
30         FROM customer
31         WHERE store_num = mr_custrec.store_num
32       IF (SQLCA.SQLCODE = 0)THEN
33         ERROR "Store number already exists."
34         LET cont_ok = FALSE
35         CALL display_cust()
36         EXIT INPUT
37       END IF
38     END IF
39
40   AFTER FIELD store_name
41     IF (mr_custrec.store_name IS NULL) THEN
42       ERROR "You must enter a company name."
43       NEXT FIELD store_name
44     END IF
45
46   END INPUT
47
48   IF (INT_FLAG) THEN
49     LET INT_FLAG = FALSE
50     LET cont_ok = FALSE
51     MESSAGE "Operation cancelled by user"
52     INITIALIZE mr_custrec.* TO NULL
53   END IF
54
55   RETURN cont_ok
56
57 END FUNCTION
```

- Line `01` The function accepts a parameter defined as `CHAR(1)`. In order to use the same function for both the input of a new record and the update of an existing one, the `CALL` to this function in the `MENU` statement in `main.4gl` will pass a value "A" for add, and "U" for update.
- Line `06` The variable `cont_ok` is a flag to indicate whether the update operation should continue; set initially to `TRUE`.
- Lines `08` thru `12` test the value of the parameter `au_flag`. If the value of `au_flag` is "A" the operation is an Add of a new record, and a `MESSAGE` is displayed. Since this is an Add, the modular program record values are initialized to `NULL` prior to calling the `INPUT` statement, so the user will have empty form fields in which to enter data.
- Line `14` sets the `INT_FLAG` global variable to `FALSE` prior to the `INPUT` statement, so the program can determine if the user cancels the dialog.

- Line 17 The UNBUFFERED and WITHOUT DEFAULTS clauses of the INPUT statement are used. The UNBUFFERED attribute insures that the program array the screen array of the form are automatically synchronized for input and output. The WITHOUT DEFAULTS clause is used since this statement will also implement record updates, to prevent the existing values displayed on the form from being erased or replaced with default values.
- Lines 19 thru 38 Each time the value in store_num changes, the customer table is searched to see if that store_num already exists. If so, the values in the mr_custrec record are displayed in the form, the variable cont_ok is set to FALSE, and the INPUT statement is immediately terminated.
- Lines 40 thru 44 The AFTER FIELD control block verifies that store_name was not left blank. If so, the NEXT FIELD statement returns the focus to the store_name field so the user may enter a value.
- Line 46 END INPUT is required when any of the optional control blocks of the INPUT statement are used.
- Lines 48 thru 53 The INT_FLAG is checked to see if the user has canceled the input. If so, the variable cont_ok is set to FALSE, and the program record mr_custrec is set to NULL . The UNBUFFERED attribute of the INPUT statement assures that the NULL values in the program record are automatically displayed on the form.
- Line 55 returns the value of cont_ok, indicating whether the input was successful.

Module custquery.4gl (function insert_cust)

A new function, insert_cust, in the custquery.4gl module, contains the logic to add the new row to the customer table.

Function insert_cust:

```
01 FUNCTION insert_cust()
02
03   WHENEVER ERROR CONTINUE
04   INSERT INTO customer (
05     store_num,
06     store_name,
07     addr,
08     addr2,
09     city,
10     state,
11     zip-code,
12     contact_name,
13     phone
14     )VALUES (mr_custrec.*)
15   WHENEVER ERROR STOP
16
17   IF (SQLCA.SQLCODE = 0) THEN
18     MESSAGE "Row added"
19   ELSE
20     ERROR SQLERRMESSAGE
21   END IF
22
23 END FUNCTION
```

Note:

- Lines 04 thru 14 contain an embedded SQL statement to insert the values in the program record mr_custrec into the customer table. This syntax can be used when the order in which the members of the program record were defined matches the order of the columns listed in the SELECT statement. Otherwise, the individual members of the program record must be listed separately. Since there is no BEGIN WORK / COMMIT WORK syntax used here, this statement will be treated as a singleton transaction and the database driver will automatically send the appropriate COMMIT statement. The INSERT statement is surrounded by WHENEVER ERROR statements.

- Lines 17 thru 21 test the SQLCA.SQLCODE that was returned from the INSERT statement. If the INSERT was not successful, the corresponding error message is displayed to the user.

Updating an existing Row

Updating an existing row in a database table provides more opportunity for concurrency and consistency errors than inserting a new row. Use techniques shown in this section to help minimize the errors.

Using a work record

A work record and a local record, both identical to the program record, are defined to allow the program to compare the values.

1. A SCROLL CURSOR is used to allow the user to scroll through a result set generated by a query. The scroll cursor is declared WITH HOLD so it will not be closed when a COMMIT WORK or ROLLBACK WORK is executed.
2. When the user chooses **Update**, the values in the current program record are copied to the work record.
3. The INPUT statement accepts the user's input and stores it in the program record. The WITHOUT DEFAULTS keywords are used to insure that the original values retrieved from the database were not replaced with default values.
4. If the user accepts the input, a transaction is started with BEGIN WORK.
5. The primary key stored in the program record is used to SELECT the same row into the local record. FOR UPDATE locks the row.
6. The SQLCA.SQLCODE is checked, in case the database row was deleted after the initial query.
7. The work record and the local record are compared, in case the database row was changed after the initial query.
8. If the work and local records are identical, the database row is updated using the new program record values input by the user.
9. If the UPDATE is successful, a COMMIT WORK is issued. Otherwise, a ROLLBACK WORK is issued.
10. The SCROLL CURSOR has remained open, allowing the user to continue to scroll through the query result set.

SELECT ... FOR UPDATE

To explicitly lock a database row prior to updating, a SELECT...FOR UPDATE statement may be used to instruct the database server to lock the row that was selected. SELECT ... FOR UPDATE cannot be used outside of an explicit transaction. The locks are held until the end of the transaction.

SCROLL CURSOR WITH HOLD

Like many programs that perform database maintenance, the Query program uses a SCROLL CURSOR to move through an SQL result set, updating or deleting the rows as needed. BDL cursors are automatically closed by the database interface when a COMMIT WORK or ROLLBACK WORK statement is performed. To allow the user to continue to scroll through the result set, the SCROLL CURSOR can be declared WITH HOLD, keeping it open across multiple transactions.

Example: Updating a Row in the customer table

Functions are modified in the `custquery.4gl` module to allow users to update existing rows in the `customer` table.

Module custquery.4gl

The module has been modified to define a `work_custrec` record that can be used as working storage when a row is being updated.

Module `custquery.4gl`:

```
01
02 SCHEMA custdemo
03
04 DEFINE  mr_custrec, work_custrec RECORD
05     store_num    LIKE customer.store_num,
06     store_name   LIKE customer.store_name,
07     addr         LIKE customer.addr,
08     addr2        LIKE customer.addr2,
09     city         LIKE customer.city,
10     state        LIKE customer.state,
11     zip-code      LIKE customer.zip-code,
12     contact_name LIKE customer.contact_name,
13     phone        LIKE customer.phone
14    END RECORD
...
```

Note:

- Lines `04` thru `15` define a `work_custrec` record that is modular in scope and contains the identical structure as the `mr_custrec` program record.

The function `inpupd_cust` in the `custquery.4gl` module has been modified so it can also be used to obtain values for the Update of existing rows in the `customer` table.

Function `inpupd_cust` (`custquery.4gl`)

```
01 FUNCTION inpupd_cust(au_flag)
02   DEFINE au_flag CHAR(1),
03          cont_ok SMALLINT
04
05   INITIALIZE work_custrec.* TO NULL
06   LET cont_ok = TRUE
07
08   IF (au_flag = "A") THEN
09     MESSAGE "Add a new customer"
10     LET mr_custrec.* = work_custrec.*
11   ELSE
12     MESSAGE "Update customer"
13     LET work_custrec.* = mr_custrec.*
14   END IF
15
16   LET INT_FLAG = FALSE
17
18   INPUT BY NAME mr_custrec.*
19     WITHOUT DEFAULTS ATTRIBUTES(UNBUFFERED)
20
21     BEFORE FIELD store_num
22       IF (au_flag = "U") THEN
23         NEXT FIELD store_name
24       END IF
25
26     ON CHANGE store_num
```

```
27       IF (au_flag = "A") THEN
...
28     AFTER FIELD store_name
29       IF (mr_custrec.store_name IS NULL) THEN
...
30
31   END INPUT
```

Note:

- Line 05 sets the work_custrec program record to NULL.
- Line 10 For an **Add**, the mr_custrec program record is set equal to the work_custrec record, in effect setting mr_custrec to NULL. The LET statement uses less resources than INITIALIZE.
- Line 13 For an **Update**, the values in the mr_custrec program record are copied into work_custrec, saving them for comparison later.
- Lines 21 thru 24 A BEFORE FIELD store_num clause has been added to the INPUT statement. If this is an Update, the user should not be allowed to change store_num, and the NEXT FIELD instruction moves the focus to the store_name field.
- Line 26 The ON CHANGE store_num control block, which will only execute if the au_flag is set to "A" (the operation is an Add) remains the same.
- Line 28 The AFTER FIELD store_name control block remains the same, and will execute if the operation is an **Add** or an **Update**.

A new function update_cust in the custquery.4gl module updates the row in the customer table.

Function update_cust (custquery.4gl)

```
01 FUNCTION update_cust()
02   DEFINE l_custrec RECORD
03     store_num    LIKE customer.store_num,
04     store_name   LIKE customer.store_name,
05     addr         LIKE customer.addr,
06     addr2        LIKE customer.addr2,
07     city         LIKE customer.city,
08     state        LIKE customer.state,
09     zip-code     LIKE customer.zip-code,
10     contact_name LIKE customer.contact_name,
11     phone        LIKE customer.phone
12     END RECORD,
13     cont_ok INTEGER
14
15   LET cont_ok = FALSE
16
17   BEGIN WORK
18
19   SELECT store_num,
20          store_name,
21          addr,
22          addr2,
23          city,
24          state,
25          zip-code,
26          contact_name,
27          phone
28    INTO l_custrec.* FROM customer
29    WHERE store_num = mr_custrec.store_num
30    FOR UPDATE
31
32   IF (SQLCA.SQLCODE = NOTFOUND) THEN
33     ERROR "Store has been deleted"
34     LET cont_ok = FALSE
```

```
35   ELSE
36     IF (l_custrec.* = work_custrec.*) THEN
37       WHENEVER ERROR CONTINUE
38       UPDATE customer SET
39         store_name = mr_custrec.store_name,
40         addr = mr_custrec.addr,
41         addr2 = mr_custrec.addr2,
42         city = mr_custrec.city,
43         state = mr_custrec.state,
44         zip-code = mr_custrec.zip-code,
45         contact_name = mr_custrec.contact_name,
46         phone = mr_custrec.phone
47         WHERE store_num = mr_custrec.store_num
48       WHENEVER ERROR STOP
49       IF (SQLCA.SQLCODE = 0) THEN
50         LET cont_ok = TRUE
51         MESSAGE "Row updated"
52       ELSE
53         LET cont_ok = FALSE
54         ERROR SQLERRMESSAGE
55       END IF
56     ELSE
57       LET cont_ok = FALSE
58       LET mr_custrec.* = l_custrec.*
59       MESSAGE "Row updated by another user."
60     END IF
61   END IF
62
63   IF (cont_ok = TRUE) THEN
64     COMMIT WORK
65   ELSE
66     ROLLBACK WORK
67   END IF
68
69 END FUNCTION
```

- Lines 02 thru 12 define a local record, l_custrec with the same structure as the modular program records mr_custrec and work_custrec.
- Line 15 The cont_ok variable will be used as a flag to determine whether the update should be committed or rolled back.
- Line 17 Since this will be a multiple-statement transaction, the BEGIN WORK statement is used to start the transaction.
- Lines 19 thru 30 use the store_num value in the program record to re-select the row. FOR UPDATE locks the database row until the transaction ends.
- Lines 32 thru 34 check SQLCA.SQLCODE to make sure the record has not been deleted by another user. If so, an error message is displayed, and the variable cont_ok is set to FALSE.
- Lines 36 thru 60 are to be executed if the database row was found.
- Line 36 compares the values in the l_custrec local record with the work_custrec record that contains the original values of the database row. All the values must match for the condition to be TRUE.
- Lines 37 thru 55 are executed if the values matched. An embedded SQL statement is used to UPDATE the row in the customer table using the values which the user has previously entered in the mr_custrec program record. The SQL UPDATE statement is surrounded by WHENEVER ERROR statements. The SQLCA.SQLCODE is checked after the UPDATE, and if it indicates the update was not successful the variable cont_ok is set to FALSE and an error message is displayed.
- Lines 57 through 59 are executed if the values in l_custrec and work_custrec did not match. The variable cont_ok is set to FALSE. The values in the mr_custrec program record are set to the values in the l_custrec record (the current values in the database row, retrieved by the SELECT ... FOR UPDATE statement.) The UNBUFFERED attribute of the INPUT statement assures that the values will be

automatically displayed in the form. A message is displayed indicating the row had been changed by another user.

- Lines 63 thru 67 If the variable `cont_ok` is `TRUE` (the update was successful) the program issues a `COMMIT WORK` to end the transaction begun on Line 17. If not, a `ROLLBACK WORK` is issued. All locks placed on the database row are automatically released.

Deleting a Row

The SQL `DELETE` statement can be used to delete rows from the database table. The primary key of the row to be deleted can be obtained from the values in the program record.

Using a dialog Menu to prompt for validation

The `MENU` statement has an optional `STYLE` attribute that can be set to 'dialog', automatically opening a temporary modal window. You can also define a message and icon with the `COMMENT` and `IMAGE` attributes. This provides a simple way to prompt the user to confirm some action or operation that has been selected.

The menu options appear as buttons at the bottom of the window. Unlike standard menus, the dialog menu is automatically exited after any action clause such as `ON ACTION`, `COMMAND` or `ON IDLE`. You do not need an `EXIT MENU` statement.

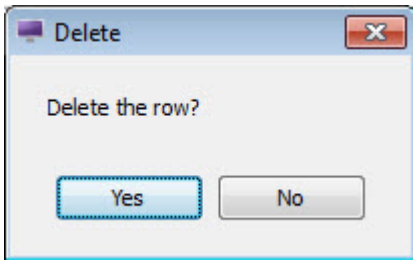

Figure 13: Using a dialog Menu

Example: Deleting a Row

Function `delete_check` is added to the `custquery.4gl` module to check whether a store has any orders in the database before allowing the user to delete the store from the `customer` table. If there are no existing orders, a dialog `MENU` is used to prompt the user for confirmation.

Function delete_check (`custquery.4gl`)

```
01 FUNCTION delete_check()
02   DEFINE del_ok SMALLINT,
03          ord_count SMALLINT
04
05   LET del_ok = FALSE
06
07   SELECT COUNT(*) INTO ord_count
08     FROM orders
09     WHERE orders.store_num =
10           mr_custrec.store_num
11
12   IF ord_count > 0 THEN
13     MESSAGE "Store has existing orders"
14   ELSE
15     MENU "Delete" ATTRIBUTES (STYLE="dialog",
16                   COMMENT="Delete the row?")
17      COMMAND "Yes"
18        LET del_ok = TRUE
19      COMMAND "No"
```

```
20        MESSAGE "Delete canceled"
21      END MENU
22   END IF
23
24   RETURN del_ok
25
26 END FUNCTION
```

- Line 02 defines a variable del_ok to be used as a flag to determine if the delete operation should continue.
- Line 05 sets del_ok to FALSE.
- Lines 07 thru 10 use the store_num value in the mr_custrec program record in an SQL statement to determine whether there are orders in the database for that store_num. The variable ord_count is used to store the value returned by the SELECT statement.
- Lines 12 thru 13 If the count is greater than zero, there are existing rows in the orders table for the store_num. A message is displayed to the user. del_ok remains set to FALSE.
- Lines 15thru 21 If the count is zero, the delete operation can continue. A MENU statement is used to prompt the user to confirm the Delete action. The STYLE attribute is set to "dialog" to automatically display the MENU in a modal dialog window. If the user selects **Yes**, the variable del_ok is set to TRUE. Otherwise a message is displayed to the user indicating the delete will be canceled.
- Line 24 returns the value of del_ok to the delete_cust function.

The function delete_cust is added to the custquery.4gl module to delete the row from the customer table.

Function delete_cust (custquery.4gl)

```
01 FUNCTION delete_cust()
02
03   WHENEVER ERROR CONTINUE
04   DELETE FROM customer
05     WHERE store_num = mr_custrec.store_num
06   WHENEVER ERROR STOP
07   IF SQLCA.SQLCODE = 0 THEN
08     MESSAGE "Row deleted"
09     INITIALIZE mr_custrec.* TO NULL
10   ELSE
11     ERROR SQLERRMESSAGE
12   END IF
13
14 END FUNCTION
```

Note:

- Lines 04 and 05 contains an embedded SQL DELETE statement that uses the store_num value in the program record mr_custrec to delete the database row. The SQL statement is surrounded by WHENEVER ERROR statements. This is a singleton transaction that will be automatically committed if it is successful.
- Lines 07 thru 12 check the SQLCA.SQLCODE returned for the SQL DELETE statement. If the DELETE was successful, a message is displayed and the mr_custrec program record values are set to NULL and automatically displayed on the form. Otherwise, an error message is displayed.

Tutorial Chapter 7: Array Display

The example in this chapter displays multiple `customer` records at once. The `disparray` program defines a program array to hold the records, and displays the records in a form containing a table and a screen array. The example program is then modified to dynamically fill the array as needed. This program illustrates a library function - the example is written so it can be used in multiple programs, maximizing code reuse.

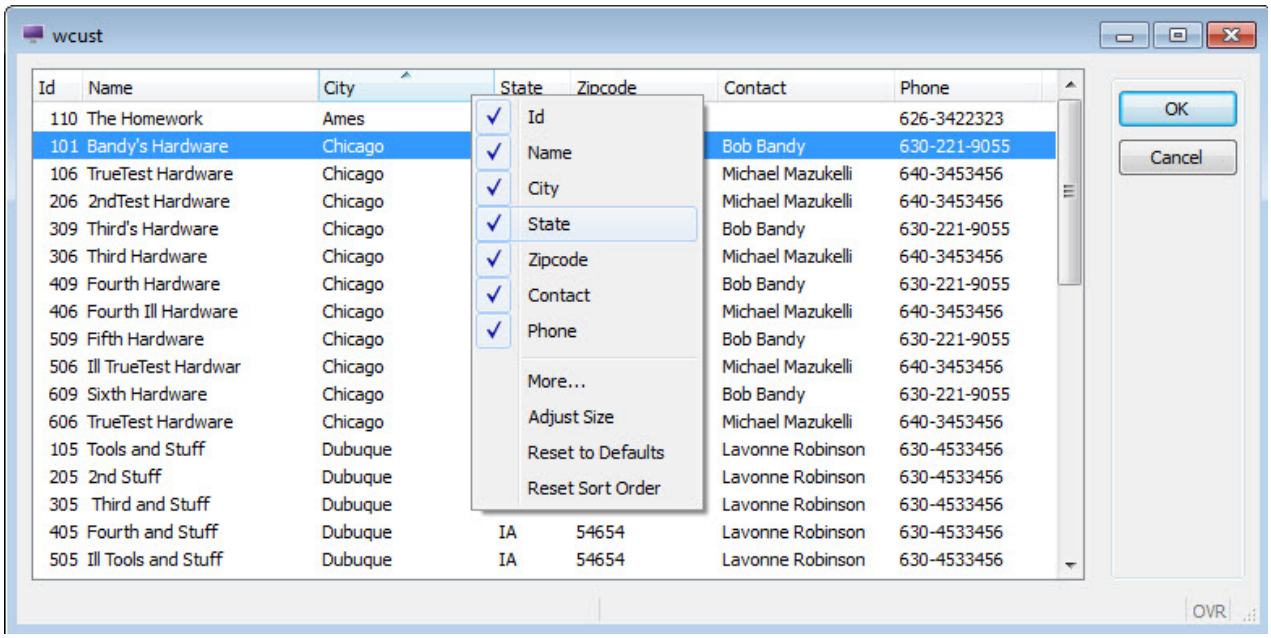

Figure 14: Array Display

In the illustration, the table is sorted by `City`. A right mouse click has displayed a dropdown list of the columns, with checkboxes allowing the user to hide or show a specific column. After the user validates the row selected, the store number and store name are returned to the calling function.

Defining the Form

The scrolling list of customer records demonstrated in this chapter requires a form specification file containing a screen array of screen records.

Screen Arrays

In a text-based form specification file (`.per`), a screen array is usually a repetitive array of fields in the `LAYOUT` section, each containing identical groups of screen fields.

Each "row" of a screen array is a screen record. Each "column" of a screen array consists of fields with the same item tag in the `LAYOUT` section of the form specification file. You must declare screen arrays in the `INSTRUCTIONS` section.

TABLE Containers

The TABLE container in a form defines the presentation of a list of records, bound to a screen array.

When this layout container is used with curly braces defining the container area, the position of the static labels and item tags is automatically detected by the form compiler to build a graphical object displaying a list of records.

The first line of the TABLE area contains text entries defining the column titles. The second line contains field item tags that define the columns of the table receiving the data. This line is repeated to allow the display of multiple records at once.

The user can sort the rows displayed in the form table by a mouse-click on the title of the column that is to be used for the sort. This sort is performed on the client side only. The columns and the entire form can be stretched and re-sized. A right-mouse-click on a column title displays a dropdown list-box of column names, with radio buttons allowing the user to indicate whether a specific column is to be hidden or shown.

The INSTRUCTIONS section

You must declare a screen array in the INSTRUCTIONS section of a text-based form specification file (.per) with the SCREEN RECORD keyword. You can reference the names of the screen array in the DISPLAY ARRAY statement of the program.

Form example: manycust.per

The manycust.per form specification file contains a TABLE Container and screen array used to display a list of customer rows

Form manycust.per:

```
01 SCHEMA custdemo
02
03 LAYOUT
04  TABLE
05  {
06   Id   Name           ...   Zip-code   Contact          Phone
07  [f01][f02           ]     [f05     ][f06            ][f07         ]
08  [f01][f02           ]     [f05     ][f06            ][f07         ]
09  [f01][f02           ]     [f05     ][f06            ][f07         ]
10  [f01][f02           ]     [f05     ][f06            ][f07         ]
11  [f01][f02           ]     [f05     ][f06            ][f07         ]
12  [f01][f02           ]     [f05     ][f06            ][f07         ]
13  }
14  END
15 END
16
17 TABLES
18   customer
19 END
20
21 ATTRIBUTES
22 EDIT f01=customer.store_num;
23 EDIT f02=customer.store_name;
24 EDIT f03=customer.city;
25 EDIT f04=customer.state;
26 EDIT f05=customer.zip-code;
27 EDIT f06=customer.contact_name;
28 EDIT f07=customer.phone;
29 END
30
31 INSTRUCTIONS
32 SCREEN RECORD sa_cust (customer.*);
33 END
```

Note:

In order to fit on the page, the layout section of the form is truncated, not displaying the `city` and `state` columns.

- Line `01` The `custdemo` schema will be used by the compiler to determine the data types of the form fields.
- Line `06` contains the titles for the columns in the `TABLE`.
- Lines `07` thru `12` define the display area for the screen records. These rows must be identical in a `TABLE`. (The fields for `city` and `state` are indicated by .... so the layout will fit on this page.)
- Line `21` thru `29` In the `ATTRIBUTES` section the field item tags are linked to the field description. Although there are multiple occurrences of each item tags in the form, the description is listed only once for each unique field item tag.
- Line `32` defines the screen array in the `INSTRUCTIONS` section. The screen record must contain the same number of elements as the records in the `TABLE` container. This example defines the screen record with all fields defined with the `customer` prefix, but you can list each field name individually.

Creating the Function

The main module, `cust_stub.4gl` calls the library function `display_custarr`, which uses a cursor with a `FOREACH` statement to load rows from the customer table into a program array. The `DISPLAY ARRAY` statement displays the records in the program array to the screen array defined in the form specification file.

Program Arrays

A program array is an ordered set of elements all of the same data type. You can create one-, two-, or three-dimensional arrays. The elements of the array can be simple types or they can be records.

Arrays can be:

- static - defined with an explicit size for all dimensions.
- dynamic - has a variable size. Dynamic arrays have no theoretical size limit.

All elements of static arrays are initialized even if the array is not used. Therefore, defining huge static arrays may use a lot of memory. The elements of dynamic arrays are allocated automatically by the runtime system, as needed.

Example of a dynamic array of records definition:

```
01 DEFINE cust_arr DYNAMIC ARRAY OF RECORD
02                    store_num LIKE customer.store_num,
03                    city      LIKE customer.city
04                 END RECORD
```

This array variable is named `cust_arr`; each element of the array contains the members `store_num` and `city`. The size of the array will be determined by the runtime system, based on the program logic that is written to fill the array. The first element of any array is indexed with subscript 1. You would access the `store_num` member of the 10th element of the array by writing `cust_arr[10].store_num`.

Loading the Array: the FOREACH Statement

The FOREACH statement is equivalent to using the OPEN, FETCH and CLOSE statements to retrieve and process all the rows selected by a query, and is especially useful when loading arrays.

To load the program array in the example, you must retrieve the values from the result set of a query and load them into the elements of the array. You must DECLARE the cursor before the FOREACH statement can retrieve the rows.

```
01 DECLARE custlist_curs CURSOR FOR
02        SELECT store_num, city FROM customer
03  CALL cust_arr.clear()
04  FOREACH custlist_curs INTO cust_rec.*
05      CALL cust_arr.appendElement()
06      LET cust_arr[cust_arr.getLength()].* = cust_rec.*
07  END FOREACH
```

The FOREACH statement shown:

1. Opens the `custlist_curs` cursor.
2. Clears the `cust_arr` array.
3. Fetches a row into the record `cust_rec`. This record must be defined as having the same structure as a single element of the `cust_arr` array (`store_num`, `city`).
4. Appends an empty element to the `cust_arr` array.
5. Copies the `cust_rec` record into the array `cust_arr` using the `getLength` method to determine the index of the element that was newly appended to the array.
6. Repeats steps 3, 4 and 5 until no more rows are retrieved from the database table (automatically checks for the NOTFOUND condition).
7. Closes the cursor and exits from the FOREACH loop.

The DISPLAY ARRAY Statement

The DISPLAY ARRAY statement lets the user view the contents of an array of records, scrolling through the display.

The example defines a program array of records, each record having members that correspond to the fields of the screen records defined in the form specification file. The DISPLAY ARRAY statement displays all the records in the program array into the rows of the screen array. Typically the program array has many more rows of data than will fit on the screen.

The COUNT attribute

When using a static array, the number of rows to be displayed is defined by the COUNT attribute. If you do not use the COUNT attribute, the runtime system cannot determine how much data to display, and so the screen array remains empty.

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.

Example:

```
01  DISPLAY ARRAY cust_arr TO sa_cust.*
```

This statement will display the program array `cust_arr` to the form fields defined in the `sa_cust` screen array of the form.

By default, the DISPLAY ARRAY statement does not terminate until the user accepts or cancels the dialog; the `Accept` and `Cancel` actions are predefined and display on the form. Your program can accept the dialog instead, using the ACCEPT DISPLAY instruction.

The ARR_CURR function

When the user accepts or cancels a dialog, the ARR_CURR built-in function returns the index (subscript number) of the row in the program array that was selected (current).

Example Library module: cust_lib.4gl

The `cust_lib.4gl` module contains the library function `display_custarr`, that can be reused by other programs that reference the `customer` table.

Module `cust_lib.4gl`:

```
01 SCHEMA custdemo
02
03 FUNCTION display_custarr()
04
05   DEFINE cust_arr DYNAMIC ARRAY OF RECORD
06          store_num    LIKE customer.store_num,
07          store_name   LIKE customer.store_name,
08          city         LIKE customer.city,
09          state        LIKE customer.state,
10          zip-code      LIKE customer.zip-code,
11          contact_name LIKE customer.contact_name,
12          phone        LIKE customer.phone
13    END RECORD,
14          cust_rec RECORD
15          store_num    LIKE customer.store_num,
16          store_name   LIKE customer.store_name,
17          city         LIKE customer.city,
18          state        LIKE customer.state,
19          zip-code      LIKE customer.zip-code,
20          contact_name LIKE customer.contact_name,
21          phone        LIKE customer.phone
22    END RECORD,
23    ret_num LIKE customer.store_num,
24    ret_name LIKE customer.store_name,
25    curr_pa SMALLINT
26
27    OPEN WINDOW wcust WITH FORM "manycust"
28
29    DECLARE custlist_curs CURSOR FOR
30      SELECT store_num,
31             store_name,
32             city,
33             state,
34             zip-code,
35             contact_name,
36             phone
37        FROM customer
38        ORDER BY store_num
39
40
41   CALL cust_arr.clear()
42   FOREACH custlist_curs INTO cust_rec.*
43      CALL cust_arr.appendElement()
44      LET cust_arr[cust_arr.getLength()].* = cust_rec.*
45   END FOREACH
46
47   LET ret_num = 0
48   LET ret_name = NULL
49
50   IF (cust_arr.getLength() > 0) THEN
51      DISPLAY ARRAY cust_arr TO sa_cust.*
52      IF NOT INT_FLAG THEN
```

```
53         LET curr_pa = arr_curr()
54         LET ret_num = cust_arr[curr_pa].store_num
55         LET ret_name = cust_arr[curr_pa].store_name
56      END IF
57   END IF
58
59   CLOSE WINDOW wcust
60   RETURN ret_num, ret_name
61
62 END FUNCTION
```

Note:

- Lines `05` thru `13` define a local program array, `cust_arr`.
- Lines `14` thru `22` define a local program record, `cust_rec`. This record is used as temporary storage for the row data retrieved by the `FOREACH` loop in line 42.
- Lines `23` and `24`define local variables to hold the store number and name values to be returned to the calling function.
- Line `25` defines a variable to store the value of the program array index.
- Line `27` opens a window with the form containing the array.
- Lines `29` thru `38` `DECLARE` the cursor `custlist_curs` to retrieve the rows from the `customer` table.
- Line `40` sets the variable `idx` to 0, this variable will be incremented in the `FOREACH` loop.
- Line `41` clear the dynamic array.
- Line `42` `uses FOREACH` to retrieve each row from the result set into the program record, `cust_rec`.
- Lines `43` thru `44` are executed for each row that is retrieved by the `FOREACH`. They append a new element to the array `cust_arr`, and transfer the data from the program record into the new element, using the method `getLength` to identify the index of the element. When the `FOREACH` statement has retrieved all the rows the cursor is closed and the `FOREACH` is exited.
- Lines `47` and `48` Initialize the variables used to return the customer number and customer name.
- Lines `50` thru `57` If the length of the `cust_arr` array is greater than 0, the `FOREACH` statement did retrieve some rows.
- Line `52` `DISPLAY ARRAY` turns control over to the user, and waits for the user to accept or cancel the dialog.
- Line `52` The `INT_FLAG` variable is tested to check if the user validated the dialog.
- Line `53` If the user has validated the dialog, the built-in function `ARR_CURR` is used to store the index for the program array element the user had selected (corresponding to the highlighted row in the screen array) in the variable `curr_pa`.
- Lines `54` and `55` The variable `curr_pa` is used to retrieve the current values of `store_num` and `store_name` from the program array and store them in the variables `ret_num` and `ret_name`.
- Line `59` closes the window.
- Line `60` returns `ret_num` and `ret_name` to the calling function.

Paged Mode of DISPLAY ARRAY

The previous example retrieves all the rows from the customer table into the program array prior to the data being displayed by the `DISPLAY ARRAY` statement. Using this full list mode, you must copy into the array all the data you want to display. Using the `DISPLAY ARRAY` statement in paged mode allows you to provide data rows dynamically during the dialog, using a dynamic array to hold one page of data.

The following example modifies the program to use a `SCROLL CURSOR` to retrieve only the `store_num` values from the customer table. As the user scrolls thru the result set, statements in the `ON FILL BUFFER` clause of the `DISPLAY ARRAY` statement are used to retrieve and display the remainder of each row, a

page of data at a time. This helps to minimize the possibility that the rows have been changed, since the rows are re-selected immediately prior to the page being displayed.

What is the Paged mode?

The paged mode allows a program to display a very large number of rows without copying all the rows into the program array at once. The program array holds only the current visible page.

A "page" of data is the total number of rows of data that can be displayed in the form at one time. The length of a page can change dynamically, since the user has the option of resizing the window containing the form. The runtime system automatically keeps track of the current length of a page.

The `ON FILL BUFFER` clause feeds the `DISPLAY ARRAY` instruction with pages of data. The following built-in functions are used in the `ON FILL BUFFER` clause to provide the rows of data for the page:

- `FGL_DIALOG_GETBUFFER START()` - retrieves the offset in the `SCROLL CURSOR` result set, and is used to determine the starting point for retrieving and displaying the complete rows.
- `FGL_DIALOG_GETBUFFERLENGTH()` - retrieves the current length of the page, and is used to determine the number of rows that must be provided.

The statements in the `ON FILL BUFFER` clause of `DISPLAY ARRAY` are executed automatically by the runtime system each time a new page of data is needed. For example, if the current size of the window indicates that ten rows can be displayed at one time, the statements in the `ON FILL BUFFER` clause will automatically maintain the dynamic array so that the relevant ten rows are retrieved and/or displayed as the user scrolls up and down through the table on the form. If the window is re-sized by the user, the statements in the `ON FILL BUFFER` clause will automatically retrieve and display the new number of rows.

AFTER DISPLAY block

The `AFTER DISPLAY` block is executed one time, after the user has accepted or canceled the dialog, but before executing the next statement in the program.

In this program, the statements in this block determine the current position of the cursor when the user presses **OK** or **Cancel**, so the correct `store number` and `name` can be returned to the calling function.

Example of paged mode

In the first example, the records in the `customer` table are loaded into the program array and the user uses the form to scroll through the program array. In this example, the user is actually scrolling through the result set created by a `SCROLL CURSOR`. This `SCROLL CURSOR` retrieves only the store number, and another SQL `SELECT` statement is used to retrieve the remainder of the row as needed.

Module `cust_lib2.4gl`:

```
01 SCHEMA custdemo
02
03 FUNCTION display_custarr()
04
05  DEFINE cust_arr DYNAMIC ARRAY OF RECORD
06         store_num     LIKE customer.store_num,
07         store_name    LIKE customer.store_name,
08         city          LIKE customer.city,
09         state         LIKE customer.state,
10         zip-code       LIKE customer.zip-code,
11         contact_name  LIKE customer.contact_name,
12         phone         LIKE customer.phone
13     END RECORD,
14     ret_num      LIKE customer.store_num,
15     ret_name     LIKE customer.store_name,
16     ofs, len, i  SMALLINT,
17      sql_text     STRING,
18      rec_count    SMALLINT,
```

```
    curr_pa      SMALLINT

OPEN WINDOW wcust WITH FORM "manycust"

LET rec_count = 0
SELECT COUNT(*) INTO rec_count FROM customer
IF (rec_count == 0) THEN
   RETURN 0, NULL
END IF

LET sql_text =
   "SLECT tore_num, store_name, city,"
   || " state, zip-code, contact_name,"
   || " phone"
   || " FROM customer WHERE store_num = ?"
PREPARE rec_all FROM sql_text

DECLARE num_curs SCROLL CURSOR FOR
       SELECT store_num FROM customer
OPEN num_curs

DISPLAY ARRAY cust_arr TO sa_cust.*
     ATTRIBUTES(UNBUFFERED, COUNT=rec_count)

  ON FILL BUFFER
    LET ofs = FGL_DIALOG_GETBUFFERSTART()
    LET len = FGL_DIALOG_GETBUFFERLENGTH()
    FOR i = 1 TO len
      WHENEVER ERROR CONTINUE
     FETCH ABSOLUTE ofs+i-1 num_curs
                  INTO cust_arr[i].store_num
       EXECUTE rec_all INTO cust_arr[i].*
                USING cust_arr[i].store_num
      WHENEVER ERROR STOP
      IF (SQLCA.SQLCODE = NOTFOUND) THEN
        MESSAGE "Row deleted by another user."
        CONTINUE FOR
      ELSE
        IF (SQLCA.SQLCODE < 0) THEN
          ERROR SQLERRMESSAGE
          CONTINUE FOR
        END IF
      END IF
    END FOR

  AFTER DISPLAY
    IF (INT_FLAG) THEN
       LET ret_num = 0
       LET ret_name = NULL
    ELSE
       LET curr_pa = ARR_CURR()- ofs + 1
       LET ret_num = cust_arr[curr_pa].store_num
       LET ret_name = cust_arr[curr_pa].store_name
    END IF

END DISPLAY

CLOSE num_curs
FREE num_curs
FREE rec_all

CLOSE WINDOW wcust
RETURN ret_num, ret_name

```

```
83 END FUNCTION
```

Note:

- Lines 16 thru 19 define some new variables to be used, including `cont_disp` to indicate whether the function should continue.
- Line 24 uses an embedded SQL statement to store the total number of rows in the `customer` table in the variable `rec_count`.
- Lines 25 thru 27 If the total number of rows is zero, function returns immediately `0` and `NULL`.
- Lines 29 thru 33 contain the text of an SQL `SELECT` statement to retrieve values from a single row in the `customer` table. The `?` placeholder will be replaced with the `store number` when the statement is executed. This text is assigned to a string variable, `sql_text`.
- Line 34 uses the SQL `PREPARE` statement to convert the string into an executable statement, `rec_all`. This statement will be executed when needed, to populate the rest of the values in the row of the program array.
- Lines 36 thru 37 `DECLARE` a `SCROLL CURSOR` `num_curs` to retrieve only the `store number` from the `customer` table.
- Line 38 opens the `SCROLL CURSOR` `num_curs`.
- Lines 40 and 41 call the `DISPLAY ARRAY` statement, providing the `COUNT` to let the statement know the total number of rows in the SQL result set.
- Lines 43 thru 62 contain the logic for the `ON FILL BUFFER` clause of the `DISPLAY ARRAY` statement. This control block will be executed automatically whenever a new page of data is required.
- Line 44 uses the built-in function to get the offset for the page, the starting point for the retrieval of rows, and stores it in the variable `ofs`.
- Line 45 uses the built-in function to get the page length, and stores it in the variable `len`.
- Lines 46 thru 62 contain a `FOR` loop to populate each row in the page with values from the `customer` table. The variable `i` is incremented to populate successive rows. The first value of `i` is 1.
- Lines 48 and 49 use the `SCROLL CURSOR num_curs` with the syntax `FETCH ABSOLUTE` <*row_number*> to retrieve the store number from a specified row in the result set, and to store it in row *i* of the program array. Since `i` was started at 1, the following calculation is used to determine the row number of the row to be retrieved:

  ```
  (Offset for the page) PLUS iMINUS 1
  ```

  Notice that rows 1 thru (*page_length*) of the program array are filled each time a new page is required.
- Lines 50 and 51 execute the prepared statement `rec_all` to retrieve the rest of the values for row *i* in the program array, using the `store number` retrieved by the `SCROLL CURSOR`. Although this statement is within the `FOR` loop, it was prepared earlier in the program, outside of the loop, to avoid unnecessary reprocessing each time the loop is executed.
- Lines 53 thru 61 test whether fetching the entire row was successful. If not, a message is displayed to the user, and the `CONTINUE FOR` instruction continues the `FOR` loop with the next iteration.
- Lines 64 thru 72 use an `AFTER DISPLAY` statement to get the row number of the row in the array that the user had selected. If the dialog was canceled, `ret_num` is set to 0 and `ret_name` is set to blanks. Otherwise the values of `ret_num` and `ret_name` are set based on the row number. The row number in the `SCROLL CURSOR` result set does not correlate directly to the program array number, because the program array was filled starting at row 1 each time. So the following calculation is used to return the correct row number of the program array:

  ```
  (Row number returned by ARR_CURR) MINUS
  (Offset for the page) PLUS 1
  ```

- Line 74 is the end of the `DISPLAY ARRAY` statement.

- Lines 76 and 77 CLOSE and FREE the cursor.
- Line 78 frees the prepared statement.
- Line 81 closes the window.
- Line 82 returns the values of the variables ret_num and ret_name to the calling function.

Compiling and using a Library

Since this is a function that could be used by other programs that reference the customer table, the function will be compiled into a library. The library can then be linked into any program, and the function called.

The function will always return store_num and store_name. If the FOREACH fails, or returns no rows, the calling program will have a store_num of zero and a NULL store_name returned.

The function is contained in a file named cust_lib.4gl. This file would usually contain additional library functions. To compile (and link, if there were additional .4gl files to be included in the library):

```
fgl2p -o cust_lib.42x cust_lib.4gl
```

Since a library has no MAIN function, we will need to create a small stub program if we want to test the library function independently. This program contains the minimal functionality to test the function.

Example: cust_stub.4gl

The module cust_stub.4gl calls the library function display_custarr in cust_lib.4gl.

Module cust_stub.4gl:

```
01 SCHEMA custdemo
02
03 MAIN
04   DEFINE store_num LIKE customer.store_num,
05          store_name LIKE customer.store_name
06
07  DEFER INTERRUPT
08  CONNECT TO "custdemo"
09  CLOSE WINDOW SCREEN
10
11  CALL display_custarr()
12           RETURNING store_num, store_name
13  DISPLAY store_num, store_name
14
15  DISCONNECT CURRENT
16
17 END MAIN
```

Note:

- Lines 04 and 05 define variables to hold the values returned by the display_custarr function.
- Lines 07 thru 09 are required simply for the test program, to set the program up and connect to the database.
- Line 11 calls the library function display_custarr.
- Line 13 displays the returned values to standard output for the purposes of the test.

Now we can compile the form file and the test program, and link the library, and then test to see if it works properly.

```
fglform manycust.per
fgl2p -o test.42r cust_stub.4gl cust_lib.42x
```

```
fglrun test.42r
```

Tutorial Chapter 8: Array Input

The program in this chapter allows the user to view and change a list of records displayed on a form. As each record in the program array is added, updated, or deleted, the program logic makes corresponding changes in the rows of the corresponding database table.

This program uses a form and a screen array to allow the user to view and change multiple records of a program array at once. The `INPUT ARRAY` statement and its control blocks are used by the program to control and monitor the changes made by the user to the records. As each record in the program array is Added, Updated, or Deleted, the program logic makes corresponding changes in the rows of the `customer` database table.

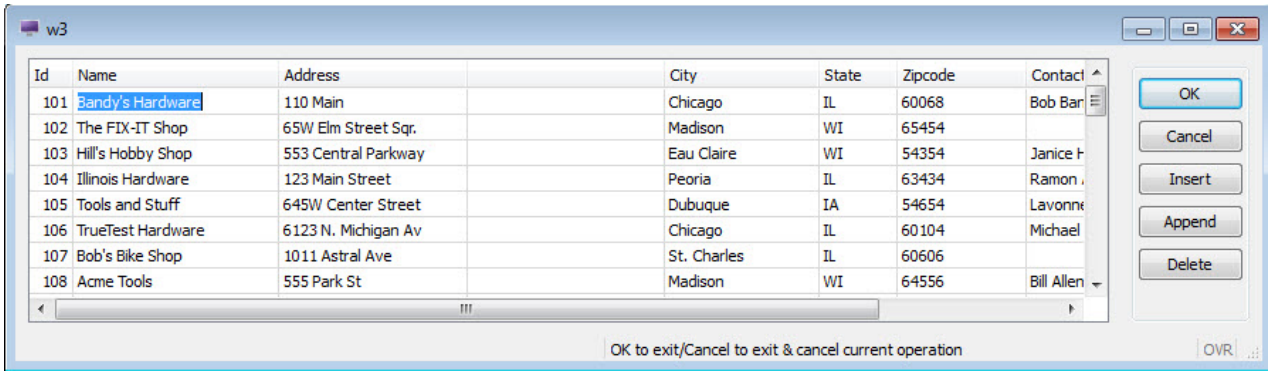

Figure 15: INPUT ARRAY example on a Windows™ platform

The INPUT ARRAY statement

The `INPUT ARRAY` statement supports data entry by users into a screen array, and stores the entered data in a program array of records. During the `INPUT ARRAY` execution, the user can edit or delete existing records, insert new records, and move inside the list of records. The program can then use the `INSERT`, `DELETE` or `UPDATE` SQL statements to modify the appropriate database tables. The `INPUT ARRAY` statement does not terminate until the user validates or cancels the dialog.

```
INPUT ARRAY cust_arr WITHOUT DEFAULTS FROM sa_cust.*
      ATTRIBUTES (UNBUFFERED)
```

The example `INPUT ARRAY` statement binds the screen array fields in `sa_cust` to the member records of the program array `cust_arr`. 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). Each mapped variable must have the same data type or a compatible data type as the corresponding field.

WITHOUT DEFAULTS clause

The `WITHOUT DEFAULTS` clause instructs the `INPUT ARRAY` statement to use and display the rows currently stored in the program array. Without this clause, the `INPUT ARRAY` would start with an empty list.

When creating a new row with the `insert` or `append` action, the `REQUIRED` attribute is always taken into account by `INPUT ARRAY`, even if the `WITHOUT DEFAULTS` clause is used.

The `WITHOUT DEFAULTS` clause prevents BDL from displaying any default values that have been defined for form fields. You must use this clause if you want to see the values of the program array.

The UNBUFFERED attribute

As in the `INPUT` statement, when the `UNBUFFERED` attribute is used, the `INPUT ARRAY` statement is sensitive to program variable changes.

If you need to display new data during the 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. This sensitivity applies to `ON ACTION` control blocks, as well: Before executing the code corresponding to the invoked action, the content of the field is converted and assigned to the corresponding program variable.

COUNT and MAXCOUNT attributes

`INPUT ARRAY` supports the `COUNT` and `MAXCOUNT` attributes to manage program arrays.

- The `COUNT` attribute of `INPUT ARRAY` defines the number of valid rows in the program array to be displayed as default rows.
  - When using a static array, if you do not use the `COUNT` attribute, the runtime system cannot determine how much data to display, so the screen array remains empty.
  - When using a dynamic array, the `COUNT` attribute is ignored: The number of elements in the dynamic array is used.
- The `MAXCOUNT` attribute defines the maximum number of data rows that can be entered in the program array. In a dynamic array, the user can enter an infinite number of rows if the `MAXCOUNT` attribute is not set.

Control Blocks

Your program can control and monitor the changes made by the user by using control blocks with the `INPUT ARRAY` statement.

The control blocks that are used in the example program are:

- The `BEFORE INPUT` block - executed one time, before the runtime system gives control to the user. You can implement initialization in this block.
- The `BEFORE ROW` block - executed each time the user moves to another row, after the destination row is made the current one.
- The `ON ROW CHANGE` block - executed when the user moves to another row after modifications have been made to the current row.
- The `ON CHANGE` <*fieldname*> block - executed when the cursor leaves a specified field and the value was changed by the user after the field got the focus.
- The `BEFORE INSERT` block - executed each time the user inserts a new row in the array, before the new row is created and made the current one.

- The `AFTER INSERT` block - executed each time the user inserts a new row in the array, after the new row is created. You can cancel the insert operation with the `CANCEL INSERT` keywords.
- The `BEFORE DELETE` block - executed each time the user deletes a row from the array, before the row is removed from the list. You can cancel the delete operation with the `CANCEL DELETE` keywords.
- The `AFTER ROW` block - 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.

For a more detailed explanation of the priority of control blocks see *INPUT control blocks* in the *Genero Business Development Language User Guide*.

Built-in Functions - ARR_CURR

The language provides several built-in functions to use in an `INPUT ARRAY` statement. The example program uses the `ARR_CURR` function to tell which array element is being changed. This function returns the row number within the program array that is displayed in the current line of a screen array.

Predefined actions

There are some predefined actions that are specific to the `INPUT ARRAY` statement, to handle the insertion and deletion of rows in the screen array automatically.

- The `insert` action inserts a new row before current row. When the user has filled this record, BDL inserts the data into the program array.
- The `delete` action deletes the current record from the display of the screen array and from the program array, and redraws the screen array so that the deleted record is no longer shown.
- The `append` action adds a new row at the end of the list. When the user has filled this record, BDL inserts the data into the program array.

As with the predefined actions `accept` and `cancel` actions discussed in Chapter 4, if your form specification does not contain action views for these actions, default action views (buttons on the form) are automatically created. Control attributes of the `INPUT ARRAY` statement allow you to prevent the creation of these actions and their accompanying buttons.

Example: Using a Screen Array to modify Data

The `arrayinput` program in chapter 8 uses the `INPUT ARRAY` statement with a Screen Array to allow the user to modify data in the `customer` table.

The Form Specification File

The `custallform.per` form specification file displays multiple records at once, and is similar to the form used in chapter 7. The item type of field `f6`, containing the `state` values, has been changed to `COMBOBOX` to provide the user with a dropdown list when data is being entered.

Form `custallform.per`:

```
01 SCHEMA custdemo
02
03 LAYOUT
04  TABLE
05  {
06   Id   Name          ..  Zip-code   Contact          Phone
07  [f01][f02          ]   [f07     ][f08            ][f09         ]
08  [f01][f02          ]   [f07     ][f08            ][f09         ]
09  [f01][f02          ]   [f07     ][f08            ][f09         ]
10  [f01][f02          ]   [f07     ][f08            ][f09         ]
```

```
 [f01][f02         ]   [f07     ][f08            ][f09         ]
 [f01][f02         ]   [f07     ][f08            ][f09         ]
 }
END
END

TABLES
 customer
END

ATTRIBUTES
EDIT f01 = customer.store_num, REQUIRED;
EDIT f02 = customer.store_name, REQUIRED;
EDIT f03 = customer.addr;
EDIT f04 = customer.addr2;
EDIT f05 = customer.city;
COMBOBOX f6  = customer.state, ITEMS = ("IA", "IL", "WI");
EDIT f07 = customer.zip-code;
EDIT f08 = customer.contact_name;
EDIT f09 = customer.phone;
END

INSTRUCTIONS
SCREEN RECORD sa_cust (customer.*);
END
```

The Main block

The single module program `custall.4gl` allows the user to update the `customer` table using a form that displays multiple records at once.

Main block (`custall.4gl`):

```
SCHEMA custdemo

DEFINE cust_arr DYNAMIC ARRAY OF RECORD
        store_num    LIKE customer.store_num,
        store_name   LIKE customer.store_name,
        addr         LIKE customer.addr,
        addr2       LIKE customer.addr2,
        city         LIKE customer.city,
        state       LIKE customer.state,
        zip-code      LIKE customer.zip-code,
        contact_name LIKE customer.contact_name,
        phone        LIKE customer.phone
       END RECORD


MAIN
  DEFINE idx SMALLINT

  DEFER INTERRUPT
  CONNECT TO "custdemo"
  CLOSE WINDOW SCREEN
  OPEN WINDOW w3 WITH FORM "custallform"

 CALL load_custall() RETURNING idx
 IF idx > 0 THEN
    CALL inparr_custall()
 END IF

  CLOSE WINDOW w3
  DISCONNECT CURRENT
```

```
31
32 END MAIN
```

Note:

- Lines 03 thru 13 define a dynamic array `cust_arr` having the same structure as the `customer` table. The array is modular is scope.
- Line 17 defines a local variable `idx`, to hold the returned value from the `load_custall` function.
- Line 20 connects to the `custdemo` database.
- Line 22 opens a window with the form `manycust`. This form contains a screen array `sa_cust` which is referenced in the program.
- Line 24 thru 27call the function `load_custall` to load the array, which returns the index of the array. If the load was successful (the returned index is greater than 0) the function `inparr_custall` is called. This function contains the logic for the Input/Update/Delete of rows.
- Line 29 closes the window.
- Line 30 disconnects from the database.

Function load_custall

This function loads the program array with rows from the `customer` database table.

The logic to load the rows is identical to that in Chapter 7. Although this program loads all the rows from the `customer` table, the program could be written to allow the user to query first, for a subset of the rows. A query-by-example, as illustrated in chapter 4, can also be implemented using a form containing a screen array such as `manycust`.

Function `load_custall` (`custall.4gl`):

```
01 FUNCTION load_custall()
02  DEFINE cust_rec RECORD LIKE customer.*
03
04
05   DECLARE custlist_curs CURSOR FOR
06      SELECT store_num,
07          store_name,
08          addr,
09          addr2,
10          city,
11          state,
12          zip-code,
13          contact_name,
14          phone
15        FROM customer
16        ORDER BY store_num
17
18
19   CALL cust_arr.clear()
20   FOREACH custlist_curs INTO cust_rec.*
21      CALL cust_arr.appendElement()
22      LET cust_arr[cust_arr.getLength()].* = cust_rec.*
23  END FOREACH
24
25   IF (cust_arr.getLength() == 0) THEN
26      DISPLAY "No rows loaded."
27   END IF
28
29   RETURN cust_arr.getLength()
30
31END FUNCTION
```

Note:

- Line 02 defines a local record variable, cust_rec, to hold the rows fetched in FOREACH.
- Lines 05 thru 16 declare the cursor custlist_curs to retrieve the rows from the customer table.
- Lines 20 thru 23 retrieve the rows from the result set into the program array.
- Lines 25 thru 27 If the array is empty, we display a warning message.
- Line 29 returns the number of rows to the MAIN function.

Function inparr_custall

This is the primary function of the program, driving the logic for inserting, deleting, and changing rows in the customer database table.

Each time a row in the array on the form is added, deleted, or changed, the values from the corresponding row in the program array are used to update the customer database table. The variable opflag is used by the program to indicate the status of the current operation.

- N - no action; set in the BEFORE ROW control block; this will subsequently be changed if an insert or update of a row in the array is performed.
- T - temporary; set in the BEFORE INSERT control block; indicates that an insert of a new row has been started.
- I - insert; set in the AFTER INSERT control block; indicates that the insert of the new row was completed.
- U - update; set in the ON ROW CHANGE control block; indicates that a change has been made to an existing row.

The value of opflag is tested in an AFTER ROW control block to determine whether an SQL INSERT or SQL UPDATE of the database table is performed.

This example illustrates how the order of execution of the control blocks is used by the program to set the opflag variable appropriately:

Function inparr_custall (custall.4gl):

```
01 FUNCTION inparr_custall(idx)
02
03   DEFINE curr_pa SMALLINT,
04          opflag  CHAR(1)
05
06  INPUT ARRAY cust_arr WITHOUT DEFAULTS
07      FROM sa_cust.*
08      ATTRIBUTES (UNBUFFERED)
09
10   BEFORE INPUT
11   MESSAGE "OK exits/" ||
12    "Cancel exits & cancels current operation"
13
14  BEFORE ROW
15     LET curr_pa = ARR_CURR()
16     LET opflag = "N"
17
18   BEFORE INSERT
19     LET opflag = "T"
20
21   AFTER INSERT
22     LET opflag = "I"
23
24   BEFORE DELETE
25     IF NOT (delete_cust(curr_pa)) THEN
26         CANCEL DELETE
27       END IF
```

```
28
29   ON ROW CHANGE
30      IF (opflag <> "I") THEN
31        LET opflag = "U"
32      END IF
33
34  BEFORE FIELD store_num
35   IF (opflag <> "T") THEN
36     NEXT FIELD store_name
37   END IF
38
39  ON CHANGE store_num
40   IF (opflag = "T") THEN
41       IF NOT store_num_ok(curr_pa) THEN
42        MESSAGE "Store already exists"
43       LET cust_arr[curr_pa].store_num = NULL
44       NEXT FIELD store_num
45      END IF
46    END IF
47
48    AFTER ROW
49      IF (INT_FLAG) THEN EXIT INPUT END IF
50     CASE
51     WHEN opflag = "I"
52         CALL insert_cust(curr_pa)
53       WHEN opflag = "U"
54          CALL update_cust(curr_pa)
55      END CASE
56
57  END INPUT
58
59  IF (INT_FLAG) THEN
60   LET INT_FLAG = FALSE
61  END IF
62
63 END FUNCTION -- inparr_custall
```

Note:

- Line `03` defines the variable `curr_pa`, to hold the index number of the current record in the program array.
- Line `04` defines the variable `opflag`, to indicate whether the operation being performed on a record is an Insert ("I") or an Update ("U").
- Lines `06` thru`57` contain the `INPUT ARRAY` statement, associating the program array `cust_arr` with the `sa_cust` screen array on the form. The attribute `WITHOUT DEFAULTS` is used to use and display existing records of the program array. The `UNBUFFERED` attribute insures that the program array the screen array of the form are automatically synchronized for input and output.
- Lines `10` thru `12` `BEFORE INPUT` control block: before the `INPUT ARRAY` statement is executed a `MESSAGE` is displayed to the user.
- Lines `14` thru `16` `BEFORE ROW` control block: when called in this block, the `ARR_CURR` function returns the index of the record that the user is moving into (which will become the current record). This is stored in a variable `curr_pa`, so the index can be passed to other control blocks. We also initialize the `opflag` to "N": This will be its value unless an update or insert is performed.
- Lines `18` and `19` `BEFORE INSERT` control block: just before the user is allowed to enter the values for a new record, the variable `opflag` is set to "T", indicating an Insert operation is in progress.
- Lines `21` and `22` `AFTER INSERT` control block sets the `opflag` to "I" after the insert operation has been completed.

- Lines 24 thru 27 `BEFORE DELETE` control block: Before the record is removed from the program array, the function `delete_cust` is called, which verifies that the user wants to delete the current record. In this function, when the user verifies the delete, the index of the record is used to remove the corresponding row from the database. Unless the `delete_cust` function returns `TRUE`, the record is not removed from the program array.
- Lines 29 thru32 `ON ROW CHANGE` control block: After row modification, the program checks whether the modification was an insert of a new row. If not, the `opflag` is set to "U" indicating an update of an existing row.
- Lines 34 thru 37 `BEFORE FIELD store_num` control block: the `store_num` field should not be entered by the user unless the operation is an Insert of a new row, indicated by the "T" value of `opflag`. The `store_num` column in the `customer` database table is a primary key and cannot be updated. If the operation is not an insert, the `NEXT FIELD` statement is used to move the cursor to the next field in the program array, `store_name`, allowing the user to change all the fields in the record of the program array except `store_num`.
- Lines 39 thru 46 `ON CHANGE store_num` control block: if the operation is an Insert, the `store_num_ok` function is called to verify that the value that the user has just entered into the field `store_num` of the current program array does not already exist in the `customer` database table. If the `store number` does exist, the value entered by the user is nulled out, and the cursor is returned to the `store_num` field.
- Lines 48 thru 55 `AFTER ROW` control block: First, the program checks `INT_FLAG` to see whether the user wants to interrupt the `INPUT` operation. If not, the `opflag` is checked in a `CASE` statement, and the `insert_cust` or `update_cust` function is called based on the `opflag` value. The index of the current record is passed to the function so the database table can be modified.
- Line 57 indicates the end of the `INPUT` statement.
- Lines 59 thru 61 check the value of the interrupt flag `INT_FLAG` and reset it to `FALSE` if necessary.

Function store_num_ok

When a new record is being inserted into the program array, this function verifies that the store number does not already exist in the `customer` database table. The logic in this function is virtually identical to that used in Chapter 5.

Function `store_num_ok` (`custall.4gl`):

```
01 FUNCTION store_num_ok(idx)
02   DEFINE idx SMALLINT,
03          checknum LIKE customer.store_num,
04          cont_ok SMALLINT
05
06   LET cont_ok= FALSE
07   WHENEVER ERROR CONTINUE
08   SELECT store_num INTO checknum
09    FROM customer
10     WHERE store_num =
11           cust_arr[idx].store_num
12   WHENEVER ERROR STOP
13  IF (SQLCA.SQLCODE = NOTFOUND) THEN
14    LET cont_ok = TRUE
15  ELSE
16     LET cont_ok = FALSE
17    IF (SQLCA.SQLCODE = 0) THEN
18      MESSAGE "Store Number already exists."
19     ELSE
20       ERROR SQLERRMESSAGE
21     END IF
22  END IF
23
```

```
24   RETURN cont_ok
25
26 END FUNCTION
```

Note:

- Line 02 The index of the current record in the program array is stored in the variable idx, passed to this function from the INPUT ARRAY control block ON CHANGE store_num.
- Line 03 The variable checknum is defined to hold the store_num returned by the SELECT statement.
- Line 06 sets the variable cont_ok to an initial value of FALSE. This variable is used to indicate whether the store number is unique.
- Lines 07 thru 12 use an embedded SQL SELECT statement to check whether the store_num already exists in the customer table. The index passed to this function is used to obtain the value that was entered into the store_num field on the form. The entire database row is not retrieved by the SELECT statement since the only information required by this program is whether the store number already exists in the table. The SELECT is surrounded by WHENEVER ERROR statements.
- Lines 13 thru 22 test SQLCA.SQLCODE to determine the success of the SELECT statement. The variable cont_ok is set to indicate whether the store number entered by the user is unique.
- Line 24 returns the value of cont_ok to the calling function.

Function insert_cust

This function inserts a new row into the customer database table.

Function insert_cust (custall.4gl):

```
01 FUNCTION insert_cust(idx)
02   DEFINE idx SMALLINT
03
04   WHENEVER ERROR CONTINUE
05  INSERT INTO customer
06       (store_num,
07        store_name,
08        addr,
09        addr2,
10        city,
11        state,
12        zip-code,
13        contact_name,
14        phone)
15     VALUES (cust_arr[idx].* )
16   WHENEVER ERROR STOP
17
18  IF (SQLCA.SQLCODE = 0) THEN
19     MESSAGE "Store added"
20  ELSE
21    ERROR SQLERRMESSAGE
22   END IF
23
24 END FUNCTION
```

Note:

- Line 02 This function is called from the AFTER INSERT control block of the INPUT ARRAY statement. The index of the record that was inserted into the cust_arr program array is passed to the function and stored in the variable idx.
- Lines 04 thru 16 uses an embedded SQL INSERT statement to insert a row into the customer database table. The values to be inserted into the customer table are obtained from the

record just inserted into the program array. The INSERT is surrounded by WHENEVER ERROR statements.

- Lines 18 thru 22 test the SQLCA.SQLCODE to see if the insert into the database was successful, and return an appropriate message to the user.

Function update_cust

This function updates a row in the customer database table. The functionality is very simple for illustration purposes, but it could be enhanced with additional error checking routines similar to the example in chapter 6.

Function update_cust (custall.4gl):

```
01 FUNCTION update_cust(idx)
02   DEFINE idx SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   UPDATE customer
06   SET
07     store_name   = cust_arr[idx].store_name,
08     addr         = cust_arr[idx].addr,
09     addr2        = cust_arr[idx].addr2,
10     city         = cust_arr[idx].city,
11     state        = cust_arr[idx].state,
12     zip-code      = cust_arr[idx].zip-code,
13     contact_name = cust_arr[idx].contact_name,
14     phone        = cust_arr[idx].phone
15    WHERE store_num = cust_arr[idx].store_num
16   WHENEVER ERROR STOP
17
18  IF (SQLCA.SQLCODE = 0) THEN
19      MESSAGE "Dealer updated."
20  ELSE
21     ERROR SQLERRMESSAGE
22   END IF
23
24 END FUNCTION
```

Note:

- Line 02 The index of the current record in the cust_arr program array is passed as idx from the ON ROW CHANGE control block.
- Lines 04 thru 16 use an embedded SQL UPDATE statement to update a row in the customer database table. The index of the current record in the program array is used to obtain the value of store_num that is to be matched in the customer table. The customer row is updated with the values stored in the current record of the program array. The UPDATE is surrounded by WHENEVER ERROR statements.
- Lines 18 thru 22 test the SQLCA.SQLCODE to see if the update of the row in the database was successful, and return an appropriate message to the user.

Function delete_cust

This function deletes a row from the customer database table. A modal Menu similar to that illustrated in Chapter 6 is used to verify that the user wants to delete the row.

Function delete_cust (custall.4gl):

```
01 FUNCTION delete_cust(idx)
02   DEFINE idx     SMALLINT,
03          del_ok      SMALLINT
04
05   LET del_ok = FALSE
```

```
06
07   MENU "Delete" ATTRIBUTES (STYLE="dialog",
08               COMMENT="Delete this row?")
09    COMMAND "OK"
10      LET del_ok = TRUE
11      EXIT MENU
12    COMMAND "Cancel"
13     LET del_ok = FALSE
14     EXIT MENU
15  END MENU
16
17  IF del_ok = TRUE THEN
18     WHENEVER ERROR CONTINUE
20     DELETE FROM customer
21       WHERE store_num = cust_arr[idx].store_num
22     WHENEVER ERROR STOP
23
24     IF (SQLCA.SQLCODE = 0) THEN
25      LET del_ok = TRUE
26      MESSAGE "Dealer deleted."
27     ELSE
28      LET del_ok = FALSE
29      ERROR SQLERRMESSAGE
30     END IF
31   END IF
32
33   RETURN del_ok
34
35 END FUNCTION
```

Note:

- Line `02` The index of the current record in the `cust_arr` program array is passed from the `BEFORE DELETE` control block of `INPUT ARRAY`, and stored in the variable `idx`. The `BEFORE DELETE` control block is executed immediately before the record is deleted from the program array, allowing the logic in this function to be executed before the record is removed from the program array.
- Line `05` sets the initial value of `del_ok` to `FALSE`.
- Lines `07` thru `15`display the modal Menu to the user for confirmation of the Delete.
- Lines `18` thru `22`use an embedded SQL `DELETE` statement to delete the row from the `customer` database table. The variable `idx` is used to determine the value of `store_num` in the program array record that is to be used as criteria in the `DELETE` statement. This record in the program array has not yet been removed, since this `delete_cust` function was called in a `BEFORE DELETE` control block. The `DELETE` is surrounded by `WHENEVER ERROR` statements.
- Lines `24` thru `30` test the `SQLCA.SQLCODE` to see if the update of the row in the database was successful, and return an appropriate message to the user. The value `del_ok` is set based on the success of the SQL `DELETE` statement.
- Line `33` returns the variable `del_ok` to the `BEFORE DELETE` control block, indicating whether the Delete of the `customer` row was successful.

Tutorial Chapter 9: Reports

This program generates a simple report of the data in the `customer` database table. The two parts of a report, the report driver logic and the report definition are illustrated. A technique to allow a user to interrupt a long-running report is shown.

This program generates a simple report of the data in the `customer` database table. The two parts of a report, the report driver logic and the `REPORT` program block (report definition) are illustrated. Then the program is modified to display a window containing a ProgressBar, and allowing the user to interrupt the report before it is finished.

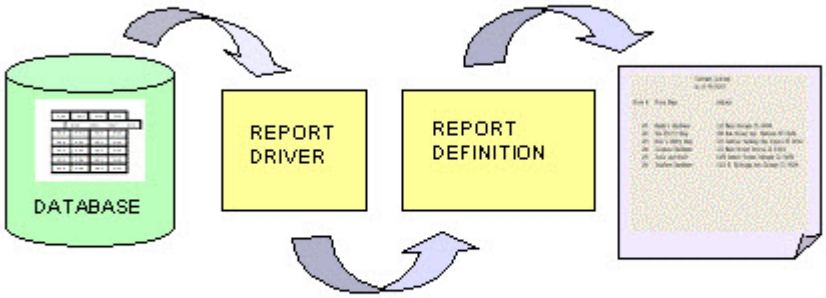

Figure 16: Report flow

BDL Reports

Genero BDL reports are easy to design and generate. The output from a report can be formatted so that the eye of the reader can easily pick out the important data.

The program logic that specifies what data to report (the report driver) is separate from the program logic that formats the output of the report (the report definition). This allows the report driver to supply data for multiple reports simultaneously, if desired. And, you can design template report definitions that might be used with report drivers that access different database tables.

The Report Driver

The part of a program that generates the rows of report data (also known as input records) is called the report driver. The primary concern of the row-producing logic is the selection of rows of data.

The actions of a report driver are:

1. Use the `START REPORT` statement to initialize each report to be produced. We recommend that clauses regarding page setup and report destination be included in this statement.
2. Use a forward-only database cursor to read rows from a database, if that is the source of the report data.
3. Whenever a row of report data is available, use `OUTPUT TO REPORT` to send it to the report definition.
4. If an error is detected, use `TERMINATE REPORT` to stop the report process.
5. When the last row has been sent, use `FINISH REPORT` to end the report.

From the standpoint of the row-producing side, these are the only statements required to create a report.

The Report Definition

The report definition uses a `REPORT` program block to format the input records.

`REPORT` is global in scope. It is not, however, a function; it is not reentrant, and `CALL` cannot invoke it.

The code within a `REPORT` program block consists of several sections, which must appear in the order shown.

The DEFINE section

Here you define the variables passed as parameter to the report, and the local variables. A report can have its own local variables for subtotals, calculated results, and other uses.

The OUTPUT section (optional)

Although you can define page setup and destination information in this section, the format of the report will be static. Providing this same information in the `START REPORT` statement provides more flexibility.

The ORDER BY section (optional)

Here you specify the required order for the data rows, when using grouping.

Include this `ORDER BY` section if 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. To avoid the creation of additional resources to sort the data, use the `ORDER EXTERNAL` statement in this section if the data to be used in the report has already been sorted by an `ORDER BY` clause in the SQL statement.

The FORMAT section

Here you describe what is to be done at a particular stage of report generation. The code blocks you write in the `FORMAT` section are the heart of the report program block and contain all its intelligence. You can use most BDL statements in the `FORMAT` section of a report; you cannot, however, include any SQL statements.

BDL invokes the sections and blocks within a report program block nonprocedurally, at the proper time, as determined by the report data. You do not have to write code to calculate when a new page should start, nor do you have to write comparisons to detect when a group of rows has started or ended. All you have to write are the statements that are appropriate to the situation, and BDL supplies the "glue" to make them work.

You can write control blocks in the `FORMAT` section to be executed for the following events:

- Top (header) of the first page of the report (`FIRST PAGE HEADER`)
- Top (header) of every page after the first (`PAGE HEADER`)
- Bottom (footer) of every page (`PAGE TRAILER`)
- Each new row as it arrives (`ON EVERY ROW`)
- The start end of a group of rows (`BEFORE GROUP OF`) - a group is one or more rows having equal values in a particular column.
- The end of a group of rows (`AFTER GROUP OF`) - in this block, you typically print subtotals and other aggregate data for the group that is ending. You can use aggregate functions to calculate and display frequencies, percentages, sums, averages, minimum, and maximum for this information.
- After the last row has been processed (`ON LAST ROW`) - aggregate functions calculated over all the rows of the report are typically printed here.

Two-pass reports

A two-pass report is one that creates temporary tables, therefore there must be an active connection to the database.

The two-pass report handles sorts internally. 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.

If your report definition includes any of the following, a two-pass report is required:

- An `ORDER BY` section without the `EXTERNAL` keyword.
- The `GROUP PERCENT(*)` aggregate function anywhere in the report.
- Any aggregate function outside the `AFTER GROUP OF` control block.

> **Note:** Some databases do not support temporary tables. Avoid a two-pass report for performance reasons and for portability.

Example: Customer Report

This example demonstrates a simple report driver and definition. The report driver extracts rows from the `customer` database table and passes them to the report definition to be formatted.

The Report Driver

The Report Driver for this example, `custreports.4gl` defines a cursor to retrieve `customer` table rows sorted by `state`, then `city`. The `START REPORT` statement initializes the report and provides destination and page setup information to the Report Definition.

Report Driver `custreports.4gl`:

```
01 SCHEMA custdemo
02
03 MAIN
04 DEFINE pr_custrec RECORD
05   store_num  LIKE customer.store_num,
06   store_name LIKE customer.store_name,
07   addr       LIKE customer.addr,
08   addr2      LIKE customer.addr2,
09   city       LIKE customer.city,
10   state      LIKE customer.state,
11   zip-code    LIKE customer.zip-code
12  END RECORD
13
14  CONNECT TO "custdemo"
15
```

```
16  DECLARE custlist CURSOR FOR
17     SELECT store_num,
18            store_name,
19            addr,
20            addr2,
21            city,
22            state,
23            zip-code
24       FROM customer
25      ORDER BY state, city
26
27  START REPORT cust_list TO FILE "customers.txt"
28    WITH LEFT MARGIN = 5, TOP MARGIN = 2,
29          BOTTOM MARGIN = 2
30
31  FOREACH custlist INTO pr_custrec.*
32   OUTPUT TO REPORT cust_list(pr_custrec.*)
33  END FOREACH
34
35  FINISH REPORTcust_list
36
37  DISCONNECT CURRENT
38
39 END MAIN
```

Note:

- Lines 04 thru 12 define a local program record `pr_custrec`, with a structure like the `customer` database table.
- Line14 connects to the `custdemo` database.
- Lines 16 thru 25 define a `custlist` cursor to retrieve the `customer` table data rows, sorted by `state`, then `city`.
- Lines 27 thru29 starts the `REPORT` program block named `cust_list`, and includes a report destination and page formatting information.
- Lines 31 thru 33 retrieve the data rows one by one into the program record `pr_custrec` and pass the record to the `REPORT` program block.
- Line 35 closes the report driver and executes any final `REPORT` control blocks to finish the report.
- Line37 disconnects from the `custdemo` database.

The Report Definition

The Report Definition uses the `REPORT` program block to format the input records from the Report Driver.

Report definition `custreport.4gl`:

```
01 REPORT cust_list(r_custrec)
02  DEFINE r_custrec RECORD
03        store_num  LIKE customer.store_num,
04        store_name LIKE customer.store_name,
05        addr       LIKE customer.addr,
06        addr2      LIKE customer.addr2,
07        city       LIKE customer.city,
08        state      LIKE customer.state,
09        zip-code    LIKE customer.zip-code
10       END RECORD
11
12  ORDER EXTERNAL BY r_custrec.state, r_custrec.city
13
14  FORMAT
15
16   PAGE HEADER
```

```
17    SKIP 2 LINES
18     PRINT COLUMN 30, "Customer Listing"
19     PRINT COLUMN 30, "As of ", TODAY USING "mm/dd/yy"
20    SKIP 2 LINES
21
22    PRINT  COLUMN 2, "Store #",
23          COLUMN 12, "Store Name",
24          COLUMN 40, "Address"
25
26    SKIP 2 LINES
27
28   ON EVERY ROW
29     PRINT COLUMN 5, r_custrec.store_num USING "####",
30          COLUMN 12, r_custrec.store_name CLIPPED,
31          COLUMN 40, r_custrec.addr CLIPPED;
32
33    IF r_custrec.addr2 IS NOT NULL THEN
34      PRINT 1SPACE, r_custrec.addr2 CLIPPED, 1 space;
35    ELSE
36        PRINT 1 SPACE;
37    END IF
38
39    PRINT r_custrec.city CLIPPED, 1 SPACE,
40          r_custrec.state, 1 SPACE,
41          r_custrec.zip-code CLIPPED
42
43   BEFORE GROUP OF r_custrec.city
44    SKIP TO TOP OF PAGE
45
46   ON LAST ROW
47    SKIP 1 LINE
48    PRINT "TOTAL number of customers: ",
49           COUNT(*) USING "#,###"
50
51   PAGE TRAILER
52    SKIP 2 LINES
53    PRINT COLUMN 30, "-", PAGENO USING "<<", " -"
54
55 END REPORT
```

Note:

- Line `01` The `REPORT` control block has the `pr_custrec` record passed as an argument.
- Lines `02` thru `10` define a local program record `r_custrec` to store the values that the calling routine passes to the report.
- Line `12` tells the `REPORT` control block that the records will be passed sorted in order by `state`, then `city`. The `ORDER EXTERNAL` syntax is used to prevent a second sorting of the program records, since they have already been sorted by the SQL statement in the report driver.
- Line `14` is the beginning of the `FORMAT` section.
- Lines`16` thru `20` The `PAGE HEADER` block specifies the layout generated at the top of each page. Each `PRINT` statement starts a new line containing text or a value. The `PRINT` statement can have multiple `COLUMN` clauses, which all print on the same line. The `COLUMN` clause specifies the offset of the first character from the first position after the left margin. The values to be printed can be program variables, static text, or built-in functions. The built-in `TODAY` operator generates the current date; the `USING` clauses formats this. The `SKIP` statement inserts empty lines. The `PAGE HEADER` for this report will appear as follows:

```
    <skipped line>
    <skipped line>
              Customer Listing              As of <date>
    <skipped line>
```

```
    <skipped line>
    Store #    Store Name          Address     <skipped line>
    <skipped line>
```

- Lines 28 thru 41 specifies the layout generated for each row. The data can be read more easily if each program record passed to the report is printed on a single row. Although there are four `PRINT` statements in this control block, the first three `PRINT` statements are terminated by semicolons. This suppresses the new line signal, resulting in just a single row of printing. The `CLIPPED` keyword eliminates any trailing blanks after the name, addresses, and city values. Any `IF` statement that is included in the `FORMAT` section must contain the same number of `PRINT` / `SKIP` statements regardless of which condition is met. Therefore, if `r_custrec.addr2` is not `NULL`, a `PRINT` statement prints the value followed by a single space; if it is `NULL`, a `PRINT` statement prints a single space. As mentioned earlier, each `PRINT` statement is followed by a semicolon to suppress the newline. The output for each row will be as follows:

```
    106 TrueTest Hardware     6123 N. Michigan Ave Chicago IL 60104
101 Bandy's Hardware      110 Main Chicago IL 60068
```

- Lines 43 and 44 start a new page for each group containing the same value for `r_custrec.city`.
- Lines 46 thru 49 specify a control block to be executed after the statements in `ON EVERY ROW` and `AFTER GROUP OF` control block. This prints at the end of the report. The aggregate function `COUNT(*)` is used to print the total number of records passed to the report. The `USING` keyword formats the number. This appears as follows:

```
    <skipped line>
    Total number of customers:   <count>
```

- Lines 51 thru 53 specifies the layout generated at the bottom of each page. The built-in function `PAGENO` is used to print the page number. The `USING` keyword formats the number, left-justified. This appears as follows:

```
    <skipped line>
    <skipped line>
                  - <pageno> -
```

Interrupting a Report

When a program performs a long process like a loop, a report, or a database query, the lack of user interaction statements within the process can prevent the user from interrupting it. In this program, the preceding example is modified to display a form containing start, exit, and interrupt buttons, as well as a progress bar showing how close the report is to completion.

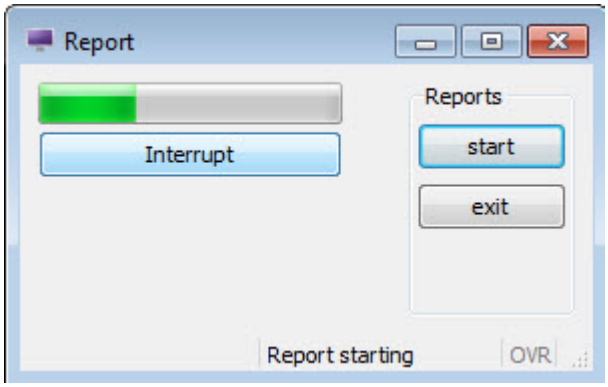

Figure 17: Interrupting a report

The interrupt action view

In order to allow a user to stop a long-running report, for example, you can define an action view with the name "interrupt". When the runtime system takes control of the program, the client automatically enables a local interrupt action to let the user send an asynchronous request to the program.

This interruption request is interpreted by the runtime system as a traditional interruption signal, as if it was generated on the server side, and the `INT_FLAG` variable is set to `TRUE`.

Refreshing the Display

The Abstract User Interface 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. 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.

The `Interface` class is a built-in class provided to manipulate the user interface. The `refresh()` method of this class synchronizes the front end with the current AUI tree. You do not need to instantiate this class before calling any of its methods:

```
CALL ui.Interface.refresh()
```

Using a ProgressBar

One of the form item types is a `PROGRESSBAR`, a horizontal line with a progress indicator. The position of the `PROGRESSBAR` is defined by the value of the corresponding form field. The value can be changed from within a BDL program by using the `DISPLAY` instruction to set the value of the field.

This type of form item does not allow data entry; it is only used to display integer values. The `VALUEMIN` and `VALUEMAX` attributes of the `PROGRESSBAR` define the lower and upper integer limit of the progress information. Any value outside this range will not be displayed.

Example: Interruption Handling

The progressbar application in chapter 9 show the changes needed to facilitate interruption handling. A form specification file, `reportprog.per` contains form fields for a `PROGRESSBAR` and `interrupt` action view. The Report Driver, `custreports.4gl`, has been modified to handle interrupts.

The Form Specification File

A form containing a progress bar is defined in the form specification file `reportprog.per`.

Form `reportprog.per`:

```
01 LAYOUT (TEXT="Report")
02  GRID
03  {
04
05        [f001                  ]
06
07        [ib                    ]
08
09
10  }
11  END
12 END
13
14 ATTRIBUTES
15 PROGRESSBAR f001 = formonly.rptbar, VALUEMIN=1,VALUEMAX=10;
16 BUTTON ib: interrupt, TEXT="Stop";
```

```
17 END
```

Note:

- Line 05 contains the form field for the PROGRESSBAR.
- Line 07 contains the form field for the `interrupt` action view.
- Line 15 defines the PROGRESSBAR as FORMONLY since its type is not derived from a database column. The values range from 1 to 10. The maximum value for the PROGRESSBAR was chosen arbitrarily, and was set rather low since there are not many rows in the customer database table.
- Line 16 defines the button `ib` as an interrupt action view with TEXT of "Stop".

Modifications to custreports.4gl

The MAIN program block has been modified to open a window containing the form with a PROGRESSBAR and a MENU, to allow the user to start the report and to exit. A new function, `cust_report`, is added for interruption handling. The report definition, the `cust_list REPORT` block, remains the same as in the previous example.

Changes to the MAIN program block (`custreport2.4gl`):

```
01 MAIN
02
03   DEFER INTERRUPT
04   CONNECT TO "custdemo"
05   CLOSE WINDOW SCREEN
06   OPEN WINDOW w3 WITH FORM "reportprog"
07
08   MENU "Reports"
09   ON ACTION start
10      MESSAGE "Report starting"
11       CALL cust_report()
12   ON ACTION exit
13       EXIT MENU
14   END MENU
15
16   CLOSE WINDOW w3
17   DISCONNECT CURRENT
18
19 END MAIN
```

Note:

- Line 03 prevents the user from interrupting the program except by using the `interrupt` action view.
- Line 06 Opens the window and form containing the PROGRESSBAR.
- Lines 08 thru 14 define a MENU with two actions:
  - **start**- displays a MESSAGE and calls the function `cust_report`
  - **exit** - quits the MENU

The cust_report function

This new function contains the report driver, together with the logic to determine whether the user has attempted to interrupt the report.

Function `cust_report` (`custreport2.4gl`):

```
21 FUNCTION cust_report()
22
23 DEFINE pr_custrec RECORD
24       store_num   LIKE customer.store_num,
25       store_name  LIKE customer.store_name,
```

```
26       addr        LIKE customer.addr,
27       addr2       LIKE customer.addr2,
28       city        LIKE customer.city,
29       state       LIKE customer.state,
30       zip-code     LIKE customer.zip-code
31       END RECORD,
32       rec_count, rec_total,
33       pbar, break_num INTEGER
34
35    LET rec_count = 0
36    LET rec_total = 0
37    LET pbar = 0
38    LET break_num = 0
39    LET INT_FLAG = FALSE
40
41    SELECT COUNT(*) INTO rec_total FROM customer
42
43   LET break_num = (rec_total/10)
44
45    DECLARE custlist CURSOR FOR
46      SELECT store_num,
47         store_name,
48         addr,
49         addr2,
50         city,
51         state,
52         zip-code 53     FROM CUSTOMER
54       ORDER BY state, city
55
56    START REPORT cust_list TO FILE "customers.txt"
57    FOREACH custlist INTO pr_custrec.*
58      OUTPUT TO REPORT cust_list(lr_custrec.*)
59      LET rec_count = rec_count+1
60     IF (rec_count MOD break_num)= 0 THEN
61        LET pbar = pbar+1
62        DISPLAY pbar TO rptbar
63        CALL ui.Interface.refresh()
64       IF (INT_FLAG) THEN
65         EXIT FOREACH
66        END IF
67      END IF
68    END FOREACH
69
70    IF (INT_FLAG) THEN
71     LET INT_FLAG = FALSE
72     MESSAGE "Report cancelled"
73   ELSE
74      FINISH REPORT cust_list
75      MESSAGE "Report finished"
76   END IF
77
78 END FUNCTION
```

Note:

- Lines 23 thru 31 now define the `pr_custrec` record in this function.
- Lines 32 thru 33 define some additional variables.
- Lines 35 thru 39 initialize the local variables.
- Line 38 sets `INT_FLAG` to `FALSE`.
- Line 41 uses an embedded SQL statement to retrieve the count of the rows in the `customer` table and stores it in the variable `rec_total`.

- Line 43 calculates the value of `break_num` based on the maximum value of the PROGRESSBAR, which is set at 10. After `break_num` rows have been processed, the program will increment the PROGRESSBAR. The front end cannot handle interruption requests properly if the display generates a lot of network traffic, so we do not recommend refreshing the AUI and checking `INT_FLAG` after every row.
- Lines 45 thru 54 declare the `custlist` cursor for the `customer` table.
- Line 56 starts the report, sending the output to the file `custout`.
- Lines 58 thru 68 contain the FOREACH statement to output each record to the same report `cust_list` used in the previous example.
- Line 59 increments `rec_count` to keep track of how many records have been output to the report.
- Line 60 tests whether a break point has been reached, using the MOD (Modulus) function.
- Line 61 If a break point has been reached, the value of `pbar` is incremented.
- Line 62 The `pbar` value is displayed to the `rptbar` PROGRESSBAR form field.
- Line 63 The front end is synced with the current AUI tree.
- Line 64 thru 66 The value of `INT_FLAG` is checked to see whether the user has interrupted the program. If so, the FOREACH loop is exited prematurely.
- Lines 70 thru 76 test `INT_FLAG` again and display a message indicating whether the report finished or was interrupted. If the user did not interrupt the report, the `FINISH REPORT` statement is executed.

Tutorial Chapter 10: Localization

Localization support and localized strings allow you to internationalize your application using different languages, and to customize it for specific industry markets in your user population. This chapter illustrates the use of localized strings in your programs.

Localization Support

Localization Support is a feature of the language that allows you to write application supporting multibyte character sets as well as date, numeric and currency formatting in accordance with a `locale`.

Localization Support is based on the system libraries handling the `locale`, a set of language and cultural rules.

See *Localization* in the *Genero Business Development Language User Guide* for more details.

Localized Strings

Localized Strings allow you to internationalize your application using different languages, and to customize it for specific industry markets in your user population. Any string that is used in your Genero BDL program, such as messages to be displayed or the text on a form, can be defined as a Localized String. At runtime, the Localized String is replaced with text stored in a String File.

String Files must be compiled, and then deployed at the user site.

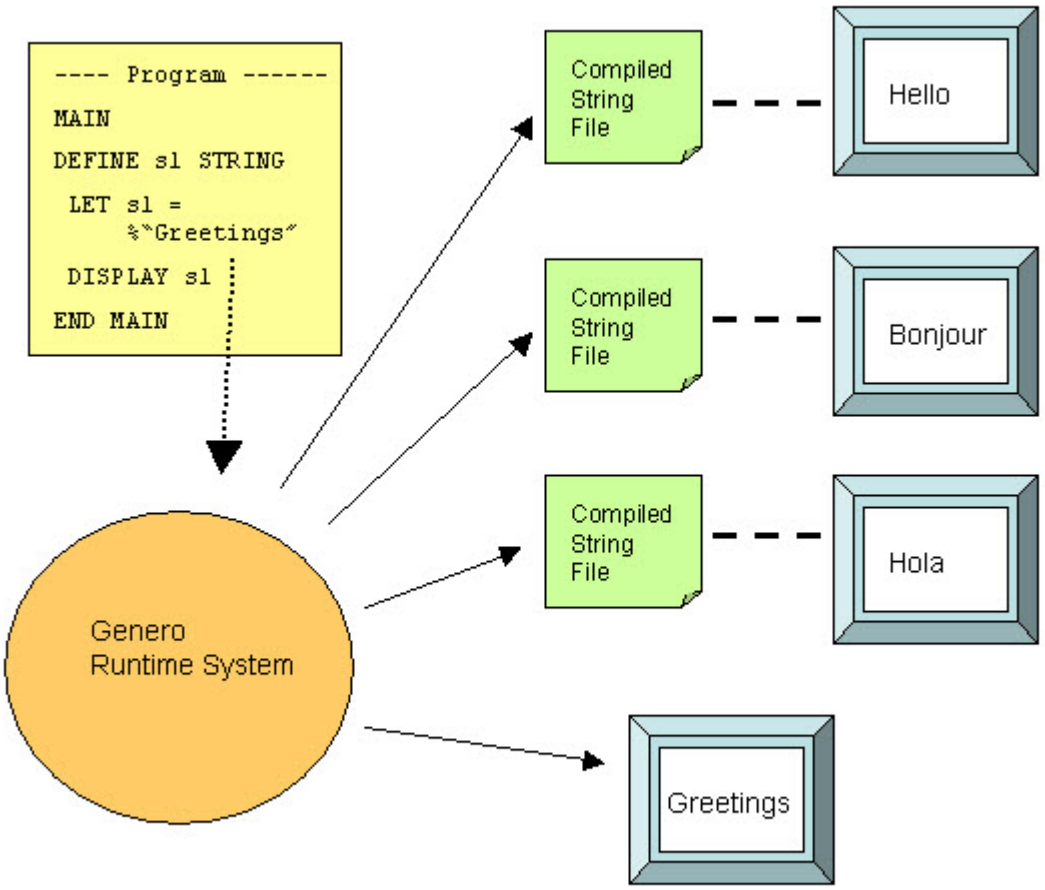

Figure 18: Localized strings

Programming Steps

These steps describe how to use Localized Strings in your sources.

1. Modify your form specification files and program module files to contain Localized Strings by inserting the % sign in front of the strings that you wish to be replaced.
2. Use the `-m` option of `fglform` to extract the Localized Strings from each form specification file into a separate Source String File(extension `.str`).
3. Use the `-m` option of `fglcomp` to extract the Localized Strings from each program module into a separate Source String File (extension `.str`).
4. Concatenate the Source String Files together logically; for example, you may have a `common.str` file containing the strings common to all applications, a `utility.str` file containing the strings common to utilities, and an `application.str` file with the strings specific to the particular application.

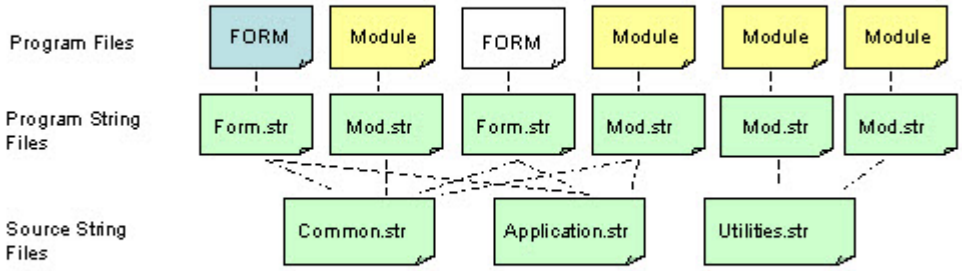

Figure 19: Example concatenation of source string files

5. At this point the names of the Localized Strings may be unwieldy, since they were derived from the actual strings in the program files. You can modify the string names in your Source String Files and the corresponding program files so they form keys that are logical. For example: `$"common.accept" = "OK" $"common.cancel"= "Cancel" $"common.quit" = "Quit"`
6. Make the Source String Files available to the programming teams for use as a reference when creating or modifying programs.
7. Copy the Source String Files, and modify the replacement text for each of your market segments or user languages.
8. Compile the Source String Files (`.42s`).
9. Create the entries in the `fglprofile` file to specify what string files must be used at runtime.
10. Deploy `.42s` compiled string files to user sites.

Strings in Sources

A Localized String begins with a percent sign (%), followed by the name of the string identifying the replacement text to be loaded from the compiled String File. Since the name is a `STRING`, you can use any characters in the name, including blanks.

```
LET s1 = %"Greetings"
```

The `STRING` "Greetings" is both the name of the string and the default text which would be used if no string resource files are provided at runtime.

Localized Strings can be used any place where a `STRING` literal can be used, including form specification files.

The `SFMT()` and `LSTR()` operators can be used to manipulate the contents of Localized Strings. For example, the program line:

```
DISPLAY SFMT( %"cust.valid", custnum )
```

reads from the associated Compiled String File:

```
"cust.valid"="customer %1 is valid"
```

resulting in the following display when the value of `custnum` is 200:

```
"customer 200 is valid"
```

Extracting Strings

You can generate a Source String File by extracting all of the Localized Strings from your program module or form specification file, using the `-m` option of `fglcomp` or `fglform`:

```
fglcomp -m mystring.4gl > mystring.str
```

The generated file would have the format:

```
"Greetings" = "Greetings"
```

You could then change the replacement text in the file:

```
"Greetings" = "Hello"
```

The source string file must have the extension `.str`.

Compiling String Source Files (fglmkstr)

String Source Files must be compiled to binary files in order to be used at runtime.

You can compile the String files using the **Compile File** or application-level **Build** option in Genero Studio, or use the command line tool `fglmkstr`.

```
fglmkstr mystring.str
```

The resulting Compiled String File has the extension `.42s` (`mystring.42s`).

Deploying String Files

The Compiled String Files must be deployed on the production sites. The file extension is `.42s`. By default, the runtime system searches for a `.42s` file with the same name prefix as the current `.42r` program. You can specify a list of string files with entries in the `fglprofile` configuration file.

```
fglrun.localization.file.count = 2
fglrun.localization.file.1.name = "firstfile"
fglrun.localization.file.2.name = "secondfile"
```

The current directory and the path defined in the `DBPATH`/ `FGLRESOURCEPATH` environment variable, are searched for the `.42s` Compiled String File.

> **Tip:** Create several string files with the same names, but locate them in different directories. You can then easily switch from one set of string files to another, just by changing the `DBPATH` / `FGLRESOURCEPATH` environment variable. You typically create one string file directory per language, and if needed, you can create subdirectories for each codeset (strings/english/iso8859-1, strings/french/windows1252).

Example: Localization

The `progstrings` program demonstrates localized strings in a form and program module.

form.per - the form specification file

The form specification file uses the `LABEL` form item type to display the text associated with the form fields containing data from the `customer` database table. `LABEL` item types contain read-only values. The `TEXT` of the `LABEL` form items contain Localized Strings. The `COMMENT` attribute of an `EDIT` item is also a Localized String.

Form `form.per`:

```
01 SCHEMA custdemo
02
03 LAYOUT
04  GRID
05   {
06    [lab1      ] [f01  ]
07
08    [lab2      ] [f02                ]
09
10    [lab3      ] [f03            ]
11   }
12   END   --grid
13 END  -- layout
14
15 TABLES customer
16
17 ATTRIBUTES
18 LABEL lab1: TEXT=%"customer.store_num";
19 EDIT  f01  = customer.store_num,
20             COMMENT=%"customer.dealermsg";
21 LABEL lab2: TEXT=%"customer.store_name";
22 EDIT  f02  = customer.store_name;
23 LABEL lab3: TEXT=%"customer.city";
24 EDIT  f03  = customer.city;
25 END  -- attributes
```

Note:

- Lines `06` and `18`: The form contains a `LABEL`, lab1; the `TEXT` of the `LABEL` is a Localized String, `customer.store_num`.
- Line `20`: The `COMMENT` of the `EDIT` f01 is a Localized String, `customer.dealermsg`.
- Lines `08` and `21`: The `TEXT` of the `LABEL` lab2 is a Localized String, `customer.store_name`.
- Lines `10` and `23`: The `TEXT` of the `LABEL` lab3 is a Localized String, `customer.city`.

These strings will be replaced at runtime.

The string file entries associated with this form

You can view the translations for the Localized Strings in the form in the `progstrings.str` string source file.

```
01 "customer.store_num"="Store No"
02 "customer.dealernummsg"="This is the dealer number"
03 "customer.store_name"="Store Name"
04 "customer.city"="City"
```

prog.4gl - the program module

The program module opens the form containing Localized Strings. The program module also contains Localized Strings for messages to be displayed.

Module `prog.4gl`:

```
01 SCHEMA custdemo
02
03 MAIN
04  CONNECT TO "custdemo"
05  CLOSE WINDOW SCREEN
06  OPEN WINDOW w1 WITH FORM "stringform"
07  MESSAGE %"customer.msg"
08  MENU %"customer.menu"
09     ON ACTION query
10       CALL query_cust()
11     ON ACTION exit
12       EXIT MENU
13  END MENU
14  CLOSE WINDOW w1
15  DISCONNECT CURRENT
16 END MAIN
17
18 FUNCTION query_cust() -- displays one row
19   DEFINE l_custrec RECORD
20          store_num LIKE customer.store_num,
21          store_name LIKE customer.store_name,
22          city LIKE customer.city
23   END RECORD,
24   msg STRING
25
26   WHENEVER ERROR CONTINUE
27   SELECT store_num, store_name, city
28    INTO l_custrec.*
29    FROM customer
30    WHERE store_num = 101
31   WHENEVER ERROR STOP
32
33   IF SQLCA.SQLCODE = 0 THEN
34      LET msg = SFMT( %"customer.valid",
35                      l_custrec.store_num )
36      MESSAGE msg
37      DISPAY BY NAME l_custrec.*
38   ELSE
39      MESSAGE %"customer.notfound"
40   END IF
41
42 END FUNCTION
```

Note:

- Lines `07`, `08`, `34` and `39` contain Localized Strings for the messages that the program displays.

These strings will be replaced at runtime.

The string file associated with this program module

You can view the translations for the Localized Strings in the program module in the `progstrings.str` string source file.

```
01 "customer.msg"="Program retrieves dealer #101"
02 "customer.menu"="Dealer"
03 "customer.valid"="Customer %1 is valid"
```

```
04 "customer.notfound"="Customer was not found"
```

Compiling the program

Compile the program and string file using Genero Studio or command line tools.

From Genero Studio

As you learned earlier in the Tutorial, the **Execute** option in the Genero Studio Project view will compile and link files in the specified application node if necessary before executing the application. This behavior also applies to String Source files (`.str`). String Source files can also be compiled independently with the **Compile File** option.

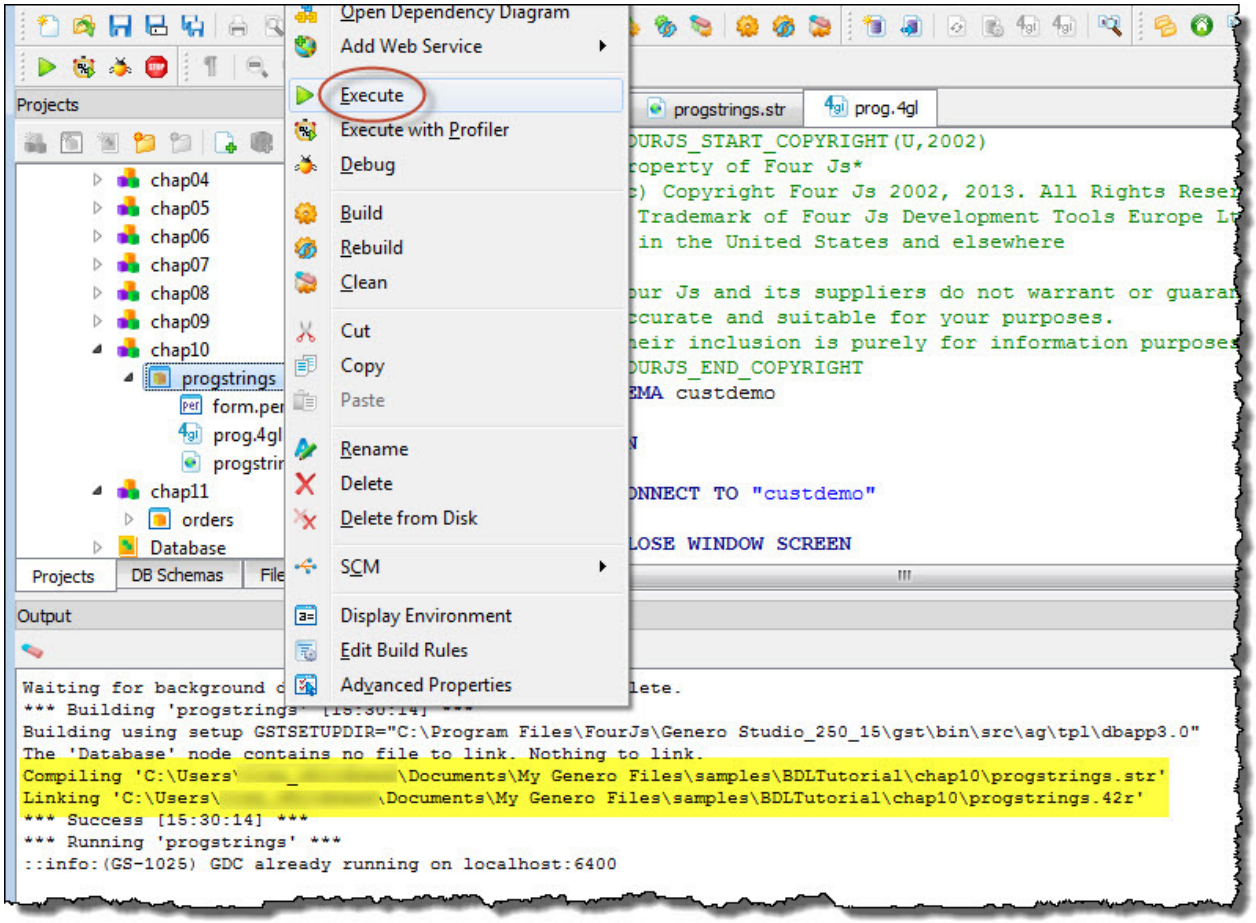

Figure 20: Using the Execute option to compile and execute the progstrings program

To Compile and execute from the command line

The program is compiled into `progstrings.42r`.

```
fgl2p -o progstrings.42r prog.4gl
```

The `progstring.str` string file must be compiled:

```
fglmkstr progstring.str
```

The resulting Compiled String File is `progstring.42s`.

Setting the list of compiled string files in the `fglprofile` file.

The list of Compiled String Files is specified in the `fglprofile` configuration file. The runtime system searches for a file with the "`42s`" extension in the current directory and in the path list defined in the `DBPATH` / `FGLRESOURCEPATH` environment variable. Specify the total number of files, and list each file with an index number.

Example `fglprofile` file

```
01 fglrun.localization.file.count = 2
02 fglrun.localization.file.1.name = "form"
03 fglrun.localization.file.2.name = "prog"
```

Setting the environment

Set the `FGLPROFILE` environment variable:

```
export FGLPROFILE=./fglprofile
```

Running the program

Run the program:

```
fglrun cust
```

The Resulting Form Display

Display of the form using the default values for the strings.

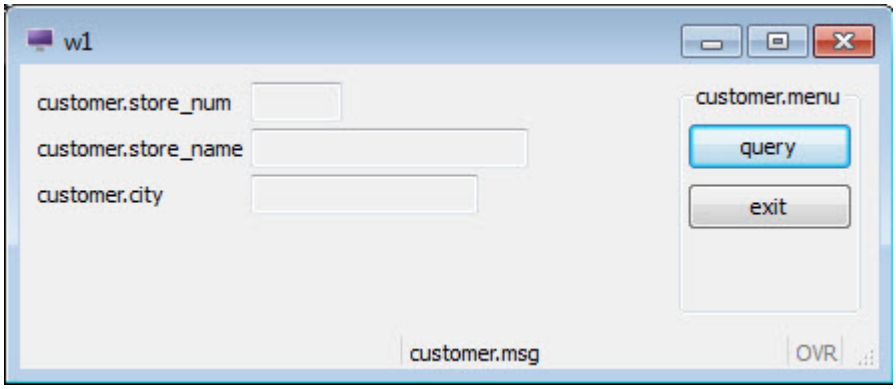

Figure 21: Form with default values for strings

Display of the form when the Compiled String File is deployed.

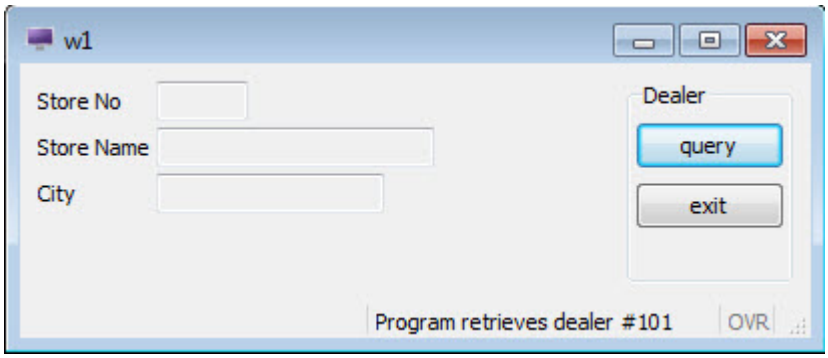

Figure 22: Form using compiled string file

Tutorial Chapter 11: Master/Detail

The form used by the program in this chapter contains fields from both the `orders` and `items` tables in the `custdemo` database, illustrating a master-detail relationship. Since there are multiple items associated with a single order, the rows from the `items` table are displayed in a table on the form. This chapter focuses on the master/detail form and the unique features of the corresponding program.

The Master-Detail sample

The example discussed in this chapter is designed for the input of order information (headers and order lines), illustrating a typical master-detail relationship. The form used by the example contains fields from both the `orders` and `items` tables in the `custdemo` database.

Since there are multiple items associated with a single order, the rows from the items table are stored in a program array and displayed in a table container on the form. Most of the functionality to query/add/update/delete has been covered in previous chapters; this chapter will focus on the master/detail form and the unique features of the corresponding program.This type of relationship can also be handled with multiple dialogs, as shown in Chapter 13.

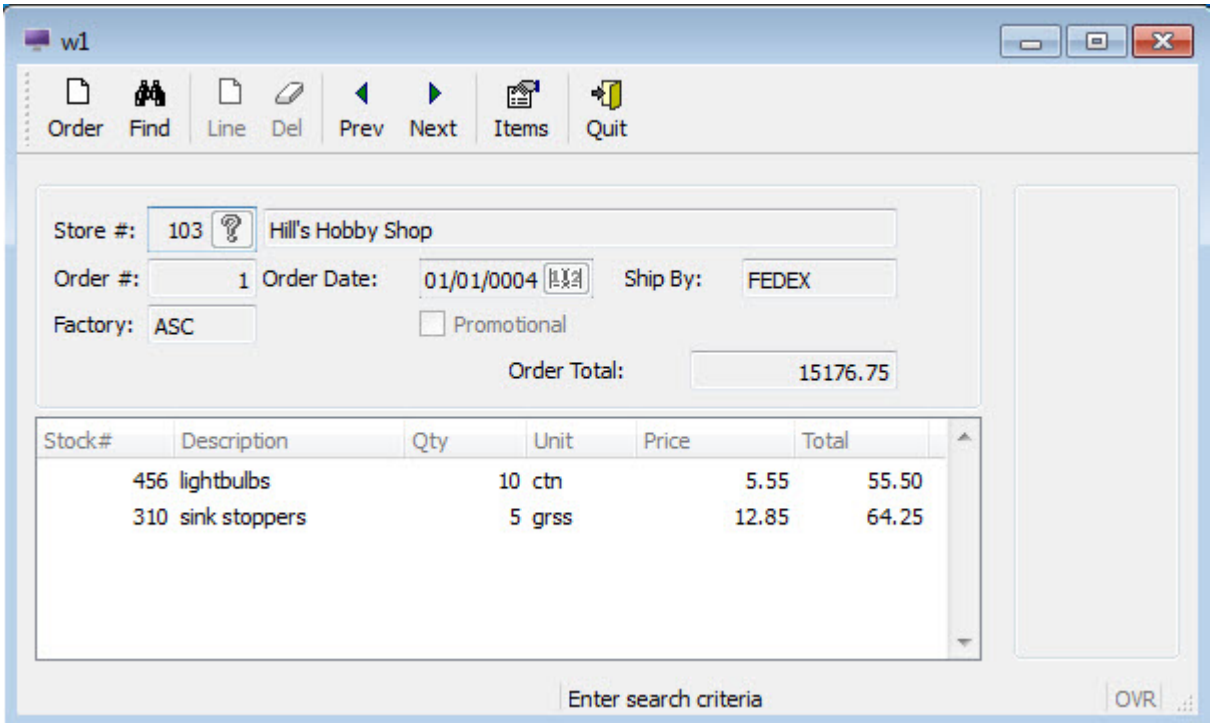

Figure 23: Master-Detail Form

The Makefile

The BDL modules and forms used by the application in this chapter can be compiled and linked in Genero Studio using the Application-level **Execute** or **Build** options. If you prefer command line tools you can compile and link using a `Makefile`. This file is interpreted by the `make` utility, which is a well-known tool to build large programs based on multiple sources and forms.

The `make` utility reads the dependency rules defined in the `Makefile` for each program component, and executes the commands associated with the rules.

This section only describes the `Makefile` used in this example. For more details about Makefiles, see *Using makefiles* in the *Genero Business Development Language User Guide*.

The `Makefile`:

```
01  all:: orders
02
03  orders.42m: orders.4gl
04          fglcomp -M orders.4gl
05
06  orderform.42f: orderform.per
07          fglform -M orderform.per
08
09  custlist.42m: custlist.4gl
10          fglcomp -M custlist.4gl
11
12  custlist.42f: custlist.per
13          fglform -M custlist.per
14
15  stocklist.42m: stocklist.4gl
16          fglcomp -M stocklist.4gl
17
18  stocklist.42f: stocklist.per
19          fglform -M stocklist.per
20
21  MODULES=\
22   orders.42m\
23   custlist.42m\
24   stocklist.42m
25
26  FORMS=\
27   orderform.42f\
28   custlist.42f\
29   stocklist.42f
30
31  orders:: $(MODULES) $(FORMS)
32        fgllink -o orders.42r $(MODULES)
33
34  run::
35        fglrun orders
36
37  clean::
38          rm -f *.42?
```

Note:

- Line `01` defines the `all` dependency rule that will be executed by default, and depends from the rule `orders` described on line `31`. You execute this rule with `make all` , or `make` since this is the first rule in the `Makefile`.

- Lines 03 and 04 define a dependency to compile the `orders.4gl` module into `orders.42m`. The file on the left (`orders.42m`) depends from the file on the right (`orders.4gl`), and the command to be executed is `fglcomp -M orders.4gl` .
- Lines 06 and 07 define a dependency to compile the `orderform.per` form.
- Lines 09 and 10 define a dependency to compile the `custlist.4gl` module.
- Lines 12 and 13 define a dependency to compile the `custlist.per` form.
- Lines 15 and 16 define a dependency to compile the `stocklist.4gl` module.
- Lines 18 and 19 define a dependency to compile the `stocklist.per` form.
- Lines 21 thru 24 define the list of compiled modules, used in the global `orders` dependency rule.
- Lines 26 thru 29 define the list of compiled form files, used in the global `orders` dependency rule.
- Lines 31 and 32 is the global '`orders`' dependency rule, defining modules or form files to be created.
- Lines 34 and 35 define a rule and command to execute the program. You execute this rule with `make run`.
- Lines 37 and 38 define a rule and command to clean the directory. You execute this rule with `make clean`.

The Customer List Module

The `custlist.4gl` module defines a 'zoom' module, to let the user select a customer from a list. The module could be reused for any application that requires the user to select a customer from a list.

This module uses the `custlist.per` form and is typical list handling using the `DISPLAY ARRAY` statement, as discussed in Chapter 07. The `display_custlist()` function in this module returns the customer id and the name. See the `custlist.4gl` source module for more details.

In the application illustrated in this chapter, the main module `orders.4gl` will call the `display_custlist()` function to retrieve a customer selected by the user.

```
01   ON ACTION zoom1
02     CALL display_custlist() RETURNING id, name
03     IF (id > 0) THEN
04         ...
```

The Stock List Module

The `stocklist.4gl` module defines a 'zoom' module, to let the user select a stock item from a list. This module uses the `stocklist.per` form and is typical list handling using the DISPLAY ARRAY statement, as discussed in Chapter 07.

See the `stocklist.4gl` source module for more details.

The main module `orders.4gl` will call the `display_stocklist()` function of the `stocklist.4gl` module to retrieve a stock item selected by the user.

The function returns the stock item id only:

```
01   ON ACTION zoom2
02     LET id = display_stocklist()
03     IF (id > 0) THEN
04         ...
```

The Master-Detail Form Specification File

The form specification file `orderform.per` defines a form for the `orders` program, and displays fields containing the values of a single order from the `orders` table. The name of the store is retrieved from the `customer` table, using the column `store_num`, and displayed. A screen array displays the associated rows from the `items` table.

Although `order_num` is also one of the fields in the `items` table, it does not have to be included in the screen array or in the screen record , since the order number will be the same for all the items displayed for a given order. For each item displayed in the screen array, the values in the `description` and `unit` columns from the `stock` table are also displayed.

The values in `FORMONLY` fields are not retrieved from a database; they are calculated by the BDL program based on the entries in other fields. In this form `FORMONLY` fields are used to display the calculations made by the BDL program for item line totals and the order total.

This form uses some of the attributes that can be assigned to fields in a form. See *Form item attributes* in the *Genero Business Development Language User Guide* for a complete list of the available attributes.

Form `orderform.per`:

```
01 SCHEMA custdemo
02
03 TOOLBAR
04  ITEM new (TEXT="Order", IMAGE="new", COMMENT="New order")
05  ITEM find (TEXT="Find", IMAGE="find")
06  SEPARATOR
07  ITEM append (TEXT="Line", IMAGE="new", COMMENT="New order line")
08  ITEM delete (TEXT="Del", IMAGE="eraser")
09  SEPARATOR
10  ITEM previous (TEXT="Prev")
11  ITEM next (TEXT="Next")
12  SEPARATOR
13  ITEM getitems (TEXT="Items", IMAGE="prop")
14  SEPARATOR
15  ITEM quit (TEXT="Quit", COMMENT="Exit the program", IMAGE="quit")
16 END
17
18 LAYOUT
19 VBOX
20 GROUP
21 GRID
22 {
23   Store #:[f01  ] [f02                                          ]
24   Order #:[f03  ]  Order Date:[f04         ] Ship By:[f06       ]
25   Factory:[f05  ]             [f07                             ]
26                                     Order Total:[f14           ]
27 }
28 END
29 END -- GROUP
30 TABLE
31 {
32  Stock#  Description       Qty     Unit    Price       Total
33 [f08    |f09              |f10    |f11    |f12        |f13      ]
34 [f08    |f09              |f10    |f11    |f12        |f13      ]
35 [f08    |f09              |f10    |f11    |f12        |f13      ]
36 [f08    |f09              |f10    |f11    |f12        |f13      ]
37 }
38 END
39 END
40 END
41
```

```
42 TABLES
43   customer, orders, items, stock
44 END
45
46 ATTRIBUTES
47  BUTTONEDIT f01 = orders.store_num, REQUIRED, ACTION=zoom1;
48  EDIT       f02 = customer.store_name, NOENTRY;
49  EDIT       f03 = orders.order_num, NOENTRY;
50  DATEEDIT   f04 = orders.order_date;
51  EDIT       f05 = orders.fac_code, UPSHIFT;
52  EDIT       f06 = orders.ship_instr;
53  CHECKBOX   f07 = orders.promo, TEXT="Promotional",
54                   VALUEUNCHECKED="N", VALUECHECKED="Y";
55  BUTTONEDIT f08 = items.stock_num, REQUIRED, ACTION=zoom2;
56  LABEL      f09 = stock.description;
57  EDIT       f10 = items.quantity, REQUIRED;
58  LABEL      f11 = stock.unit;
59  LABEL      f12 = items.price;
60  LABEL      f13 = formonly.line_total TYPE DECIMAL(9,2);
61  EDIT       f14 = formonly.order_total TYPE DECIMAL(9,2), NOENTRY;
62 END
63
64 INSTRUCTIONS
65 SCREEN RECORD sa_items(
66   items.stock_num,
67   stock.description,
68   items.quantity,
69   stock.unit,
70   items.price,
71   line_total
72 )
73 END
```

Note:

- Lines 03 thru 16 define a TOOLBAR section with typical actions.
- Lines 23 and 48 The field **f02** is a LABEL , allowing no editing. It displays the customer name associated with the orders store number
- Lines 19 and 49 Field **f03** is the order number from the orders table.
- Lines 25 and 53 The field **f07** is a CHECKBOX displaying the values of the column promo in the orders table. The box will appear checked if the value in the column is "Y", and unchecked if the value is "N".
- Lines 26 and 61 The field **f14** is a FORMONLY field This field displays the order total calculated by the BDL program logic.
- Lines 30 thru 38 describe the TABLE container for the screen array.
- Lines 33, 56 and 58 The fields **f09** and **f11** are LABELS, and display the description and unit of measure for the items stock number.
- Lines 33 and 60 the field **f13** is a LABEL and FORMONLY. This field displays the line total calculated for each line in the screen array.
- Lines 42 thru 44 The TABLES statement includes all the database tables that are listed for fields in the ATTRIBUTES section of the form.
- Line 47 The attribute REQUIRED forces the user to enter data in the field during an INPUT statement.
- Line 51 The attribute UPSHIFT makes the runtime system convert lowercase letters to uppercase letters, both on the screen display and in the program variable that stores the contents of this field.
- Line 65 The screen record includes the names of all the fields shown in the screen array.

The Orders Program orders.4gl

Much of the functionality is identical to that in earlier Tutorial examples. The query/add/delete/update of the `orders` table would be the same as the examples in Chapter 4 and Chapter 6. Only append and query are included in this program, for simplicity. The add/delete/update of the `items` table is similar to that in Chapter 8. The complete `orders` program is outlined, with examples of any new functionality.

The MAIN program block

The `MAIN` program block contains the menu for the `orders` program.

`MAIN` program block (`orders.4gl`):

```
01 SCHEMA custdemo
02
03 DEFINE  order_rec RECORD
04            store_num    LIKE orders.store_num,
05            store_name   LIKE customer.store_name,
06            order_num    LIKE orders.order_num,
07            order_date   LIKE orders.order_date,
08            fac_code     LIKE orders.fac_code,
09            ship_instr   LIKE orders.ship_instr,
10            promo        LIKE orders.promo
11         END RECORD,
12         arr_items DYNAMIC ARRAY OF RECORD
13            stock_num    LIKE items.stock_num,
14            description  LIKE stock.description,
15            quantity     LIKE items.quantity,
16            unit         LIKE stock.unit,
17            price        LIKE items.price,
18            line_total   DECIMAL(9,2)
19         END RECORD
20
21 CONSTANT msg01 = "You must query first"
22 CONSTANT msg02 = "Enter search criteria"
23 CONSTANT msg03 = "Canceled by user"
24 CONSTANT msg04 = "No rows in table"
25 CONSTANT msg05 = "End of list"
26 CONSTANT msg06 = "Beginning of list"
```

```
27 CONSTANT msg07 = "Invalid stock number"
28 CONSTANT msg08 = "Row added"
29 CONSTANT msg09 = "Row updated"
30 CONSTANT msg10 = "Row deleted"
31 CONSTANT msg11 = "Enter order"
32 CONSTANT msg12 = "This customer does not exist"
33 CONSTANT msg13 = "Quantity must be greater than zero"
34
35 MAIN
36   DEFINE has_order, query_ok SMALLINT
37   DEFER INTERRUPT
38
39   CONNECT TO "custdemo"
40   CLOSE WINDOW SCREEN
41
42   OPEN WINDOW w1 WITH FORM "orderform"
43
44   MENU
45     BEFORE MENU
46        CALL setup_actions(DIALOG,FALSE,FALSE)
47     ON ACTION new
48   CLEAR FORM
49        LET query_ok = FALSE
50        CALL close_order()
51        LET has_order = order_new()
52        IF has_order THEN
53           CALL arr_items.clear()
54           CALL items_inpupd()
55        END IF
56        CALL setup_actions(DIALOG,has_order,query_ok)
57     ON ACTION find
58        CLEAR FORM
59        LET query_ok = order_query()
60        LET has_order = query_ok
61        CALL setup_actions(DIALOG,has_order,query_ok)
62     ON ACTION next
63        CALL order_fetch_rel(1)
64     ON ACTION previous
65        CALL order_fetch_rel(-1)
66     ON ACTION getitems
67        CALL items_inpupd()
68     ON ACTION quit
69        EXIT MENU
70   END MENU
71
72   CLOSE WINDOW w1
73
74 END MAIN
```

Note:

- Lines 03 thru 11 define a record with fields for all the columns in the `orders` table, as well as `store_name` from the `customer` table.
- Lines 12 through 19 define a dynamic array with fields for all the columns in the `items` table, as well as `quantity` and `unit` from the `stock` table, and a calculated field `line_total`.
- Lines 21 thru 33 define constants to hold the program messages. This centralizes the definition of the messages, which can be used in any function in the module.
- Lines 44 thru 65 define the main menu of the application.
- Line 46 is executed before the menu is displayed; it calls the `setup_actions` function to disable navigation and item management actions by default. The `DIALOG` predefined object is passed as the first parameter to the function.

- Lines 47 thru 56 perform the add action to create a new order. The order_new function is called, and if it returns TRUE, the items_inpupd function is called to allow the user to enter items for the new order. Menu actions are enabled/disabled depending on the result of the operation, using the setup_actions function.
- Lines 57 thru 61 perform the find action to search for orders in the database. The order_query function is called and menu actions are enabled/disabled depending on the result of the operation, using the setup_actions function.
- Lines 62 thru 65 handle navigation in the order list after a search. Function order_fetch_rel is used to fetch the previous or next record.
- Line 67 calls the function items_inpupd to allow the user to edit the items associated with the displayed order .
- Line 72 closes the window before leaving the program.

Function setup_actions

This function is used by the main menu to enable or disable actions based on the context.

Function setup_actions (orders.4gl):

```
01 FUNCTION setup_actions(d, has_order, query_ok)
02   DEFINE d ui.Dialog,
03          has_order, query_ok SMALLINT
04   CALL d.setActionActive("next",query_ok)
05   CALL d.setActionActive("previous",query_ok)
06   CALL d.setActionActive("getitems",has_order)
07 END FUNCTION
```

Note:

- Line 01 Three parameters are passed to the function:
  - *d* - the predefined Dialog object
  - *has_order* - if the value is TRUE, indicates that there is a new or existing order selected.
  - *query_ok* - if the value is TRUE, indicates that the search for orders was successful.
- Lines 04 and 05 use the ui.Dialog.setActionActive method to enable or disable next and previous actions based on the value of query_ok, which indicates whether the search for orders was successful.
- Line 06 uses the same method to enable the getitems action based on the value of has_order, which indicates whether there is an order currently selected.

Function order_new

This function handles the input of an order record.

Function order_new (orders.4gl):

```
01 FUNCTION order_new()
02   DEFINE id INTEGER, name STRING
03
04   MESSAGE msg11
05   INITIALIZE order_rec.* TO NULL
06   SELECT MAX(order_num)+1 INTO order_rec.order_num
07     FROM orders
08   IF order_rec.order_num IS NULL
09    OR order_rec.order_num == 0 THEN
10      LET order_rec.order_num = 1
11   END IF
12
13   LET int_flag = FALSE
14   INPUT BY NAME
15       order_rec.store_num,
```

```
16       order_rec.store_name,
17       order_rec.order_num,
18       order_rec.order_date,
19       order_rec.fac_code,
20       order_rec.ship_instr,
21       order_rec.promo
22     WITHOUT DEFAULTS
23     ATTRIBUTES(UNBUFFERED)
24
25     BEFORE INPUT
26        LET order_rec.order_date = TODAY
27        LET order_rec.fac_code = "ASC"
28        LET order_rec.ship_instr = "FEDEX"
29        LET order_rec.promo = "N"
30
31     ON CHANGE store_num
32        SELECT store_name INTO order_rec.store_name
33          FROM customer
34         WHERE store_num = order_rec.store_num
35        IF (SQLCA.SQLCODE == NOTFOUND) THEN
36           ERROR msg12
37           NEXT FIELD store_num
38        END IF
39
40     ON ACTION zoom1
41        CALL display_custlist() RETURNING id, name
42        IF (id > 0) THEN
43           LET order_rec.store_num = id
44           LET order_rec.store_name = name
45        END IF
46
47   END INPUT
48
49   IF (int_flag) THEN
50      LET int_flag=FALSE
51      CLEAR FORM
52      MESSAGE msg03
53      RETURN FALSE
54   END IF
55
56   RETURN order_insert()
57
58 END FUNCTION
```

Note:

- Lines 06 and 11 execute a SELECT to get a new order number from the database; if no rows are found, the order number is initialized to 1.
- Lines 14 thru 47 use the INPUT interactive dialog statement to let the user input the order data.
- Lines 25 thru 29 the BEFORE INPUT block initializes some members of the order_rec record, as default values for input.
- Lines 31 thru 38 the ON CHANGE block on the store_num field retrieves the customer name for the changed store_num from the customer table, and stores it in the store_name field. If the customer doesn't exist in the customer table, an error message displays.
- Lines 40 thru 45 implement the code to open the zoom window of the store_num BUTTONEDIT field, when the action zoom1 is triggered. The function display_custlist in the custlist.4gl module allows the user to select a customer from a list. The action zoom1 is enabled during the INPUT statement only.
- Line 56 calls the order_insert function to perform the INSERT SQL statement.

Function order_insert

This function inserts a new record in the `orders` database table.

Function `order_insert` (`orders.4gl`):

```
01 FUNCTION order_insert()
02
03   WHENEVER ERROR CONTINUE
04   INSERT INTO orders (
05      store_num,
06      order_num,
07      order_date,
08      fac_code,
09      ship_instr,
10      promo
11    ) VALUES (
12      order_rec.store_num,
13      order_rec.order_num,
14      order_rec.order_date,
15      order_rec.fac_code,
16      order_rec.ship_instr,
17      order_rec.promo
18    )
19    WHENEVER ERROR STOP
20
21    IF (SQLCA.SQLCODE <> 0) THEN
22      CLEAR FORM
23      ERROR SQLERRMESSAGE
24      RETURN FALSE
25    END IF
27
28    MESSAGE "Order added"
29    RETURN TRUE
30
31 END FUNCTION
```

Note:

- Lines `03` thru `19` implement the `INSERT` SQL statement to create a new row in the `orders` table.
- Lines `21` thru `25` handle potential SQL errors, and display a message and return `FALSE` if the insert was not successful.
- Lines `28` and `29` display a message and return `TRUE` in case of success.

Function order_query

This function allows the user to enter query criteria for the `orders` table. It calls the function `order_select` to retrieve the rows from the database table.

Function `order_query` (`orders.4gl`):

```
01 FUNCTION order_query()
02   DEFINE where_clause STRING,
03          id INTEGER, name STRING
04
05   MESSAGE msg02
06
07   LET int_flag = FALSE
08   CONSTRUCT BY NAME where_clause ON
09       orders.store_num,
10       customer.store_name,
11       orders.order_num,
```

```
12       orders.order_date,
13       orders.fac_code
14
15     ON ACTION zoom1
16        CALL display_custlist() RETURNING id, name
17        IF id > 0 THEN
18           DISPLAY id TO orders.store_num
19           DISPLAY name TO customer.store_name
20        END IF
21
22   END CONSTRUCT
23
24   IF (int_flag) THEN
25      LET int_flag=FALSE
26      CLEAR FORM
27      MESSAGE msg03
28      RETURN FALSE
29   END IF
30
31   RETURN order_select(where_clause)
32
33 END FUNCTION
```

Note:

- Lines 08 thru 22 The CONSTRUCT statement allows the user to query on specific fields, restricting the columns in the orders table that can be used for query criteria.
- Lines 15 thru 20 handle the zoom1 action to let the user pick a customer from a list. The function display_custlist is called, it returns the customer number and name.
- Lines 24 through 29 check the value of the interrupt flag, and return FALSE if the user has interrupted the query.
- Line 31 the query criteria stored in the variable where_clause is passed to the function order_select. TRUE or FALSE is returned from the order_select function.

Function order_fetch

This function retrieves the row from the orders table, and is designed to be reused each time a row is needed. If the retrieval of the row from the orders table is successful, the function items_fetch is called to retrieve the corresponding rows from the items table.

Function order_fetch (orders.4gl):

```
01 FUNCTION order_fetch(p_fetch_flag)
02   DEFINE p_fetch_flag SMALLINT
03
04   IF p_fetch_flag = 1 THEN
05      FETCH NEXT order_curs INTO order_rec.*
06   ELSE
07      FETCH PREVIOUS order_curs INTO order_rec.*
08   END IF
09
10   IF (SQLCA.SQLCODE == NOTFOUND) THEN
11      RETURN FALSE
12   END IF
13
14   DISPLAY BY NAME order_rec.*
15   CALL items_fetch()
16   RETURN TRUE
17
18 END FUNCTION
```

Note:

- Line 05 When the parameter passed to this function and stored in the variable p_fetch_flag is 1, the FETCH statement retrieves the next row from the orders table.
- Line 07 When the parameter passed to this function and stored in p_fetch_flag is not 1, the FETCH statement retrieves the previous row from the orders table.
- Lines 10 thru 12 return FALSE if no row was found.
- Line 14 uses DISPLAY BY NAME to display the record order_rec.
- Line 15 calls the function items_fetch, to fetch all order lines.
- Line 16 returns TRUE indicating the fetch of the order was successful.

Function order_select

This function creates the SQL statement for the query and the corresponding cursor to retrieve the rows from the orders table. It calls the function fetch_order.

Function order_select (orders.4gl):

```
01 FUNCTION order_select(where_clause)
02   DEFINE where_clause STRING,
03          sql_text STRING
04
05   LET sql_text = "SELECT "
05       || "orders.store_num, "
06       || "customer.store_name, "
07       || "orders.order_num, "
08       || "orders.order_date, "
09       || "orders.fac_code, "
10       || "orders.ship_instr, "
11       || "orders.promo "
12       || "FROM orders, customer "
13       || "WHERE orders.store_num = customer.store_num "
14       || "AND " || where_clause
15
16   DECLARE order_curs SCROLL CURSOR FROM sql_text
17   OPEN order_curs
18   IF (NOT order_fetch(1)) THEN
19      CLEAR FORM
20      MESSAGE msg04
21      RETURN FALSE
22   END IF
23
24   RETURN TRUE
25
26 END FUNCTION
```

Note:

- Lines 05 thru 14 contain the text of the SELECT statement with the query criteria contained in the variable where_clause.
- Line 16 declares a SCROLL CURSOR for the SELECT statement stored in the variable sql_text.
- Line 17 opens the SCROLL CURSOR.
- Line 18 thru 22 call the function order_fetch, passing a parameter of 1 to fetch the next row, which in this case will be the first one. If the fetch is not successful, FALSE is returned.
- Line 24 returns TRUE, indicating the fetch was successful.

Function order_fetch_rel

This function calls the function `order_fetch` to retrieve the rows in the database; the parameter `p_fetch_flag` indicates the direction for the cursor movement. If there are no more records to be retrieved, a message is displayed to the user.

Function `order_fetch_rel`:

```
01 FUNCTION order_fetch_rel(p_fetch_flag)
02   DEFINE p_fetch_flag SMALLINT
03
04   MESSAGE " "
05   IF (NOT order_fetch(p_fetch_flag)) THEN
06     IF (p_fetch_flag = 1) THEN
07       MESSAGE msg05
08     ELSE
09       MESSAGE msg06
10     END IF
11   END IF
12
13 END FUNCTION
```

Note:

- Line `05` calls the function `order_fetch`, passing the variable `p_fletch_flag` to indicate the direction of the cursor.
- Line `07` displays a message to indicate that the cursor is at the bottom of the result set.
- Line `09` displays a message to indicate that the cursor is at the top of the result set.

Function order_total

This function calculates the total price for all of the items contained on a single order.

Function **order_total** (**orders.4gl**):

```
01 FUNCTION order_total(arr_length)
02   DEFINE order_total DECIMAL(9,2),
03          i, arr_length SMALLINT
04
05   LET order_total = 0
06   IF arr_length > 0 THEN
07      FOR i = 1 TO arr_length
08         IF arr_items[i].line_total IS NOT NULL THEN
09           LET order_total = order_total + arr_items[i].line_total
10         END IF
11      END FOR
12   END IF
13
14   DISPLAY BY NAME order_total
15
16 END FUNCTION
```

Note:

- Line `07` thru `11` contain a `FOR` loop adding the values of `line_total` from each item in the program array `arr_items`, to calculate the total price of the order and store it in the variable `order_total`.
- Line `14` displays the value of `order_total` on the form.

Function order_close

This function closes the cursor used to select orders from the database.

Function **order_close** (**orders.4gl**):

```
01 FUNCTION close_order()
02   WHENEVER ERROR CONTINUE
03   CLOSE order_curs
04   WHENEVER ERROR STOP
05 END FUNCTION
```

Note:

- Line 03 closes the order_curs cursor. The statement is surrounded by WHENEVER ERROR, to trap errors if the cursor is not open.

Function items_fetch

This function retrieves the rows from the items table that match the value of order_num in the order currently displayed on the form. The description and unit values are retrieved from the stock table, using the column stock_num. The value for line_total is calculated and retrieved. After displaying the items on the form, the function order_total is called to calculate the total price of all the items for the current order.

Function items_fetch (orders.4gl):

```
01 FUNCTION items_fetch()
02   DEFINE item_cnt INTEGER,
03          item_rec RECORD
04             stock_num    LIKE items.stock_num,
05             description  LIKE stock.description,
06             quantity     LIKE items.quantity,
07             unit         LIKE stock.unit,
08             price        LIKE items.price,
09             line_total   DECIMAL(9,2)
10          END RECORD
11
12   IF order_rec.order_num IS NULL THEN
13      RETURN
14   END IF
15
16   DECLARE items_curs CURSOR FOR
17      SELECT items.stock_num,
18             stock.description,
19             items.quantity,
20             stock.unit,
21             items.price,
22             items.price * items.quantity line_total
23          FROM items, stock
24         WHERE items.order_num = order_rec.order_num
25           AND items.stock_num = stock.stock_num
26
27   LET item_cnt = 0
28   CALL arr_items.clear()
29   FOREACH items_curs INTO item_rec.*
30       LET item_cnt = item_cnt + 1
31       LET arr_items[item_cnt].* = item_rec.*
32   END FOREACH
33   FREE items_curs
34
35   CALL items_show()
36   CALL order_total(item_cnt)
37
```

```
38 END FUNCTION
```

Note:

- Line 02 defines a variable item_cnt to hold the array count.
- Line 12 returns from the function if the order number in the program record order_rec is NULL.
- Lines 16 thru 25 declare a cursor for the SELECT statement to retrieve the rows from the items table that have the same order number as the value in the order_num field of the program record order_rec. The description and unit values are retrieved from the stock table, using the column stock_num. The value for line_total is calculated.
- Lines 29 thru32 the FOREACH statement loads the dynamic array arr_items.
- Line 33 releases the memory associated with the cursor items_curs, which is no longer needed.
- Lines 35 calls the items_show function to display the order lines to the form.
- Line 36 calls the function order_total to calculate the total price of the items on the order.

Function items_show

This function displays the line items for the order in the screen array and returns immediately.

Function **items_show** (**orders.4gl**):

```
01 FUNCTION items_show()
02   DISPLAY ARRAY arr_items TO sa_items.*
03       BEFORE DISPLAY
04         EXIT DISPLAY
05   END DISPLAY
06 END FUNCTION
```

Note:

- Line 02 executes a DISPLAY ARRAY statement with the program array containing the line items.
- Line 03 and 04 exit the instruction before control is turned over to the user.

Function items_inpupd

This function contains the program logic to allow the user to input a new row in the arr_items array, or to change or delete an existing row.

Function items_inpupd:

```
01 FUNCTION items_inpupd()
02   DEFINE opflag CHAR(1),
03          item_cnt, curr_pa SMALLINT,
04          id INTEGER
05
06   LET opflag = "U"
07
08   LET item_cnt = arr_items.getLength()
09   INPUT ARRAY arr_items WITHOUT DEFAULTS FROM sa_items.*
10     ATTRIBUTES (UNBUFFERED, INSERT ROW = FALSE)
11
12     BEFORE ROW
13       LET curr_pa = ARR_CURR()
14       LET opflag = "U"
15
16     BEFORE INSERT
17       LET opflag = "I"
18       LET arr_items[curr_pa].quantity = 1
19
20     AFTER INSERT
```

```
21        CALL item_insert(curr_pa)
22        CALL items_line_total(curr_pa)
23
24      BEFORE DELETE
25        CALL item_delete(curr_pa)
26
27      ON ROW CHANGE
28        CALL item_update(curr_pa)
29        CALL items_line_total(curr_pa)
30
31      BEFORE FIELD stock_num
32        IF opflag = "U" THEN
33           NEXT FIELD quantity
34        END IF
35
36      ON ACTION zoom2
37           LET id = display_stocklist()
38           IF id > 0 THEN
39              IF (NOT get_stock_info(curr_pa,id) ) THEN
40                 LET arr_items[curr_pa].stock_num = NULL
41              ELSE
42                 LET arr_items[curr_pa].stock_num = id
43              END IF
44           END IF
45
46      ON CHANGE stock_num
47           IF (NOT get_stock_info(curr_pa,
48                          arr_items[curr_pa].stock_num) ) THEN
49              LET arr_items[curr_pa].stock_num = NULL
50              ERROR msg07
51              NEXT FIELD stock_num
52           END IF
53
54      ON CHANGE quantity
55           IF (arr_items[curr_pa].quantity <= 0) THEN
56              ERROR msg13
57              NEXT FIELD quantity
58           END IF
59
60    END INPUT
61
62    LET item_cnt = arr_items.getLength()
63    CALL ord_total(item_cnt)
64
65    IF (int_flag) THEN
66       LET int_flag = FALSE
67    END IF
68
69  END FUNCTION
```

Note:

- Line `08` uses the `getLength` built-in function to determine the number of rows in the array `arr_items`.
- Lines `9` thru `60` contain the `INPUT ARRAY` statement.
- Lines `12` and `14` use a `BEFORE ROW` clause to store the index of the current row of the array in the variable `curr_pa`. We also set the `opflag` flag to "U", in order to indicate we are in update mode.
- Lines `16` thru `18` use a `BEFORE INSERT` clause to set the value of `opflag` to "I" if the current operation is an Insert of a new row in the array. Line `18` sets a default value for the `quantity`.

- Lines 20 thru 22 An AFTER INSERT clause calls the item_insert function to add the row to the database table, passing the index of the current row and calls the items_line_total function, passing the index of the current row.
- Lines 24 thru 25 use a BEFORE DELETE clause, to call the function item_delete, passing the index of the current row.
- Lines 27 thru 29 contain an ON ROW CHANGE clause to detect row modification. The item_update function and the items_line_total function are called, passing the index of the current row.
- Lines 31 thru 34 use a BEFORE FIELD clause to prevent entry in the stock_num field if the current operation is an update of an existing row.
- Lines 36 thru 44 implement the code for the zoom2 action, opening a list from the stock table for selection.
- Lines 46 thru 52 use an ON CHANGE clause to check whether the stock number for a new record that was entered in the field stock_num exists in the stock table.
- Line 62 uses the getLength built-in function to determine the number of rows in the array after the INPUT ARRAY statement has terminated.
- Line 63 calls the function order_total, passing the number of rows in the array.
- Lines 65 thru 67 reset the INT_FLAG to TRUE if the user has interrupted the INPUT statement.

Function items_line_total

This function calculates the value of line_total for any new rows that are inserted into the arr_items array.

Function items_line_total:

```
01 FUNCTION items_line_total(curr_pa)
02   DEFINE curr_pa SMALLINT
03   LET arr_items[curr_pa].line_total =
04       arr_items[curr_pa].quantity * arr_items[curr_pa].price
05 END FUNCTION
```

Note:

- Line 02 The index of the current row in the array is passed to this function and stored in the variable curr_pa.
- Lines 03 and 04 calculate the line_total for the current row in the array.

Function item_insert

This function inserts a new row into the items database table using the values input in the current array record on the form.

Function item_insert:

```
01 FUNCTION item_insert(curr_pa)
02   DEFINE curr_pa SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   INSERT INTO items (
06      order_num,
07      stock_num,
08      quantity,
09      price
10   ) VALUES (
11      order_rec.order_num,
12      arr_items[curr_pa].stock_num,
13      arr_items[curr_pa].quantity,
14      arr_items[curr_pa].price
15   )
```

```
16   WHENEVER ERROR STOP
17
18   IF (SQLCA.SQLCODE == 0) THEN
19      MESSAGE msg08
20   ELSE
21      ERROR SQLERRMESSAGE
22   END IF
23
24 END FUNCTION
```

Note:

- Line `02` the index of the current row in the array is passed to this function and stored in the variable `curr_pa`.
- Lines `05` thru `15` The embedded SQL `INSERT` statement uses the value of `order_num` from the current order record displayed on the form, together with the values from the current row of the `arr_items` array, to insert a new row in the `items` table.

Function item_update

This function updates a row in the `items` database table using the changes made to the current array record in the form.

Function `item_update`:

```
01 FUNCTION item_update(curr_pa)
02   DEFINE curr_pa SMALLINT
03
04   WHENEVER ERROR CONTINUE
05   UPDATE items SET
06     items.stock_num = arr_items[curr_pa].stock_num,
07     items.quantity  = arr_items[curr_pa].quantity
08      WHERE items.stock_num = arr_items[curr_pa].stock_num
09        AND items.order_num = order_rec.order_num
10   WHENEVER ERROR STOP
11
12   IF (SQLCA.SQLCODE == 0) THEN
13      MESSAGE msg09
14   ELSE
15      ERROR SQLERRMESSAGE
16   END IF
17
18 END FUNCTION
```

Note:

- Line `02` the index of the current row in the array is passed to this function and stored in the variable `curr_pa`.
- Lines `05` thru `09` The embedded SQL `UPDATE` statement uses the value of `order_num` in the current `order_rec` record, and the value of `stock_num` in the current row in the `arr_items` array, to locate the row in the `items` database table to be updated.

Function item_delete

This function deletes a row from the `items` database table, based on the values in the current record of the items array.

Function `item_delete`:

```
01 FUNCTION item_delete(curr_pa)
02   DEFINE curr_pa SMALLINT
03
04   WHENEVER ERROR CONTINUE
```

```
05   DELETE FROM items
06      WHERE items.stock_num = arr_items[curr_pa].stock_num
07      AND items.order_num = order_rec.order_num
08   WHENEVER ERROR STOP
09
10   IF (SQLCA.SQLCODE == 0) THEN
11      MESSAGE msg10
12   ELSE
13      ERROR SQLERRMESSAGE
14   END IF
15
16 END FUNCTION
```

Note:

- Line `02` the index of the current row in the array is passed to this function and stored in the variable `curr_pa`.
- Lines `05` thru `07` The embedded SQL `DELETE` statement uses the value of `order_num` in the current `order_rec` record, and the value of `stock_num` in the current row in the `arr_items` array, to locate the row in the `items` database table to be deleted.

Function get_stock_info

This function verifies that the stock number entered for a new row in the `arr_items` array exists in the `stock` table. It retrieves the description, unit of measure, and the correct price based on whether promotional pricing is in effect for the order.

Function `get_stock_info`:

```
01 FUNCTION get_stock_info(curr_pa, id)
02   DEFINE curr_pa SMALLINT,
03          id INTEGER,
04          sqltext STRING
05
06   IF id IS NULL THEN
07      RETURN FALSE
08   END IF
09
10   LET sqltext="SELECT description, unit,"
11   IF order_rec.promo = "N" THEN
12      LET sqltext=sqltext || "reg_price"
13   ELSE
14      LET sqltext=sqltext || "promo_price"
15   END IF
16   LET sqltext=sqltext ||
17       " FROM stock WHERE stock_num = ? AND fac_code = ?"
18
19   WHENEVER ERROR CONTINUE
20   PREPARE get_stock_cursor FROM sqltext
21   EXECUTE get_stock_cursor
22         INTO arr_items[curr_pa].description,
23              arr_items[curr_pa].unit,
24              arr_items[curr_pa].price
25         USING id, order_rec.fac_code
26   WHENEVER ERROR STOP
27
28   RETURN (SQLCA.SQLCODE == 0)
29
30 END FUNCTION
```

Note:

- Line `02` the index of the current row in the array is passed to this function and stored in the variable `curr_pa`.
- Lines `10` thru `17` check whether the promotional pricing is in effect for the current order, and build a `SELECT` statement to retrieve the description, unit, and regular or promotional price from the `stock` table for a new item that is being added to the `items` table.
- Lines `20` thru `25` prepare and execute the SQL statement created before.
- Line `28` checks `SQLCA.SQLCODE` and returns `TRUE` if the database could be updated without error.

Tutorial Chapter 12: Changing the User Interface Dynamically

This chapter focuses on using the classes and methods in the `ui` package of built-in classes to modify the user interface at runtime. Among the techniques illustrated are hiding or disabling form items; changing the text, style or image associated with a form item; loading a combobox from a database table; and adding toolbars and topmenus dynamically.

Built-in Classes

Included in the predefined functions that are built into Genero are special groups (classes) of functions (methods) that act upon the objects that are created when your program is running. Each class of methods interacts with a specific program object, allowing you to change the appearance or behavior of the objects. Because these methods act upon program objects, the syntax is somewhat different from that of functions.

The classes are gathered together into packages:

- `ui` - classes related to the objects in the graphical user interface (GUI)
- `base` - classes related to non-GUI program objects
- `om` - classes that provide DOM and SAX document handling utilities

This tutorial focuses on using the classes and methods in the `ui` package to modify the user interface at runtime.

> **Note:** Variable names, class identifiers, and method names are not case-sensitive; the capitalization used in the examples is for ease in reading.

Using the Classes

This example for the `Window` class also presents the general process that you should use.

The methods in the `Window` class interact with the Window objects in your program.

Getting a reference to the object

Before you can call any of the methods associated with `Window` objects, you must identify the specific `Window` object that you wish to affect, and obtain a reference to it:

- Define a variable to hold the reference to the `Window` object. The data type of the variable is the class identifier (`ui.Window`):

```
DEFINE mywin ui.Window
```

- Open a window in your program using the `OPEN WINDOW` or `OPEN WINDOW ... WITH FORM` instruction:

```
 OPEN WINDOW w1 WITH FORM "testform"
```

- Get a reference to the specific `Window` object by using one of two class methods provided by the `Window` class. class methods are called using the class identifier (`ui.Window`). You can specify the `Window` object by name from among the open windows in your program, or choose the current window.

```
LET mywin = ui.Window.getCurrent() -- returns a reference to
                                   -- the current window object
LET mywin = ui.Window.forName("w1")-- returns a reference to
                                   -- the open window named "w1"
```

Calling a method

Now that you have a reference to the object, you can use that reference to call any of the methods listed as object methods in the `Window` class documentation. For example, to change the window title for the window referenced by `mywin`:

```
CALL mywin.setText("test")
```

See *The Window class* in the *Genero Business Development Language User Guide* for a complete list of the methods in this class.

Example 1

```
01 MAIN
02 DEFINE mywin ui.Window
03
04 OPEN WINDOW w1 WITH FORM "testform"
05 LET mywin = ui.Window.getCurrent()
06 CALL mywin.setText("test")
07 MENU
08  ON ACTION quit
09    EXIT MENU
10 END MENU
11
12 END MAIN
```

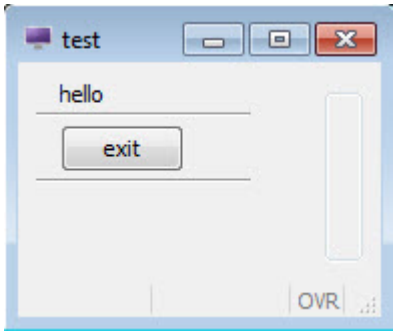

Figure 24: Form with window title changed by the ui.Window.setText method

Working with Forms

The `Form` class provides some methods that allow you to change the appearance or behavior of items on a form.

Getting a reference to the Form object

In order to use the methods, you must get a reference to the `form` object. The `Window` class has a method to get the reference to its associated form:

- Define variables for the references to the window object and to its form object. The data type for the variables is the class identifier (`ui.Window`, `ui.Form`):

```
DEFINE f1 ui.Form, mywin ui.Window
```

- Open a form in your program using the `OPEN WINDOW ... WITH FORM` instruction:

```
OPEN WINDOW w1 WITH FORM ("testform")
```

- Next, get a reference to the window object. Then, use the `getForm()` class method of the `Window` class to get a reference to the form object opened in that window:

```
LET mywin = ui.Window.getCurrent()
LET f1 = mywin.getForm() -- returns reference to form
```

Once you have the reference to the form object, you can call any of the object methods for the Form class

```
LET mywin = ui.Window.getCurrent()
LET f1 = mywin.getForm() -- get reference to form
-- call a Form class method
CALL f1.loadActionDefaults("mydefaults")
```

See *The Form class* section of the *Genero Business Development Language User Guide* for a complete list of methods.

Specifying the name of a form item

Some of the methods in the `Form` class require you to provide the name of the form item. The name of the form item in the `ATTRIBUTES` section of the form specification file corresponds to the `name` attribute of an element in the runtime form file. For example:

- In the `ATTRIBUTES` section of the `.per` file

```
LABEL a1: lb1, TEXT = "State";
EDIT a2 = state.state_name;
BUTTON a3: quit, TEXT = "exit";
EDIT a4 = FORMONLY.pflag TYPE CHAR;
```

- In the runtime `.42f` file

```
<Label name="lb1" width="9" text="State" posY="0" posX="6" gridWidth="9"/>
<FormField name="state.state_name" colName="state_name"
 sqlType="CHAR(15)"
    fieldId="0" sqlTabName="state" tabIndex="1">
<Button name="quit" width="5" text="exit" posY="4" posX="6" gridWidth="5"/
>
<FormField name="formonly.pflag" colName="pflag" sqlType="CHAR"
 fieldId="1"
    sqlTabName="formonly" tabIndex="2">
```

> **Note:** Formfield names specified as `FORMONLY` (`FORMONLY.pflag`) are converted to lowercase (`formonly.pflag`).

Although Genero BDL is not case-sensitive, XML is. When Genero creates the runtime XML file, the form item types and attribute names are converted using the CamelCase convention:

- Form item type - the first letter is always capitalized, with subsequent letters in lowercase, unless the type consists of multiple words joined together. In that case, the first letter of every subsequent word is capitalized also (Label, FormField, Button).
- Attribute name - the first letter is always lowercase, with subsequent letters in lowercase, unless the name consists of multiple words joined together. In that case, the first letter of every subsequent word is capitalized also (text, gridWidth, colName).

If you use classes or methods in your code that require the form item type or attribute name, respect the naming conventions.

Changing the text, image, and style properties of a form item

Some methods of the `Form` class allow you to change the value of specific properties of form items.

Call the methods using the reference to the form object. Provide the name of the form item and the value for the property:

- `Text` property - the value can be any text string. To set the text of the label named `lb1`:

```
CALL f1.setElementText("lb1", "Newtext")
```

- `Image` property - the value can be a simple file name, a complete or relative path, or an URL (Uniform Resource Locator) path to an image server. To set the image for the button named `quit`:

```
CALL f1.setElementImage("quit", "exit.jpg" placement="break")
```

- `Style` property - the value can be a presentation style defined in the active Presentation Styles file (`.4st` file). To set the style for the label named `lb1`:

```
CALL f1.setElementStyle("lb1", "mystyle")
```

The style `mystyle` is an example of a specific style that was defined in a custom Presentation Styles XML file, `customstyles.4st`. This style changes the text color to blue:

```
<Style name=".mystyle" >
  <StyleAttribute name="textColor" value="blue" />
</Style>
```

By default, the runtime system searches for the `default.4st` Presentation Style file. Use the following method to load a different Presentation Style file:

```
CALL ui.interface.loadStyles("customstyles")
```

See *Presentation styles* in the *Genero Business Development Language User Guide* for additional information about styles and the format of a Presentation Styles file.

Example 2

```
01 MAIN
02  DEFINE mywin ui.Window,
03           f1    ui.Form
04   CALL ui.interface.loadStyles("customstyles")
05  OPEN WINDOW w1 WITH FORM "testform"
06  LET mywin = ui.Window.getCurrent()
```

```
07    CALL mywin.setText("test")
08    LET f1 = mywin.getForm()
09  MENU
10    ON ACTION changes
11      CALL f1.setElementText("lb1", "goodbye")
12      CALL f1.setElementText("quit", "leave")
13      CALL f1.setElementImage("quit", "exit.png")
14      CALL f1.setElementStyle("lb1", "mystyle")
15    ON ACTION quit
16      EXIT MENU
17  END MENU
18 END MAIN
```

Figure 25: Display on Windows™ platform after the changes button has been clicked.

Hiding Form Items

You can use `Form` class methods to change the value of the `hidden` property of form items, hiding parts of the form from the user.

Interactive instructions such as `INPUT` or `CONSTRUCT` will automatically ignore a formfield that is hidden. The value can be:

- `0` - the form item is not hidden; it is visible
- `1` - the form item is hidden and cannot be made visible by the user
- `2` - the form item is hidden, but the user can make it visible, using the context menu for a table, for example

By default, all form items are visible.

Call the methods using the reference to the form object. Provide the name of the form item to the method and set the value for hidden.

- `setFieldHidden()` - this method can be used to hide formfields only. The prefix in the name of the formfield (`tablename.` or `formonly.`) is optional:

  ```
  CALL f1.setFieldHidden("state_name",1)
  ```

- `setElementHidden()` - this method hides any form item, including formfields. If the item is a formfield, the name must include the prefix:

  ```
  CALL f1.setElementHidden("lb1", 1)
  CALL f1.setElementHidden("state.state_name",1)
  CALL f1.setElementHidden("formonly.pflag",1)
  ```

Genero adjusts the display of the form to eliminate blank spaces caused by hiding items, where possible.

Example 3

```
01 SCHEMA custdemo
02 MAIN
03  DEFINE win ui.Window,
04         fm ui.Form,
05         mycust record like customer.*
06  CONNECT TO "custdemo"
07  OPEN WINDOW w1 WITH FORM "hidecust"
08  SELECT * INTO mycust.* FROM customer
09      WHERE store_num = 101
10  DISPLAY BY NAME mycust.*
11  LET win = ui.Window.getCurrent()
12 LET fm = win.getForm()
13  MENU
14   ON ACTION hide
15     CALL fm.setFieldHidden("contact_name",1)
16     CALL fm.setFieldHidden("addr2", 1)
17     -- hide the label for contact name
18     CALL fm.setElementHidden("lbl", 1)
19   ON ACTION quit
20     EXIT MENU
21  END MENU
22 END MAIN
```

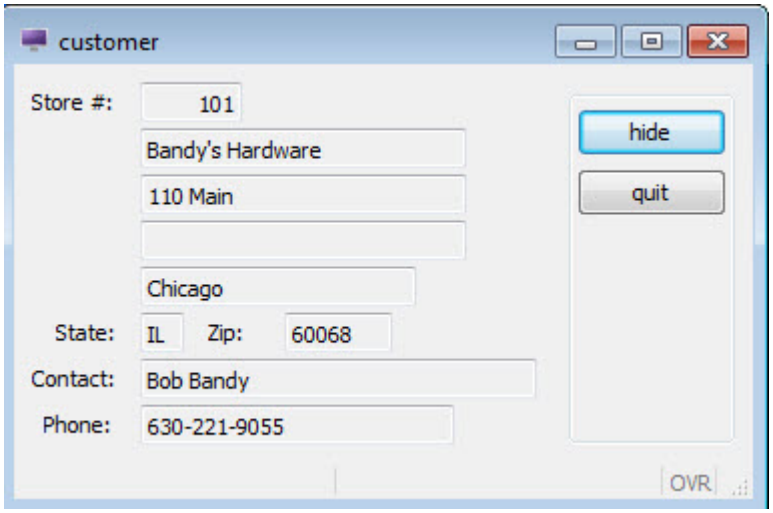

Figure 26: Form before hiding element

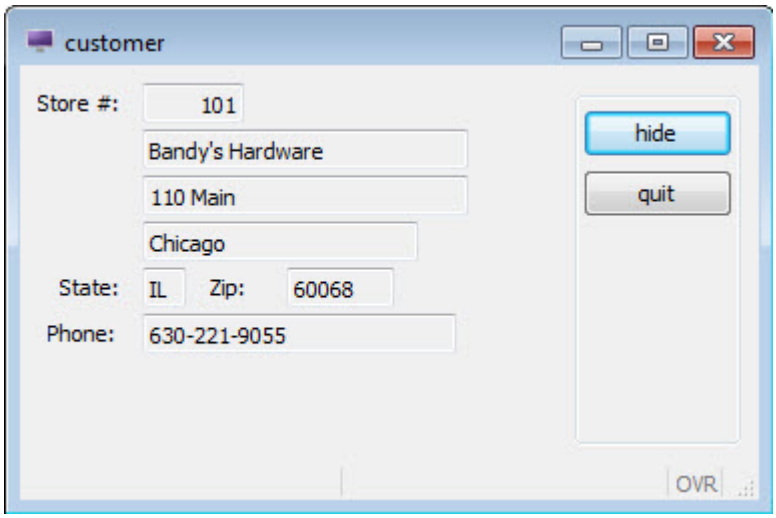

Figure 27: Form after hiding element

Adding toolbars, topmenus, and action defaults

The `Form` class provides methods that apply topmenus, toolbars, and action defaults to a form, to assist you in standardizing forms.

The topmenus, toolbars, or action defaults are defined in external XML resource files having the following extensions:

- Action Defaults - `.4ad`
- Toolbar - `.4tb`
- Topmenu - `.4tm`

Call the methods using the reference to the form object and specify the resource file name. Do not specify a path or file extension in the file name. If the file is not in the current directory and the path is not specified, Genero will search the directories indicated by the `DBPATH` / `FGLRESOURCEPATH` environment variable.

- Action defaults file - default attributes for form items associated with actions; these action defaults are local to the form. See *Action defaults files* in the *Genero Business Development Language User Guide* for information about the format and contents of the file.

```
CALL f1.loadActionDefaults("mydefaults")
```

- Toolbar file - contains a toolbar definition to be used with the referenced form object. See *Toolbars* in the *Genero Business Development Language User Guide* for information about the format and contents of the file.

```
CALL f1.loadToolBar("mytoolbar")
```

- Topmenu file - contains a topmenu definition to be used with the referenced form object. See *Topmenus* in the *Genero Business Development Language User Guide* for information about the format and contents of the file.

```
CALL f1.loadTopMenu("mytopmenu")
```

Example 4

```
01 MAIN
02 DEFINE mywin ui.Window,
03         f1    ui.Form
```

```
04 OPEN WINDOW w1 WITH FORM "testform"
05 LET mywin = ui.Window.forName("w1")
06 CALL mywin.setText("test")
07  LET f1 = mywin.getForm()
08  CALL f1.loadTopMenu("mytopmenu")
09 MENU
10   ON ACTION quit
11     EXIT MENU
12 END MENU
13
14 END MAIN
```

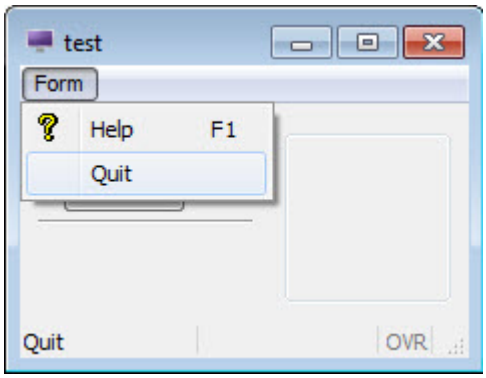

Figure 28: Display of a topmenu on Windows™ platform

Specifying a Function to Initialize all Forms

To assist in standardizing forms, you can create an initializer function in your program that will be called automatically whenever any form is opened. A reference to the form object is passed by the runtime system to the function.

Example initializer function:

```
01 FUNCTION myforminit(f1)
02  DEFINE f1 ui.Form
03
04  CALL f1.loadTopMenu("mytopmenu")
05   ...
06
07 END FUNCTION
```

The `setDefaultInitializer()` method applies to all forms, rather than to a specific form object. It is a class method, and you call it using the class name as a prefix. Specify the name of the initializer function in lowercase letters:

```
CALL ui.Form.setDefaultInitializer("myforminit")
```

You can call the `myforminit` function in your program as part of a setup routine. The `myforminit` function can be in any module in the program.

Example 5

```
01 MAIN
02  CALL ui.Form.setDefaultInitializer("myforminit")
03  OPEN WINDOW w1 WITH FORM "testform"
04  MENU
05   ON ACTION quit
```

```
06    EXIT MENU
07  END MENU
08  OPEN WINDOW w2 WITH FORM "testform2"
09  MENU
10   ON ACTION quit
11    EXIT MENU
12  END MENU
13 END MAIN
```

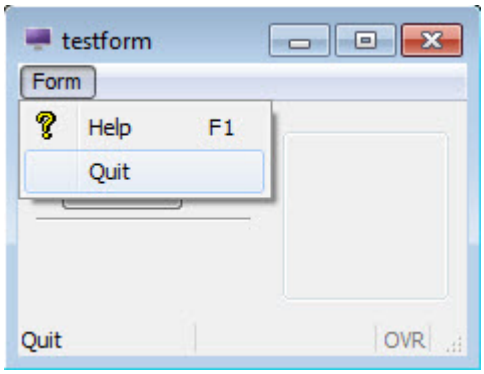

Figure 29: Form testform with initializer function

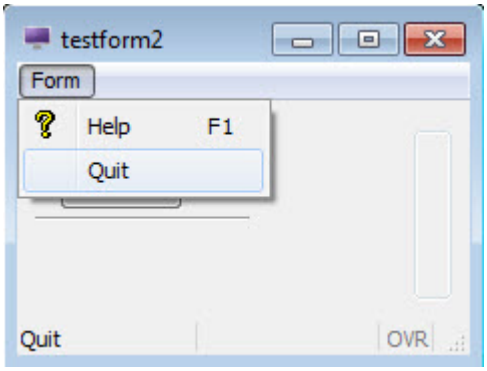

Figure 30: Form testform2 using the same initializer function.

Loading a ComboBox List

A ComboBox presents a list of values in a dropdown box on a form. The values are for the underlying formfield. For example, the following form specification file contains a ComboBox that represents the formfield `customer.state`:

```
01 SCHEMA custdemo
02 LAYOUT
03 GRID
04  {
05   Store #:[a0  ]
06      Name:[a1                ]
07     State:[a5      ]
08  }
09  END -- GRID
10 END
11 TABLES customer
12 ATTRIBUTES
13  EDIT a0=customer.store_num;
14  EDIT a1=customer.store_name;
15  COMBOBOX a5=customer.state;
```

```
16 END
```

During an `INPUT`, `INPUT ARRAY` or `CONSTRUCT` statement the ComboBox is active, and the user can select a value from the dropdown list. The value selected will be stored in the formfield named `customer.state`.

Getting a reference to the object

The `ComboBox` class contains methods that manage the values for a ComboBox. In order to use these methods you must first obtain a reference to the ComboBox object:

- Define a variable for the reference to the ComboBox object. The data type for the variables is the class identifier (`ui.ComboBox`):

  ```
  DEFINE cb ui.ComboBox
  ```

- Open a form that contains a ComboBox using `OPEN WINDOW ... WITH FORM`:

  ```
  OPEN WINDOW w1 WITH FORM ("testcb")
  ```

- Next, get a reference to the ComboBox object using the method provided. As a class method, this method is called using the class identifier. Provide the name of the formfield to the method:

  ```
  LET cb = ui.ComboBox.forName("customer.state")
  ```

Once you have a reference to the ComboBox object, you can call any of the methods defined in the class as object methods:

- To add an item to a ComboBox list

  You can instruct the ComboBox to store a code (the `name`) in the formfield that the ComboBox represents, but to display the description (the `text`) in the list to help the user make his selection. For example, to store the value "IL" (`name`) in the formfield, but to display "Illinois" (`text`) to the user:

  ```
  CALL cb.additem("IL", "Illinois")
  ```

  If `text` is `NULL`, `name` will be displayed.
- To clear the list of all values

  ```
  CALL cb.clear()
  ```

- To remove an item from the list; provide the `name`

  ```
  CALL cb.removeitem("IL")
  ```

See the *The ComboBox class* documentation in the *Genero Business Development Language User Guide* for a complete list of the methods.

Adding values to the ComboBox from a Database Table

An example in Tutorial Chapter 5 GUI Options loads a ComboBox with static values. The following example retrieves the valid list of values from a database table (`state`) instead:

Example 6

```
01 SCHEMA custdemo
02 MAIN
03  DEFINE cb ui.ComboBox
04  CONNECT TO "custdemo"
05  OPEN WINDOW w1 WITH FORM "testcb"
```

```
06  LET cb = ui.ComboBox.forName("customer.state")
07  IF cb IS NOT NULL THEN
08   CALL loadcb(cb)
09  END IF
10 ...
11 END MAIN
12
13 FUNCTION loadcb(cb)
12  DEFINE cb ui.ComboBox,
13         l_state_code LIKE state.state_code,
14         l_state_name LIKE state.state_name
15
18  DECLARE mycurs CURSOR FOR
19    SELECT state_code, state_name FROM state
20  CALL cb.clear()
21  FOREACH mycurs INTO l_state_code, l_state_name
22  -- provide name and text for the ComboBox item
23   CALL cb.addItem(l_state_code,l_state_name)
24  END FOREACH
26 END FUNCTION
```

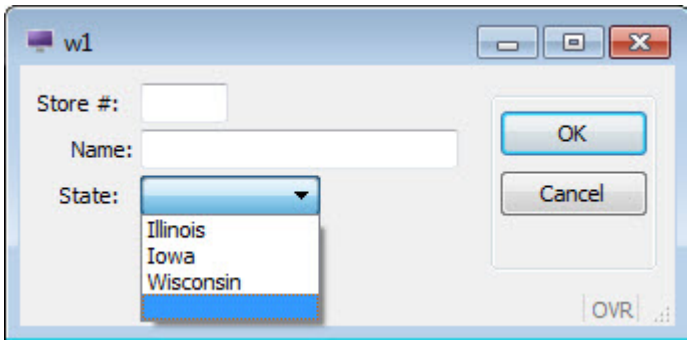

Figure 31: Loaded combobox

As an alternative to calling the `loadcb` function in your BDL program, this function can be specified as the initializer function for the ComboBox in the form specification file. When the form is opened, The initializer function is called automatically and a reference to the ComboBox object is passed to it. Provide the name of the initializer function in lowercase:

```
ATTRIBUTES
COMBOBOX a5=customer.state, INITIALIZER = loadcb;
```

Using the Dialog class in Interactive Statements

The `Dialog` class provides methods that can only be called from within an interactive instruction (dialog) such as `MENU`, `INPUT`, `INPUT ARRAY`, `DISPLAY ARRAY` and `CONSTRUCT`.

The methods are called through the predefined variable `DIALOG`, which automatically provides a reference to the Dialog object.

Tutorial Chapter 5 Enhancing the Form illustrates the use of `Dialog` class methods to disable/enable actions during a `MENU` interactive statement.

Hiding Default Action Views

To hide default action views (the buttons that appear on the form when there is no specific action view for an action), use the following `Dialog` class method.

Values for the hidden state of the action view can be:

- 0 - `FALSE`, the action is visible
- 1 - `TRUE`, the action is hidden

```
MENU
  BEFORE MENU
    CALL DIALOG.setActionHidden("next",1)
    ...
END MENU
```

This example hides the action that has the name `next`. The reference to the `DIALOG` object was provided by the runtime system.

Enabling and Disabling Fields

This method in the `Dialog` class allows you to disable fields on a form during the interactive statement; the field is still visible, but the user cannot edit the value.

Values for the active state of the field can be:

- 0 - `FALSE`, the field is disabled
- 1 - `TRUE`, the field is enabled

The reference to the `DIALOG` object is provided by the runtime system. Provide the name of the field and its state to the method.

The following example disables the `store_name` field during an `INPUT` statement:

```
   INPUT BY NAME customer.*
    BEFORE INPUT
      CALL DIALOG.setFieldActive("customer.store_name",0)
    ...
   END INPUT
```

See the *The Dialog class* section in the *Genero Business Development Language User Guide* for a complete list of its methods.

Using the Interface Class

Methods in the `Interface` class allow you interact with the user interface, as shown in the examples.

You do not need to get an object reference to the Interface; call the methods in the `Interface` class using the class identifier, `ui.Interface`.

Refresh the interface

The User Interface on the Client is synchronized with the DOM tree of the runtime system when an interactive statement is active. 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.

As shown in the Tutorial Chapter 9 Reports, the changes made in the program to the value of the progress bar are not displayed on the user's window, since the report is a batch process and no user interaction is

required. To force the changes in the progress bar to be reflected on the screen, the following method from the `Interface` Class is used:

```
CALL ui.Interface.refresh()
```

Load custom XML files

- Start Menus, Toolbar icons, and Topmenus can each be defined in a unique XML file.

Use the appropriate extension:

- Start Menu - `.4sm`
- Toolbar - `.4tb`
- Topmenu - `.4tm`

Use the corresponding method to load the file:

```
CALL ui.Interface.loadStartMenu("mystartmenu")
CALL ui.Interface.loadTopMenu("tmstandard")
CALL ui.Interface.loadToolbar("tbstandard")
```

Do not specify a path or file extension in the file name. The runtime system automatically searches for a file with the correct extension in the current directory and in the path list defined in the `DBPATH` / `FGLRESOURCEPATH` environment variable.

See the *Loading a start menu from an XML file*, *ui.Interface.loadTopMenu*, or *ui.Interface.loadToolBar* documentation in the *Genero Business Development Language User Guide* for details on the format and contents of the files.

- Custom Presentation Styles and global Action Defaults must each be defined in a unique file.

Use the appropriate extension:

- Presentation Styles - `.4st`
- Action Defaults - `.4ad`

Use the corresponding method to load the file:

```
CALL ui.Interface.loadStyles("mystyles")
CALL ui.Interface.loadActionDefaults("mydefaults")
```

You can provide an absolute path with the corresponding extension, or a simple file name without the extension. If you give the simple file name, the runtime system searches for the file in the current directory. If the file does not exist, it searches in the directories defined in the `DBPATH`/ `FGLRESOURCEPATH` environment variable.

The action defaults are applied only once, to newly created elements. For example, if you first load a toolbar, then you load a global Action defaults file, the attribute of the toolbar items will not be updated with the last loaded Action defaults.

See *Presentation styles* and *Action defaults files* in the *Genero Business Development Language User Guide* for details on the format and contents of the file.

Identify the Genero client

You can use methods in the `Interface` Class to identify the type and version of the Genero client currently being used by the program:

```
CALL ui.Interface.getFrontEndName() RETURNING typestring
CALL ui.Interface.getFrontEndVersion() RETURNING versionstring
```

Each method returns a string. The type will be "Gdc" or "Console".

Some of the other methods in the `ui.Interface` class allow you to:

- Set and retrieve program names and titles
- Call Front End functions that reside on the Genero client
- Work with MDI windows

See the *The Interface class* documentation in the *Genero Business Development Language User Guide* for a complete list of the methods.

Tutorial Chapter 13: Master/Detail using Multiple Dialogs

This chapter shows how to implement order and items input in a unique `DIALOG` statement. In chapter 11 the order input is detached from the items input. The code example in chapter 13 makes both order and item input fields active at the same time, which is more natural in GUI applications.

The Master-Detail sample

The example discussed in this chapter is designed for the input of order information (headers and order lines), illustrating a typical master-detail relationship. The form used by the example contains fields from both the `orders` and `items` tables in the `custdemo` database. The result is very similar to the example of chapter 11. However, in this program the end user can input order and items data simultaneously, because the form is driven by a `DIALOG` instruction.

When the program starts, the existing rows from the `orders` and `items` tables have already been retrieved and are displayed on the form. The user can browse through the orders and items to update or delete them, add new orders or items, and search for specific orders by entering criteria in the form.

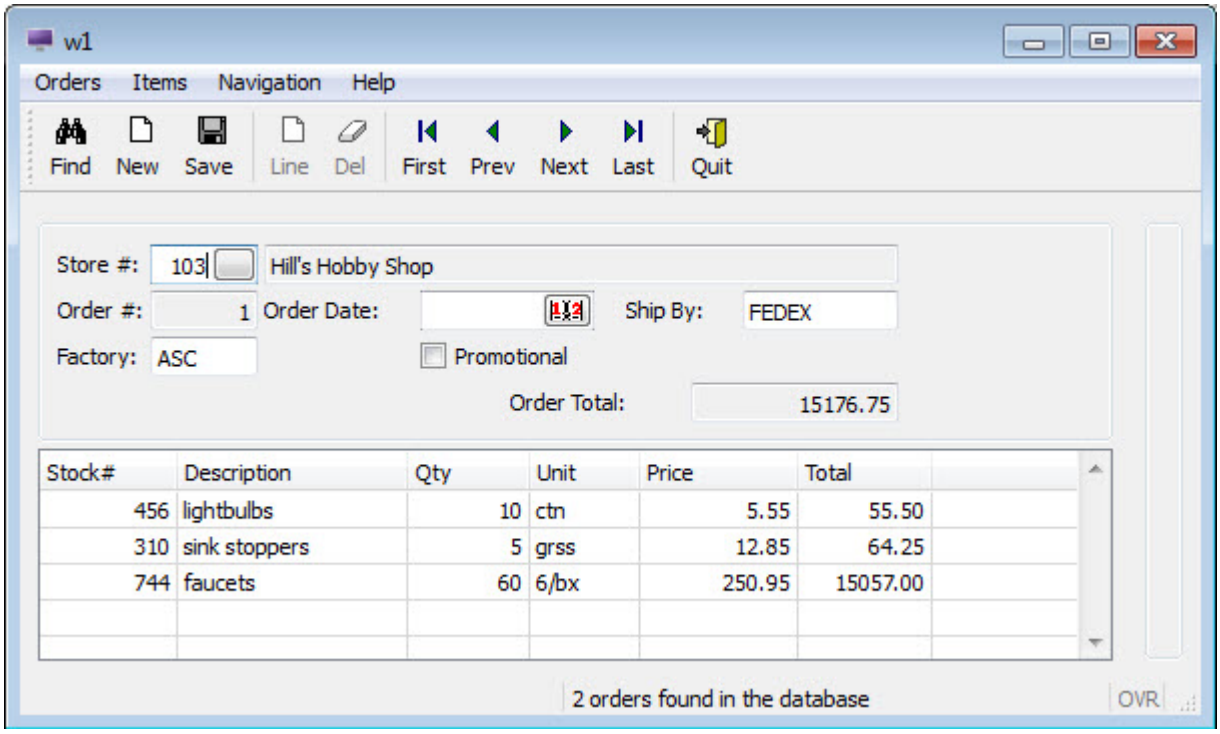

Figure 32: Master-Detail form

There are different ways to implement a Master/Detail form with multiple dialogs. This chapter shows one of them. Genero provides the basics bricks, then it's up to you to adapt the programming pattern, according to the ergonomics you want to expose to the end user.

The Customer List Form

The Customer List form displays when the user clicks the button next to the store number field (the buttonEdit widget). The `custlist.per` form defines a typical 'zoom' form with a filter field and record list where the user can pick an element to be used in a field of the main form. Using this form, the user can scroll through the list to pick a store, or can enter query criteria to filter the list prior to picking. The fields that make up the columns of the table that display the list are defined as `FORMONLY` fields. When `TYPE` is not defined, the default data type for `FORMONLY` fields is `CHAR`.

Form `custlist.per`:

```
001 SCHEMA custdemo
002
003 LAYOUT
004 GRID
005 {
006 <g g1                                          >
007  Store name: [fc                     :fe   ]
008 <                                             >
009 <t t1                                         >
010   Id   Name                  City
011  [f01 |f02                  |f03            ]
012  [f01 |f02                  |f03            ]
013  [f01 |f02                  |f03            ]
014  [f01 |f02                  |f03            ]
015 <                                             >
016 }
017 END
018 END
019
020 TABLES
021    customer
022 END
023
024 ATTRIBUTES
025 GROUP g1: TEXT="Filter";
026 EDIT fc = customer.store_name;
027 BUTTON fe: fetch, IMAGE="filter";
028 EDIT f01=FORMONLY.s_num;
029 EDIT f02=FORMONLY.s_name;
030 EDIT f03=FORMONLY.s_city;
031 END
032
033 INSTRUCTIONS
034 SCREEN RECORD sa_cust (FORMONLY.*);
035 END
```

Note:

- Line `001` defines the database schema to be used by this form.
- Lines `003` thru `018` define a `LAYOUT` section that describes the layout of the form.
  - Lines `006` thru `008` define a `GROUPBOX` with the `fc` field where the user can enter a search criteria, and the `fe` button to trigger the query.
  - Lines `009` thru `015` define a `TABLE` that will be used to display the result set of the query.
- Lines `020` thru `022` define a `TABLES` section to reference database schema tables.
- Lines `024` thru `031` define an `ATTRIBUTES` section with the details of form fields.
  - Line `026` defines the query field with a reference to the `customer.store_name` database column. This will implicitly define the data type of the field and the Query by Example input rules.

    - Line `027` defines the `BUTTON` that will invoke the database query.
    - Lines `028` thru `030` define the columns of the table with the `FORMONLY` prefix.
- Lines `033` thru `035` define an `INSTRUCTIONS` section to group item fields in a screen array.

The Customer List Module

The `custlist.4gl` module defines a 'zoom' module, to let the user select a customer from a list. The module could be reused for any application that requires the user to select a customer from a list.

This module uses the `custlist.per` form and is implemented with a `DIALOG` instruction defining a `CONSTRUCT` sub-dialog and a `DISPLAY ARRAY` sub-dialog. The `display_custlist()` function in this module returns the customer id and the name.

In the application illustrated in this chapter, the main module `orders.4gl` will call the `display_custlist()` function to retrieve a customer selected by the user.

```
01   ON ACTION zoom1
02     CALL display_custlist() RETURNING id, name
03     IF (id > 0) THEN
04         ...
```

Here is the complete source code.

Module `custlist.4gl`:

```
001 SCHEMA custdemo
002
003 TYPE cust_t RECORD
004            store_num     LIKE customer.store_num,
005            store_name    LIKE customer.store_name,
006            city          LIKE customer.city
007        END RECORD
008
009 DEFINE cust_arr DYNAMIC ARRAY OF cust_t
010
011 FUNCTION custlist_fill(where_clause)
012   DEFINE where_clause STRING
013   DEFINE idx SMALLINT
014   DEFINE cust_rec cust_t
015
016   DECLARE custlist_curs CURSOR FROM
017     "SELECT store_num, store_name, city "||
018     "  FROM customer"||
019     "  WHERE "||where_clause||
020     " ORDER BY store_num"
021
022   LET idx = 0
023   CALL cust_arr.clear()
024   FOREACH custlist_curs INTO cust_rec.*
025     LET idx = idx + 1
026     LET cust_arr[idx].* = cust_rec.*
027   END FOREACH
028
029 END FUNCTION
030
031 FUNCTION display_custlist()
033   DEFINE ret_num LIKE customer.store_num
034   DEFINE ret_name LIKE customer.store_name
035   DEFINE where_clause STRING
036   DEFINE idx SMALLINT
037
```

```
038   OPEN WINDOW wcust WITH FORM "custlist"
039
040   LET ret_num = 0
041   LET ret_name = NULL
042
043   DIALOG ATTRIBUTES(UNBUFFERED)
044
045      CONSTRUCT BY NAME where_clause ON customer.store_name
046      END CONSTRUCT
047
048      DISPLAY ARRAY cust_arr TO sa_cust.*
049      END DISPLAY
050
051      BEFORE DIALOG
052         CALL custlist_fill("1 = 1")
053
054      ON ACTION fetch
055         CALL custlist_fill(where_clause)
056
057      ON ACTION accept
058         LET idx = DIALOG.getCurrentRow("sa_cust")
059         IF idx > 0 THEN
060            LET ret_num = cust_arr[idx].store_num
061            LET ret_name = cust_arr[idx].store_name
062            EXIT DIALOG
063         END IF
064
065      ON ACTION cancel
066         EXIT DIALOG
067
068   END DIALOG
069
070   CLOSE WINDOW wcust
071
072   RETURN ret_num, ret_name
073
074 END FUNCTION
```

Note:

- Line `001` defines the database schema to be used by this module.
- Lines `003` thru `007` define the `cust_t TYPE` as a `RECORD` with three members declared with a `LIKE` reference to the database column.
- Line `009` defines the `cust_arr` program array with the type defined in previous lines.
- Lines `011` thru `029` define the `custlist_fill()` function which fills `cust_arr` with the values of database rows.
  - Lines `016` thru `020` declare the `custlist_curs` SQL cursor by using the `where_clause` condition passed as the parameter.
  - Lines `022` thru `027` fetch the database rows into `cust_arr`.
- Lines `031` thru `074` implement the `display_custlist()` function to be called by the main module.
  - Lines `040` and `041` initialize the `ret_num` and `ret_name` variables. If the user cancels the dialog, the function will return these values to let the caller decide what to do.
  - Lines `043` thru `068` define a `DIALOG` instruction implementing the controller of the form.
    - Lines `045` thru `046` define the `CONSTRUCT` sub-dialog controlling the `customer.store_name` query field.
    - Lines `048` thru `049` define the `DISPLAY ARRAY` sub-dialog controlling the `sa_cust` screen array.

- Lines 051 thru 052 implement the BEFORE DIALOG trigger, to fill the list with an initial result set by passing the query criteria as "1 =1" to the cust_list_fill() function.
- Lines 054 thru 055 implement the fetch ON ACTION trigger, executed when the user presses the fe button in the form, to fill the list with a result set by passing the query criteria in where_clause to the cust_list_fill function.
- Lines 057 thru 063 implement the accept ON ACTION trigger, executed when the user validates the dialog with the **OK** button or with a double-click in a row of the list. The code initializes the return values ret_num and ret_name with the current row.
- Lines 065 thru 066 implement the cancel ON ACTION trigger, to leave the dialog when the user hits the **Cancel** button.

- Line 072 returns the values of the ret_num and ret_name variables.

The Orders Form

The form specification file orderform.per defines a form for the orders program, and displays fields containing the values of a single order from the orders table. The name of the store is retrieved from the customer table, using the column store_num, and displayed.

A screen array displays the associated rows from the items table. Although order_num is also one of the fields in the items table, it does not have to be included in the screen array or in the screen record, since the order number will be the same for all the items displayed for a given order. For each item displayed in the screen array, the values in the description and unit columns from the stock table are also displayed.

The values in FORMONLY fields are not retrieved from a database; they are calculated by the BDL program based on the entries in other fields. In this form FORMONLY fields are used to display the calculations made by the BDL program for item line totals and the order total. Their data type is defined as DECIMAL.

This form uses some of the attributes that can be assigned to fields in a form. See the *ATTRIBUTES section* in the *Genero Business Development Language User Guide* for a complete list of the available attributes.

The form defines a toolbar and a topmenu. The decoration of toolbar or topmenu action views is centralized in an ACTION DEFAULTS section.

Form orderform.per:

```
001 SCHEMA custdemo
002
003 ACTION DEFAULTS
004   ACTION find (TEXT="Find", IMAGE="find",
 COMMENT="Query database")
005   ACTION new (TEXT="New", IMAGE="new",
 COMMENT="New order")
006   ACTION save (TEXT="Save", IMAGE="disk",
 COMMENT="Check and save order info")
007   ACTION append (TEXT="Line", IMAGE="new",
 COMMENT="New order line")
008   ACTION delete (TEXT="Del", IMAGE="eraser",
 COMMENT="Delete current order line")
009   ACTION first (TEXT="First",
 COMMENT="Move to first order in list")
010   ACTION previous (TEXT="Prev",
 COMMENT="Move to previous order in list")
011   ACTION next (TEXT="Next",
 COMMENT="Move to next order in list")
012   ACTION last (TEXT="Last",
 COMMENT="Move to last order in list")
013   ACTION quit (TEXT="Quit",
 COMMENT="Exit the program", IMAGE="quit")
```

```
014 END
015
016 TOPMENU
017   GROUP ord (TEXT="Orders")
018     COMMAND find
019     COMMAND new
020     COMMAND save
021     SEPARATOR
022      COMMAND quit
023   END
024   GROUP ord (TEXT="Items")
025     COMMAND append
026     COMMAND delete
027   END
028   GROUP navi (TEXT="Navigation")
029     COMMAND first
030     COMMAND previous
031     COMMAND next
032     COMMAND last
033   END
034   GROUP help (TEXT="Help")
035     COMMAND about (TEXT="About")
036   END
037 END
038
039 TOOLBAR
040   ITEM find
041   ITEM new
042   ITEM save
043   SEPARATOR
044   ITEM append
045   ITEM delete
046   SEPARATOR
047   ITEM first
048   ITEM previous
049   ITEM next
050   ITEM last
051   SEPARATOR
052   ITEM quit
053 END
054
055 LAYOUT
056 VBOX
057 GROUP
058 GRID
059 {
060   Store #:[f01  ] [f02                                          ]
061   Order #:[f03  ]  Order Date:[f04         ] Ship By:[f06       ]
062   Factory:[f05  ]             [f07                              ]
063                                    Order Total:[f14           ]
064 }
065 END
066 END -- GROUP
067 TABLE
068 {
069  Stock#  Description       Qty     Unit    Price       Total
070 [f08    |f09              |f10    |f11    |f12        |f13      ]
071 [f08    |f09              |f10    |f11    |f12        |f13      ]
072 [f08    |f09              |f10    |f11    |f12        |f13      ]
073 [f08    |f09              |f10    |f11    |f12        |f13      ]
074 }
075 END
076 END
077 END
```

```
078
079 TABLES
080   customer, orders, items, stock
081 END
082
083 ATTRIBUTES
084  BUTTONEDIT f01 = orders.store_num, REQUIRED, ACTION=zoom1;
085  EDIT       f02 = customer.store_name, NOENTRY;
086  EDIT       f03 = orders.order_num, NOENTRY;
087  DATEEDIT   f04 = orders.order_date;
088  EDIT       f05 = orders.fac_code, UPSHIFT;
089  EDIT       f06 = orders.ship_instr;
090  CHECKBOX   f07 = orders.promo, TEXT="Promotional",
091                   VALUEUNCHECKED="N", VALUECHECKED="Y";
092  BUTTONEDIT f08 = items.stock_num, REQUIRED, ACTION=zoom2;
093  LABEL      f09 = stock.description;
094  EDIT       f10 = items.quantity, REQUIRED;
095  LABEL      f11 = stock.unit;
096  LABEL      f12 = items.price;
097  LABEL      f13 = formonly.line_total TYPE DECIMAL(9,2);
098  EDIT       f14 = formonly.order_total TYPE DECIMAL(9,2), NOENTRY;
099 END
100
101 INSTRUCTIONS
102 SCREEN RECORD sa_items(
103   items.stock_num,
104   stock.description,
105   items.quantity,
106   stock.unit,
107   items.price,
108   line_total
109 )
110 END
```

Note:

- Line `001` defines the database schema to be used by this form.
- Lines `003` thru `014` define a `ACTION DEFAULTS` section with view defaults such as text and comments.
- Lines `016` thru `037` define a `TOPMENU` section for a pull-down menu.
- Lines `039` thru `053` define a `TOOLBAR` section for a typical toolbar.
- Lines `055` thru `077` define a `LAYOUT` section that describes the layout of the form.
- Lines `079` thru `081` define a `TABLES` section to list all the database schema tables that are referenced for fields in the `ATTRIBUTES` section of the form.
- Lines `083` thru `099` define an `ATTRIBUTES` section with the details of form fields.
  - Lines `084` and `092` define `BUTTONEDIT` fields, with buttons that allow the user to trigger actions defined in the `.4gl` module.
- Lines `101` thru `110` define an `INSTRUCTIONS` section to group item fields in a screen array.

The Orders Program orders.4gl

The `orders.4gl` module implements the main form controller. Most of the functionality has been described in previous chapters. In this section we will only focus on the `DIALOG` instruction programming. The program implements a `DIALOG` instruction, including an `INPUT BY NAME` sub-dialog for the order fields input, and an `INPUT ARRAY` sub-dialog for the items input. Unlike traditional 4GL programs using singular dialogs, you typically start the program in the multiple dialog instruction, eliminating the global `MENU` instruction.

- Module variables of orders.4gl on page 156

Module variables of orders.4gl

The module variables are used by the `orders.4gl` module.

Module variables of `orders.4gl`

```
001 SCHEMA custdemo
002
003 TYPE order_t RECORD
004            store_name   LIKE customer.store_name,
005            order_num    LIKE orders.order_num,
006            order_date   LIKE orders.order_date,
007            fac_code     LIKE orders.fac_code,
008            ship_instr   LIKE orders.ship_instr,
009            promo        LIKE orders.promo
010       END RECORD,
011       item_t RECORD
012            stock_num    LIKE items.stock_num,
013            description  LIKE stock.description,
014            quantity     LIKE items.quantity,
015            unit         LIKE stock.unit,
016            price        LIKE items.price,
017            line_total   DECIMAL(9,2)
018       END RECORD
019
020 DEFINE order_rec order_t,
021      arr_ordnums DYNAMIC ARRAY OF INTEGER,
022      orders_index INTEGER,
023      arr_items DYNAMIC ARRAY OF item_t,
024      order_total DECIMAL(9,2)
025
026 CONSTANT title1 = "Orders"
027 CONSTANT title2 = "Items"
028
029 CONSTANT msg01 = "You must query first"
030 CONSTANT msg02 = "Enter search criteria"
031 CONSTANT msg03 = "Canceled by user"
032 CONSTANT msg04 = "No rows found, enter new search criteria"
033 CONSTANT msg05 = "End of list"
034 CONSTANT msg06 = "Beginning of list"
035 CONSTANT msg07 = "Invalid stock number"
036 CONSTANT msg08 = "Row added to the database"
037 CONSTANT msg09 = "Row updated in the database"
038 CONSTANT msg10 = "Row deleted from the database"
039 CONSTANT msg11 = "New order record created"
040 CONSTANT msg12 = "This customer does not exist"
041 CONSTANT msg13 = "Quantity must be greater than zero"
042 CONSTANT msg14 = "%1 orders found in the database"
043 CONSTANT msg15 = "There are no orders selected, exit program?"
044 CONSTANT msg16 = "Item is not available in current factory %1"
045 CONSTANT msg17 = "Order %1 saved in database"
046 CONSTANT msg18 = "Order input program, version 1.01"
047 CONSTANT msg19 = "To save changes, move focus to another row
 or to the order header"
048
049 CONSTANT move_first = -2
050 CONSTANT move_prev  = -1
```

```
051 CONSTANT move_next  = 1
052 CONSTANT move_last  = 2
```

Note:

- Line `001` defines the database schema to be used by this module.
- Lines `003` thru `010` define the `order_t TYPE` as a `RECORD` with six members declared with a `LIKE` reference to the database column. This type will be used for the `orders` records.
- Lines `011` thru `018` define the `item_t TYPE` as a `RECORD` to be used for the `items` records.
- Line `020` defines the `order_rec` variable, to hold the data of the current order header.
- Line `021` defines the `arr_ordnums` array, to hold the list of order numbers fetched from the last query. This array will be used to navigate in the current list of orders.
- Line `022` defines the `orders_index` variable, defining the current order in the `arr_ordnums` array.
- Line `023` defines the `arr_items` array with the `item_t` type, to hold the lines of the current order.
- Line `024` defines the `order_total` variable, containing the order amount.
- Lines `026` thru `047` define string constants with text messages used by the `orders.4gl` module.
- Lines `049` thru `052` define numeric constants used for the `order_move()` navigation function.

Function orditems_dialog

This is the most important function of the program. It implements the multiple dialog instruction to control order and items input simultaneously.

The function uses the `opflag` variable to determine the state of the operations for `items`:

- `N` - no current operation
- `T` - temporary row was created
- `I` - row insertion was done in the list
- `M` - row in the list was modified

Function `orditems_dialog` (`orders.4gl`)

```
001 FUNCTION orditems_dialog()
002   DEFINE query_ok SMALLINT,
003          id INTEGER,
004          name LIKE customer.store_name,
005          opflag CHAR(1),
006          curr_pa INTEGER
007
008   DIALOG ATTRIBUTES(UNBUFFERED)
009
010    INPUT BY NAME order_rec.*, order_total
011      ATTRIBUTES(WITHOUT DEFAULTS, NAME="order")
012
013     ON ACTION find
014        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
015        CALL order_query()
016
017     ON ACTION new
018        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
019       IF NOT order_new() THEN
020          EXIT PROGRAM
021       END IF
022
023     ON ACTION save
024        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
025
```

```
026     ON CHANGE store_num
027        IF NOT order_check_store_num() THEN NEXT FIELD CURRENT END IF
028
029     ON ACTION zoom1
030        CALL display_custlist() RETURNING id, name
031        IF id > 0 THEN
032           LET order_rec.store_num = id
033           LET order_rec.store_name = name
034           CALL DIALOG.setFieldTouched("store_num", TRUE)
035        END IF
036
037     AFTER INPUT
038        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
039
040     ON ACTION first
041        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
042        CALL order_move(move_first)
043     ON ACTION previous
044        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
045        CALL order_move(move_prev)
046     ON ACTION next
047        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
048        CALL order_move(move_next)
049     ON ACTION last
050        IF NOT order_update(DIALOG) THEN NEXT FIELD CURRENT END IF
051        CALL order_move(move_last)
052
053   END INPUT
054
055   INPUT ARRAY arr_items FROM sa_items.*
056     ATTRIBUTES (WITHOUT DEFAULTS, INSERT ROW =FALSE)
057
058     BEFORE INPUT
059       MESSAGE msg19
060
061     BEFORE ROW
062       LET opflag = "N"
063       LET curr_pa = DIALOG.getCurrentRow("sa_items")
064       CALL DIALOG.setFieldActive("stock_num", FALSE)
065
066     BEFORE INSERT
067       LET opflag = "T"
068       LET arr_items[curr_pa].quantity = 1
069       CALL DIALOG.setFieldActive("stock_num", TRUE)
070
071     AFTER INSERT
072       LET opflag = "I"
073
074     BEFORE DELETE
075       IF opflag="N" THEN
076          IF NOT item_delete(curr_pa) THEN
077             CANCEL DELETE
078          END IF
079       END IF
080
081     AFTER DELETE
082       LET opflag="N"
083
084     ON ROW CHANGE
085       IF opflag != "I" THEN LET opflag = "M" END IF
086
087     AFTER ROW
088       IF opflag == "I" THEN
089          IF NOT item_insert(curr_pa) THEN
```

```
090             NEXT FIELD CURRENT
091          END IF
092          CALL items_line_total(curr_pa)
093       END IF
094       IF opflag == "M" THEN
095          IF NOT item_update(curr_pa) THEN
096             NEXT FIELD CURRENT
097          END IF
098          CALL items_line_total(curr_pa)
099       END IF
100
101     ON ACTION zoom2
102        LET id = display_stocklist()
103        IF id > 0 THEN
104           IF NOT get_stock_info(curr_pa,id) THEN
105              LET arr_items[curr_pa].stock_num = NULL
106           ELSE
107              LET arr_items[curr_pa].stock_num = id
108           END IF
109           CALL DIALOG.setFieldTouched("stock_num", TRUE)
110        END IF
111
112     ON CHANGE stock_num
113        IF NOT get_stock_info(curr_pa,
114                  arr_items[curr_pa].stock_num) THEN
115           LET arr_items[curr_pa].stock_num = NULL
116           CALL __mbox_ok(title2,msg07,"stop")
117           NEXT FIELD stock_num
118        ELSE
119           CALL items_line_total(curr_pa)
120        END IF
121
122     ON CHANGE quantity
123        IF arr_items[curr_pa].quantity <= 0 THEN
124           CALL __mbox_ok(title2,msg13,"stop")
125           NEXT FIELD quantity
126        ELSE
127           CALL items_line_total(curr_pa)
128        END IF
129
130   END INPUT
131
132   BEFORE DIALOG
133      IF NOT order_select("1=1") THEN
134        CALL order_query()
135      END IF
136
137   ON ACTION about
138      CALL __mbox_ok(title1,msg18,"information")
139
140   ON ACTION quit
141      EXIT DIALOG
142
143   END DIALOG
144
145 END FUNCTION
```

Note:

- Lines `002` thru `006` define the variables used by this function.
- Lines `008` thru `143` define a `DIALOG` instruction implementing the controller of the form.

- Lines 010 thru 053 implement the INPUT BY NAME sub-dialog, controlling the order_rec record input. All actions triggers declared inside the INPUT BY NAME sub-dialog will only be activated if the focus is in this sub-dialog. Data validation will occur when focus is lost by this sub-dialog, or when the user presses the **Save** button.
  - Lines 013 thru 015 implement the find ON ACTION trigger, to execute a Query By Example with the order_query() function. Before calling the query function, we must validate and save current modifications in the order record with the order_update() function. If the validation/save fails, the cursor remains in the current field (when the user clicks an action view, such as a Toolbar icon, the focus does not change.)
  - Lines 017 thru 021 implement the new ON ACTION trigger, to create a new order record. Before calling the new function, we must validate and save current modifications in the order record with the order_update() function.
  - Lines 023 thru 024 implement the save ON ACTION trigger, to validate and save current modifications in the order record with the order_update() function.
  - Lines 026 thru 027 declare the ON CHANGE trigger for the store_num field, to check if the number is a valid store identifier with the order_check_store_num() function. If the function returns FALSE, we execute a NEXT FIELD to stay in the field.
  - Lines 029 thru 035 implement the zoom1 ON ACTION trigger for the f01 field, to open a typical "zoom" window with the display_custlist() function. If the user selects a customer from the list, we mark the field as touched with the DIALOG.setFieldTouched() method. This simulates a real user input.
  - Lines 037 thru 038 implement the AFTER INPUT trigger, to validate and save current modifications with the order_update() function when the focus is lost by the order header sub-dialog.
  - Lines 040 thru 051 implement the ON ACTION triggers for the four navigation actions to move in the order list with the order_move() function. Before calling the query function, we must validate and save current modifications with the order_update() function.
- Lines 055 thru 130 implement the INPUT ARRAY sub-dialog, controlling the arr_items array input. All actions triggers declared inside the INPUT ARRAY sub-dialog will only be activated if the focus is in this sub-dialog. The sub-dialog uses the opflag technique to implement SQL instructions inside the dialog code and update the database on the fly.
  - Lines 058 thru 059 implement the BEFORE INPUT trigger, to display information message to the user, indicating that item row data will be validated and saved in the database when the user moves to another row or when the focus is lost by the item list.
  - Lines 061 thru 064 implement the BEFORE ROW trigger, initialize the opflag operation flag to "N" (no current operation), save the current row index in curr_pa variable and disable the stock_num field (only editable when creating a new line).
  - Lines 066 thru 069 implement the BEFORE INSERT trigger, to set the opflag to "T" (meaning a temporary row was created). A row will be fully validated and ready for SQL INSERT when we reach the AFTER INSERT trigger, there we will set opflag to "I". The code initializes the quantity to 1 and enables the stock_num field for user input.
  - Lines 071 thru 072 implement the AFTER INSERT trigger, to set the opflag to "I" (row insertion done in list). Data is now ready to be inserted in the database. This is done in the AFTER ROW trigger, according to opflag.
  - Lines 074 thru 079 implement the BEFORE DELETE trigger. We execute the SQL DELETE only if opflag equals "N", indicating that we are in a normal browse mode (and not inserting a new temporary row, which can be deleted from the list without any associated SQL instruction).
  - Lines 081 thru 082 implement the AFTER DELETE trigger, to reset the opflag to "N" (no current operation). This is done to clean the flag after deleting a new inserted row, when data validation or SQL insert failed in AFTER ROW. In that case, opflag equals "I" in the next AFTER DELETE / AFTER ROW sequence and would invoke validation rules again.

- Lines 084 thru 085 implement the ON ROW CHANGE trigger, to set the opflag to "M" (row was modified), but only if we are not currently doing a row insertion: Row insertion can have failed in AFTER ROW and AFTER INSERT would not be executed again, but ON ROW CHANGE would. The real SQL UPDATE will be done later in AFTER ROW.
- Lines 087 thru 099 implement the AFTER ROW trigger, executing INSERT or UPDATE SQL instructions according to the opflag flag. If the SQL statement fails (for example, because a constraint is violated), we set the focus back to the current field with NEXT FIELD CURRENT and keep the opflag value as is. If the SQL instruction succeeds, opflag will be reset to "N" in the next BEFORE ROW.
- Lines 101 thru 103 implement the zoom2 ON ACTION trigger for the f08 field, to open a typical "zoom" window with the display_stocklist() function. If the user selects a stock from the list, we mark the field as touched with the DIALOG.setFieldTouched() method. This simulates a real user input.
- Lines 112 thru 120 declare the ON CHANGE trigger for the stock_num field, to check if the number is a valid stock identifier with the get_stock_info() lookup function. If the function returns FALSE, we execute a NEXT FIELD to stay in the field, otherwise we recalculate the line total with items_line_total().
- Lines 122 thru 128 declare the ON CHANGE trigger for the quantity field, to check if the value is greater than zero. If the value is invalid, we execute a NEXT FIELD to stay in the field, otherwise we recalculate the line total with items_line_total().

- Lines 132 thru 134 implement the BEFORE DIALOG trigger, to fill the list of orders with an initial result set.
- Lines 137 thru 138 implement the about ON ACTION trigger, to display a message box with the version of the program.
- Lines 140 thru 141 implement the quit ON ACTION trigger, to leave the dialog (and quit the program).

Function order_update

This function validates that the values in the order_rec program record are correct, and then executes an SQL statement to update the row in the orders database table.

Function order_update (orders.4gl):

```
01 FUNCTION order_update(d)
02   DEFINE d ui.Dialog
03
04   IF NOT order_validate(d) THEN RETURN FALSE END IF
05
06   WHENEVER ERROR CONTINUE
07   UPDATE orders SET
08           store_num  = order_rec.store_num,
09           order_date = order_rec.order_date,
10           fac_code   = order_rec.fac_code,
11           ship_instr = order_rec.ship_instr,
12           promo      = order_rec.promo
13     WHERE orders.order_num = order_rec.order_num
14   WHENEVER ERROR STOP
15
16   IF SQLCA.SQLCODE <> 0 THEN
17     CALL __mbox_ok(title1,SQLERRMESSAGE,"stop")
18     RETURN FALSE
19   END IF
20
21   CALL d.setFieldTouched("orders.*", FALSE)
22   MESSAGE SFMT(msg17, order_rec.order_num)
23
24   RETURN TRUE
```

```
25
26 END FUNCTION
```

Note:

- Line 01 Since you cannot use the DIALOG keyword outside the DIALOG statement, a dialog object is passed to this function in order to use the methods of the DIALOG class.
- Line 04 calls the order_validate function, passing the dialog object. If the fields in the dialog are not validated, the function returns without updating the database row.
- Lines 06 thru 14 execute the SQL statement to update a row in the orders database table using values from the order_rec program record.
- Lines 16 thru18 return an error and exits the function if the SQLCA.SQLCODE indicates the database update was not successful.
- Lines 21 resets the touched flags of the fields in the orders screen record, after the database is successfully updated, to get back to the initial state of the dialog.
- Line 22 displays a message to the user indicating the database update was successful.
- Line 24 returns TRUE to the calling function if the database update was successful.

Function order_new

This function inserts a new row in the database table orders, using the values from the order_rec program record.

Function order_new (orders.4gl)

```
01 FUNCTION order_new()
02   SELECT MAX(order_num)+1 INTO order_rec.order_num
03     FROM orders
04   IF order_rec.order_num IS NULL
05    OR order_rec.order_num == 0 THEN
06     LET order_rec.order_num = 1
07   END IF
08  LET order_total = 0
09   -- We keep the same store...
10   LET order_rec.order_date = TODAY
11   LET order_rec.fac_code = "ASC"
12   LET order_rec.ship_instr = "FEDEX"
13   LET order_rec.promo = "N"
14
15   WHENEVER ERROR CONTINUE
16   INSERT INTO orders (
17      store_num,
18      order_num,
19      order_date,
20      fac_code,
21      ship_instr,
22      promo
23   ) VALUES (
24      order_rec.store_num,
25      order_rec.order_num,
26      order_rec.order_date,
27      order_rec.fac_code,
28      order_rec.ship_instr,
29      order_rec.promo
30   )
31   WHENEVER ERROR STOP
32   IF SQLCA.SQLCODE <> 0 THEN
33      CLEAR FORM
34      CALL __mbox_ok(title1,SQLERRMESSAGE,"stop")
35      RETURN FALSE
36   END IF
```

```
37   CALL arr_ordnums.insertElement(1)
38  LET arr_ordnums[1] = order_rec.order_num
39   CALL arr_items.clear()
40   MESSAGE msg11
41   RETURN TRUE
42 END FUNCTION
```

Note:

- Lines 02 thru 07 add the next unused order number to the order_num field of the order_rec program record, based on the existing order numbers in the orders database table.
- Lines 08 thru 13 set the order total to zero, and add default values to some order_rec fields.
- Lines 15 thru 31 execute the SQL statement to insert a new row in the orders database table using values from the order_rec program record.
- Lines 32 thru 36 clear the form and display an error message if the insert into the database table failed, and return FALSE to the calling function.
- Line 37 inserts a new empty element into the arr_ordnums array at the first position, after the successful insert into the orders table.
- Line 38 sets the value of the new element to the order number of the order_rec program record. The arr_ordnums array keeps track of the order numbers of the orders that were retrieved from the database or newly inserted.
- Line 39 clears the program array for items, preparing for the addition of items for the new order.
- Line 40 displays a message indicating the insert of a new row in the orders database table was successful.
- Line 42 returns TRUE to the calling function, indicating the insert into the orders database table was successful.

Function order_validate

This function validates the entries in the fields of the orders screen record.

Function order_validate (orders.4gl):

```
01 FUNCTION order_validate(d)
02   DEFINE d ui.Dialog
03   IF NOT d.getFieldTouched("orders.*") THEN
04      RETURN TRUE
05   END IF
06   IF d.validate("orders.*") < 0 THEN
07      RETURN FALSE
08   END IF
09   IF NOT order_check_store_num() THEN
10      RETURN FALSE
11   END IF
12   RETURN TRUE
13 END FUNCTION
```

Note:

- Line 01 The dialog object is passed to this function, allowing the use of methods of the DIALOG class.
- Lines 03 thru 05 return TRUE to the calling function if the fields in the orders record have not been touched.
- Lines 06 thru 08 call the validate() method of the dialog object to execute any NOT NULL, REQUIRED, and INCLUDE validation rules defined in the form specification file for the fields in the orders screen record. If this validation fails, FALSE is returned to the calling function.

- Lines 09 thru11 call the order_check_store_num function to verify that the store_num value exists in the customer database table. If this validation fails, FALSE is returned to the calling function.
- Line 12 returns TRUE to the calling function when the validation is successful.

Function order_query

This function allows the user to search for a specific order by entering criteria into the form (Query by Example). This CONSTRUCT statement is not a sub-dialog of a DIALOG statement. It is a stand-alone statement called by the action find, triggered when the user selects the corresponding menu item or toolbar icon on the form orderform.

Function order_query (orders.4gl):

```
01 FUNCTION order_query()
02   DEFINE where_clause STRING,
03          id INTEGER, name STRING
04
05   MESSAGE msg02
06   CLEAR FORM
07
08   WHILE TRUE
09     LET int_flag = FALSE
10     CONSTRUCT BY NAME where_clause ON
11       orders.store_num,
12       customer.store_name,
13       orders.order_num,
14       orders.order_date,
15       orders.fac_code
16
17       ON ACTION zoom1
18         CALL display_custlist() RETURNING id, name
19         IF id > 0 THEN
20           DISPLAY id TO orders.store_num
21           DISPLAY name TO customer.store_name
22         END IF
23
24       ON ACTION about
25         CALL __mbox_ok(title1,msg18,"information")
26
27     END CONSTRUCT
28
29     IF int_flag THEN
30       MESSAGE msg03
31       IF arr_ordnums.getLength()==0 THEN
32         IF __mbox_yn(title1,msg15,"stop") THEN
33           EXIT PROGRAM
34         END IF
35         CONTINUE WHILE
36       END IF
37       RETURN
38     ELSE
39       IF order_select(where_clause) THEN
40         EXIT WHILE
41       END IF
42     END IF
43   END WHILE
44
45 END FUNCTION
```

Note:

- Line 02 defines a STRING variable, where_clause, to hold the WHERE clause created from the criteria entered in the form fields by the user.
- Line03 defines an integer variable, id, to hold the store number selected by the user after triggering the display_custlist function of the custlist.4gl module.
- Line 05 displays a message instructing the user to enter search criteria.
- Lines 08 thru 43 contain the WHILE statement that is executed until an order is successfully selected or the user cancels the operation.
- Lines 10 thru 15 specify the form fields that will contain the search criteria for the CONSTRUCT statement.
- Lines 11 thru 22 define an ON ACTION clause for the zoom1 button in the orderform form specification file. After the user selects the desired customer from the customer list that is displayed, the customer number and name are stored in the corresponding fields of orderform.
- Lines 24 thru25 display the message when the user selects the about menu item on the orderform form.
- Lines 29 thru 42 test whether the user wants to interrupt the dialog and responds accordingly.
- Lines 31 thru 37 When the user interrupts, a message box is displayed if the arr_ordnums array is empty, allowing the user to exit the program, or to continue. If the array is not empty, the function simply returns.
- Lines 39 thru 42 when the user has not interrupted, the order_select function is called to retrieve the order information; then the WHILE loop is exited.