

IBM Informix ESQL/C

Programmer's Manual

Version 5.2
November 2002
Part No. 000-9144

Note:
Before using this information and the product it supports, read the information in the appendix entitled "Notices."

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2002. All rights reserved.

US Government User Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Preface

The *INFORMIX-ESQL/C Programmer's Manual* is a complete guide to the features that make up the Informix implementation of embedded SQL (Structured Query Language) for C. It assumes that you know C programming and are familiar with the structure of relational databases. In addition, a knowledge of SQL would be useful. Informix SQL is described in detail in *The Informix Guide to SQL: Tutorial* and *The Informix Guide to SQL: Reference*.

The *INFORMIX-ESQL/C Programmer's Manual* explains how to use **INFORMIX-ESQL/C** to access a database server and describes the special features that the product offers. It is designed specifically as a programmer's manual and progresses from general topics to more advanced programming techniques and examples.

Summary of Chapters

The *INFORMIX-ESQL/C Programmer's Manual* includes the following chapters:

- This Preface provides general information about the manual and lists additional reference materials that will help you understand **ESQL/C** concepts.
- The Introduction tells how **ESQL/C** fits into the Informix family of products and manuals, explains how to use the manual, introduces the demonstration database from which the product examples are drawn, describes the **Informix Messages and Corrections** product, and lists the new features for Version 5.0 of Informix server products.
- Chapter 1, "Programming with **INFORMIX-ESQL/C**," provides the basic information that you need to program in **ESQL/C**. This includes how to use the libraries provided with the product, how to embed SQL statements in

your programs, how to use host variables, and how to implement simple error checking.

- Chapter 2, “INFORMIX-ESQL/C Data Types,” describes the various data types that you can use in an SQL database, their corresponding host variables, and the conversion routines that you use with them.
- Chapter 3, “Working with Character and String Data Types,” describes the details of using the available character and string data types, including the VARCHAR data type. It also lists the routines that you can use with string and character data types.
- Chapter 4, “Working with the DECIMAL Data Type,” describes how to use the DECIMAL data type and the routines that are available with DECIMAL values.
- Chapter 5, “Working with Time Data Types,” contains detailed descriptions of the library functions that permit the manipulation of DATE, DATETIME, and INTERVAL data types.
- Chapter 6, “Working with Binary Large Objects,” describes binary large objects (blobs) and their data types (TEXT and BYTE). It also discusses the routines provided for creating and manipulating blobs.
- Chapter 7, “Error Handling,” explains how to use error checking effectively in your INFORMIX-ESQL/C programs.
- Chapter 8, “Working with the Database Server,” describes miscellaneous run-time routines that you can use to interact with the database server and the operating system.
- Chapter 9, “Dynamic Management in INFORMIX-ESQL/C,” describes when and how to use dynamic memory management in ESQL/C. This includes how to write code using SQL statements that either do and do not receive values at run time and how to use the `sqllda` structure in conjunction with statements that do not receive values at run time.
- Chapter 10, “List of INFORMIX-ESQL/C Routines,” lists the ESQL/C library routines described throughout the manual. The list includes page references and descriptions for all routines.

Related Reading

If you have had no prior experience with database management, you may want to refer to an introductory text like C. J. Date's *Database: A Primer* (Addison-Wesley Publishing, 1983). If you want more technical information on database management, consider consulting the following texts, also by C. J. Date:

- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

This guide assumes you are familiar with your computer operating system. If you have limited UNIX system experience, you may want to look at your operating system manual or a good introductory text before starting to learn about **INFORMIX-ESQL/C**. You might also want to keep your compiler documentation at hand.

Some suggested texts about UNIX systems follow:

- *A Practical Guide to the UNIX System* by M. Sobell (Benjamin/Cummings Publishing, 1984)
- *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985)
- *UNIX for People* by Birns, Brown, and Muster (Prentice-Hall, 1985)



Table of Contents

Introduction

INFORMIX-ESQL/C and Other Informix Products	3
Other Useful Documentation	4
How to Use This Manual	4
Typographical Conventions	5
Command-Line Conventions	5
Useful On-Line Files	7
ASCII and PostScript Error Message Files	8
Using the ASCII Error Message File	8
Using the PostScript Error Message Files	10
The Demonstration Database	11
Creating the Demonstration Database on INFORMIX-OnLine	12
Creating the Demonstration Database on INFORMIX-SE	12
Compliance with Industry Standards	13
New Features in INFORMIX-ESQL/C, Version 5.0	14
Dynamic SQL in X/Open Mode	14
New Routines	15
New Compile Options	15
New Compile Options for Backward Compatibility	15
Other New Features	16
New Features in Informix Server Products, Version 5.0	16

Chapter 1

Programming with INFORMIX-ESQL/C

Chapter Overview	1-3
What Is INFORMIX-ESQL/C?	1-3
Embedding SQL Statements in C Routines	1-4
Case Sensitivity in ESQL/C Files	1-4
Inserting Comments	1-5

Header Files	1-5
ESQL/C Preprocessor Support	1-7
Include Files	1-7
The <i>\$define</i> and <i>\$undef</i> Statements	1-8
The <i>ifdef</i> , <i>ifndef</i> , <i>else</i> , <i>elif</i> , and <i>endif</i> Statements	1-9
Using Host Variables in SQL Statements	1-9
Declaring Host Variables	1-10
Scope of Host Variables	1-11
Types of Host Variables	1-12
Arrays of Host Variables	1-14
Structures as Host Variables	1-14
<i>typedef</i> Expressions as Host Variables	1-15
Null Values in Host Variables	1-15
Character Pointers as Host Variables	1-16
Host Variables as Function Parameters	1-16
Indicator Variables	1-17
Declaring Indicator Variables	1-17
Values Returned in Indicator Variables	1-18
Using Indicator Variables	1-18
Compiling INFORMIX-ESQL/C Programs	1-20
Syntax of the <i>esql</i> Command	1-21
Preprocessing Without Compiling or Linking	1-22
Using the Preprocessing Options	1-23
Preprocessing, Compiling, and Linking with the <i>esql</i> Command	1-26
A Sample INFORMIX-ESQL/C Program	1-28
Guide to <i>demo1.ec</i>	1-28

Chapter 2

INFORMIX-ESQL/C Data Types

Chapter Overview	2-3
Choosing Data Types for Host Variables	2-3
Defined Integers for Data Types	2-5
Character Data Type Choices	2-5
Data Conversion	2-6
When Conversion Occurs	2-6
What Happens in a Conversion	2-7
Numbers to Strings	2-8
Numbers to Numbers	2-8
Operations on Numeric Values	2-8
Data Conversion When Fetching Rows	2-9
Converting Between DATETIME and DATE Data Types	2-9
Converting Between VARCHAR and Character Data Types	2-10

Data Type Function Descriptions	2-12
RISNULL	2-13
RSETNULL	2-16
RTYPALIGN	2-19
RTYPMSIZE	2-22
RTYPNAME	2-25
RTYPWIDTH	2-28
Numeric-Formatting Routines	2-31
Formatting Numeric Strings	2-31
Example Format String	2-33
Example Format String	2-34
Example Format String	2-35
Example Format String	2-36
RFMTDOUBLE	2-37
RFMTLONG	2-40

Chapter 3

Working with Character and String Data Types

Chapter Overview	3-3
Character and String Functions	3-4
BYCMPR	3-5
BYCOPY	3-7
BYFILL	3-9
BYLENG	3-11
LDCHAR	3-13
RDOWNSHIFT	3-15
RSTOD	3-16
RSTOI	3-18
RSTOL	3-20
RUPSHIFT	3-22
STCAT	3-23
STCHAR	3-25
STCMPR	3-27
STCOPY	3-29
STLENG	3-30
Programming with a VARCHAR Data Type	3-32
Declaring a Host Variable for a VARCHAR Data Type	3-32
VARCHAR Macros	3-33

Chapter 4	Working with the DECIMAL Data Type
	Chapter Overview 4-3
	The DECIMAL Data Type 4-3
	Decimal Type Functions 4-5
	DECCVASC 4-6
	DECTOASC 4-9
	DECCVINT 4-12
	DECTOINT 4-14
	DECCVLONG 4-16
	DECTOLONG 4-18
	DECCVDBL 4-20
	DECTODBL 4-22
	DECADD, DECSUB, DECMUL, and DECDIV 4-25
	DECCMP 4-31
	DECCOPY 4-33
	DECECVT and DECFCVT 4-35
	DECROUND 4-41
	DECTRUNC 4-43
	RFMTDEC 4-45
Chapter 5	Working with Time Data Types
	Chapter Overview 5-3
	The DATE Data Type 5-3
	DATE Functions 5-4
	RDATESTR 5-5
	RDAYOFWEEK 5-7
	RDEFMTDATE 5-9
	RFMTDATE 5-12
	RJULMDY 5-15
	RLEAPYEAR 5-17
	RMDYJUL 5-19
	RSTRDATE 5-21
	RTODAY 5-23
	DATETIME and INTERVAL Data Types 5-24
	DATETIME and INTERVAL Columns 5-24
	Declaring DATETIME and INTERVAL Host Variables 5-25
	Fetching DATETIME and INTERVAL Values 5-26
	Storing DATETIME and INTERVAL Values 5-27
	Converting Between DATETIME and DATE Data Types 5-27
	DATETIME and INTERVAL Data Type Functions 5-28
	DTCURRENT 5-30
	DTCVASC 5-32
	DTCVFMTASC 5-35

DTEXTEND 5-37
DTTOASC 5-40
DTTOFMTASC 5-42
INCVASC 5-44
INCVFMTASC 5-46
INTOASC 5-48
INTOFMTASC 5-50

Chapter 6

Working with Binary Large Objects

Chapter Overview 6-3
Programming with Blobs 6-3
 Fields Common to All Data Locations 6-5
 Locating Blobs in Memory 6-6
 Reading a Blob into Memory 6-7
 Writing a Blob from Memory 6-9
 Locating Blobs in Open Files 6-10
 Locating Blobs in Named Files 6-14
 User-Programmed Location 6-17
 LOC_DESCRIPTOR 6-19
Guide to *dispcat_pic* 6-22
 Before Using *dispcat_pic* 6-22
 Using the Conditional Display Logic 6-23
 Loading the *cat_picture* Column 6-23
 Using *blobload* 6-24
 The *dispcat_pic* Program 6-26

Chapter 7

Error Handling

Chapter Overview 7-3
 The Role of the *sqlca* Structure 7-3
General Error Handling 7-5
 Error Status < 0 7-5
 Error Status = 0 7-5
 Error Status > 0 and < 100 7-5
 Error Status = SQLNOTFOUND or 100 7-6
 Using the SQLCODE Variable 7-7
 Checking for an Error Using In-Line Code 7-8
 Automatically Checking for Errors with the WHENEVER Statement 7-9
 Checking for Warnings 7-10
 Errors After a PREPARE Statement 7-12
 Errors After an EXECUTE Statement 7-12
 RGETMSG 7-13
A Program That Uses Full Error Checking 7-15

Chapter 8	Working with the Database Server
	Chapter Overview 8-3
	Database Server Control Functions 8-3
	SQLBREAK 8-4
	SQLDETACH 8-5
	SQLEXIT 8-6
	SQLSTART 8-7
Chapter 9	Dynamic Management in INFORMIX-ESQL/C
	Chapter Overview 9-3
	Dynamic SQL Statements and Management Techniques 9-4
	Types of Dynamic Management Situations 9-4
	The System Descriptor Area 9-5
	The <i>sqlda</i> Structure 9-6
	Constants in <i>sqltypes.h</i> 9-7
	Constants and <i>sqlstype.h</i> 9-9
	Non-SELECT Statements That Do Not Receive Values at Run Time 9-12
	Using EXECUTE IMMEDIATE 9-12
	SELECT Statements in Which Select-List Values Are Determined at Run Time 9-13
	Using Descriptors 9-13
	Using an <i>sqlda</i> Structure 9-17
	SELECT Statements That Receive WHERE-Clause Values at Run Time 9-21
	Using Host Variables 9-21
	Using a System Descriptor Area 9-24
	Using an <i>sqlda</i> Structure 9-29
	Non-SELECT Statements That Receive Values at Run Time 9-30
	Using Host Variables 9-31
	Using a System Descriptor Area 9-31
	Using an <i>sqlda</i> Structure 9-33
Chapter 10	List of INFORMIX-ESQL/C Routines
	List of Routines 10-3
Appendix A	Notices

Introduction

INFORMIX-ESQL/C and Other Informix Products	3
Other Useful Documentation	4
How to Use This Manual	4
Typographical Conventions	5
Command-Line Conventions	5
Useful On-Line Files	7
ASCII and PostScript Error Message Files	8
Using the ASCII Error Message File	8
The finderr Script	9
The rofferr Script	9
Using the PostScript Error Message Files	10
The Demonstration Database	11
Creating the Demonstration Database on INFORMIX-OnLine	12
Creating the Demonstration Database on INFORMIX-SE	12
Compliance with Industry Standards	13
New Features in INFORMIX-ESQL/C, Version 5.0	14
Dynamic SQL in X/Open Mode	14
New Routines	15
New Compile Options	15
New Compile Options for Backward Compatibility	15
Other New Features	16
New Features in Informix Server Products, Version 5.0	16



INFORMIX-ESQL/C is an application development tool that is designed for the C programmer who wants to create custom C applications with database management capabilities. **INFORMIX-ESQL/C** allows you to use a third-generation language with which you are familiar and takes advantage of the Informix enhanced Structured Query Language (SQL).

You use the libraries, header files, and preprocessors that are provided with **INFORMIX-ESQL/C** to embed database management instructions, in the form of concise SQL statements, directly into a C program. This product also provides you with an extensive library of routines to manipulate and use SQL data types. It also provides routines that you can use to interpret messages sent by the operating system. The preprocessor allows you to preprocess embedded C code directly into C executable code, or you can use it to create C source code for your inspection or manipulation.

INFORMIX-ESQL/C and Other Informix Products

INFORMIX-ESQL/C is one of many application development tools and CASE tools produced by Informix Software, Inc. Other tools currently available include **INFORMIX-4GL** and the **Interactive Debugger** and other Informix embedded-language products, such as **INFORMIX-ESQL/COBOL** and **INFORMIX-ESQL/FORTRAN**.

INFORMIX-ESQL/C works with a database server, either **INFORMIX-OnLine** or **INFORMIX-SE**. If you are developing or running applications on a network, you are using an Informix client/server product such as **INFORMIX-NET** or **INFORMIX-STAR**. **INFORMIX-NET** is the communication facility for multiple **INFORMIX-SE** database servers. **INFORMIX-STAR** allows distributed database access to multiple **INFORMIX-OnLine** database servers.

Other Useful Documentation

You may want to refer to a number of related Informix product documents that complement the *INFORMIX-ESQL/C Programmer's Manual*.

- If you have never used SQL (Structured Query Language) or an Informix application development tool, read *The Informix Guide to SQL: Tutorial* to learn basic database design and implementation concepts.
- A companion volume to the Tutorial, *The Informix Guide to SQL: Reference*, provides full information on the structure and contents of the demonstration database that is provided with **INFORMIX-ESQL/C**. It includes details of the Informix system catalog tables, describes Informix and common UNIX environment variables that should be set, and defines column data types supported by Informix products. Further, it provides a detailed description of all the SQL statements supported by Informix products. It also contains a glossary of useful terms.
- The *SQL Quick Syntax Guide* contains syntax diagrams for all of the statements and segments described in *The Informix Guide to SQL: Reference*.
- You, or whoever installs **INFORMIX-ESQL/C**, should refer to the *UNIX Products Installation Guide* for your particular version to ensure that **INFORMIX-ESQL/C** is properly set up before you begin to work with it.
- If you are using **INFORMIX-ESQL/C** across a network, you may also want to refer to the *INFORMIX-NET/INFORMIX-STAR Installation and Configuration Guide*.
- Depending on the database server you are using, you or your system administrator need either the *INFORMIX-OnLine Administrator's Guide* or the *INFORMIX-SE Administrator's Guide*.
- When errors occur, you can look them up, by number, and find their cause and solution in the *Informix Error Messages* manual. If you prefer, you can look up the error messages in the on-line message file described in the section "ASCII and PostScript Error Message Files" later in this Introduction.

How to Use This Manual

This manual assumes that you are using **INFORMIX-OnLine** as your database server. Features and behavior specific to **INFORMIX-SE** are noted throughout the manual.

Typographical Conventions

The *INFORMIX-ESQL/C Programmer's Manual* uses a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout the manual:

<i>italics</i>	When new terms are introduced, they are printed in italics.
boldface	Database names, table names, column names, filenames, utilities, and other similar terms are printed in boldface.
computer	Information that INFORMIX-ESQL/C displays and information that you enter are printed in a computer typeface.
KEYWORD	All keywords appear in uppercase letters.

Additionally, when you are instructed to “enter” or “execute” text, immediately press RETURN after the entry. When you are instructed to “type” the text, no RETURN is required.

Command-Line Conventions

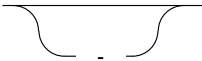


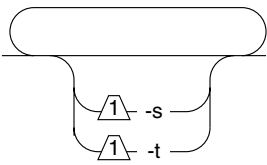
INFORMIX-ESQL/C supports a variety of command-line options. These are commands that you enter at the operating system prompt to perform certain functions in **INFORMIX-ESQL/C**. Each valid command-line option is illustrated in a diagram in Chapter 1 of this manual.

This section defines and illustrates the format of the commands available in **INFORMIX-ESQL/C** and other Informix products. These commands have their own conventions, which may include alternative forms of a command, required and optional parts of the command, and so forth.

Each diagram displays the sequences of required and optional elements that are valid in a command. A diagram begins at the upper left with a command. It ends at the upper right with a vertical line. Between these points, you can trace any path that does not stop or back up. Each path describes a valid form of the command. You must supply a value for words that are in italics.

Along a command-line path, you may encounter the following elements:

<i>command</i>	This required element is usually the product name or other short word used to invoke the product or call the compiler or preprocessor script for a compiled Informix product. It may appear alone or precede one or more options. You must
----------------	--

	spell a command exactly as shown and must use lowercase letters.
<i>variable</i>	A word in italics represents a value that you must supply, such as a database, file, or program name. The nature of the value is explained immediately following the diagram.
<i>-flag</i>	A flag is usually an abbreviation for a function, menu, or option name or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.
<i>.ext</i>	A filename extension, such as .sql or .cob , may follow a variable representing a filename. Type this extension exactly as shown, immediately after the name of the file and a period. The extension may be optional in certain products.
(.,;+*-/)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
" "	Double quotes are literal symbols that you must enter as shown. You can replace a pair of double quotes with a pair of single quotes, if you prefer. You cannot mix double and single quotes.
<div style="border: 1px solid black; padding: 2px; display: inline-block;">Privileges p. 6-17</div>	A reference in a box represents a subdiagram on the same page or another page. Imagine that the subdiagram is spliced into the main diagram at this point.
— ALL —	A shaded option is the default. Even if you do not explicitly type the option, it will be in effect unless you choose another option.
	A branch below the main line indicates an optional path.
	The vertical line is a terminator and indicates that the statement is complete.
	Commands enclosed in a pair of arrows indicate that this is a subdiagram.
	A gate (\square) in an option indicates that you can only use that option once, even though it is within a larger loop.

The following diagram shows the elements of a fictional command used to echo file input to the screen:

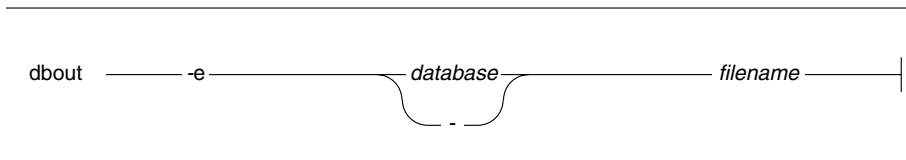


Figure 1 Elements of a command-line diagram

To construct a similar command, start at the top left with the command `dbout`. Then follow the diagram to the right, including the elements that you want. This diagram conveys the following information:

1. You must type the word `dbout`.
2. You can echo the SQL statements in a command file to the screen by typing the flag `-e` before the database name.
3. You must supply a *database* name or use a hyphen (`-`) to indicate that a database name is specified in the command file that you want to run.
4. You must specify the *filename* of a command file whose SQL statements you want to echo to the screen.

On some command-line diagrams, you can take the direct route to the terminator, or you can take an optional path indicated by a branch below the main line.

Once you are back at the main diagram, you come to the terminator. Your `dbout` command is complete. Press RETURN to execute the command.

Useful On-Line Files

In addition to the Informix set of manuals, the following on-line files, located in the `$INFORMIXDIR/release` directory, may supplement the information in the *INFORMIX-ESQL/C Programmer's Manual*:

- | | |
|----------------------------|--|
| Documentation Notes | describe features not covered in the manual or which have been modified since publication. The file containing the Documentation Notes for this product is called ESQLDOC_5.0 . |
| Release Notes | describe feature differences from earlier versions of Informix products and how these differences may affect current products. The file containing the Release Notes for INFORMIX-ESQL/C and other products is called ENGREL_5.0 . |

Machine Notes describe any special actions required to configure and use Informix products on your machine. The file containing the Machine Notes for **INFORMIX-ESQL/C** is called **ESQLC_5.0**.

Please examine these files because they contain vital information about application and performance issues.

This manual makes extensive references to sample programs. These programs are provided on-line as part of the product.

ASCII and PostScript Error Message Files

Informix software products provide ASCII files that contain all the Informix error messages and their corrective actions. To access the error messages in the ASCII file, Informix provides scripts that let you display error messages on the terminal or print formatted error messages.

The optional **Informix Messages and Corrections** product provides PostScript files that contain the error messages and their corrective actions. If you have installed this product, you can print the PostScript files on a PostScript printer.

Using the ASCII Error Message File

You can use the file that contains the ASCII text version of the error messages and their corrective actions in two ways:

- Use the **finderr** script to display one or more error messages on the terminal screen.
- Use the **rofferr** script to print one error message or a range of error messages.

The scripts are in the **\$INFORMIXDIR/bin** directory. The ASCII file has the following path:

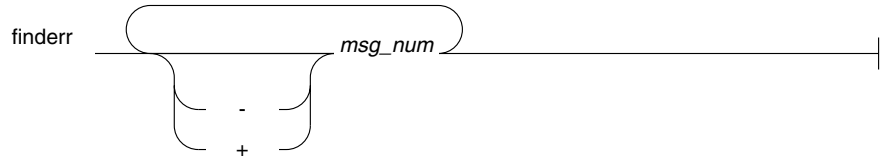
```
$INFORMIXDIR/msg/errmsg.txt
```

The error message numbers range from -1 to -33000. When you specify these numbers for the **finderr** or **rofferr** scripts, you can omit the minus sign. A few messages have positive numbers. In the unlikely event that you want to display them, you must precede the message number with a + sign.

The messages numbered -1 to -100 can be platform-dependent. If the message text for a message in this range does not apply to your platform, check the operating system documentation for the precise meaning of the message number.

The *finderr* Script

Use the **finderr** script to display one or more error messages, and their corrective actions, on the terminal screen. The **finderr** script has the following syntax:



msg_num is the number of the error message to display.

You can specify any number of error messages per **finderr** command. The **finderr** command copies all the specified messages, and their corrective actions, to standard output.

For example, to display the -359 error message, you can enter the following command:

```
finderr -359
```

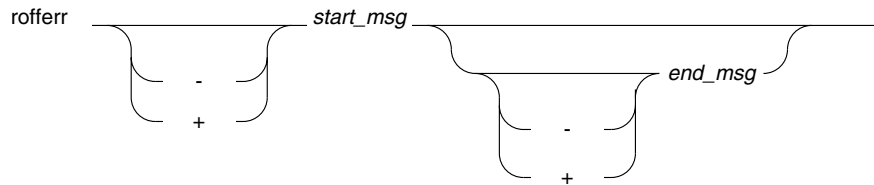
The following example demonstrates how to specify a list of error messages. This example also pipes the output to the UNIX **more** command to control the display. You also can redirect the output to another file so that you can save or print the error messages:

```
finderr 233 107 113 134 143 144 154 | more
```

The *rofferr* Script

Use the **rofferr** script to format one error message, or a range of error messages, for printing. By default, **rofferr** displays output on the screen. You need to send the output to **nroff** to interpret the formatting commands and then to a printer, or to a file where the **nroff** output is stored until you are ready to print. You can then print the file. For information on using **nroff** and on printing files, see your UNIX documentation.

The **rofferr** script has the following syntax:



start_msg is the number of the first error message to format. This error message number is required.

end_msg is the number of the last error message to format. This error message number is optional. If you omit *end_msg*, only *start_msg* is formatted.

The following example formats error message -359. It pipes the formatted error message into **nroff** and sends the output of **nroff** to the default printer:

```
rofferr 359 | nroff -man | lpr
```

The following example formats and then prints all the error messages between -1300 and -4999:

```
rofferr -1300 -4999 | nroff -man | lpr
```

Using the PostScript Error Message Files

Use the **Informix Messages and Corrections** product to print the error messages, and their corrective actions, on a PostScript printer. The PostScript error messages are distributed in a number of files of the format **errmsg1.ps**, **errmsg2.ps**, and so on. These files are located in the **\$INFORMIXDIR/msg** directory. Each file contains approximately 50 printed pages of error messages.

The Demonstration Database

Your Informix software includes a demonstration database called **stores5** that contains information about a fictitious wholesale sporting-goods distributor. The source files that make up a demonstration application are included as well.

Most of the examples in this manual are based on the **stores5** demonstration database. The **stores5** database is described in detail and its contents are listed in Chapter 1 of *The Informix Guide to SQL: Reference*.

The script you use to install the demonstration database is called **esqldemo5** and is located in the **\$INFORMIXDIR/bin** directory. The database name that you supply is the name given to the demonstration database. If you do not supply a database name, the name defaults to **stores5**. Follow these rules for naming your database:

- Names for databases can be up to 10 characters long.
- The first character of a name must be a letter.
- You can use letters, characters, and underscores (`_`) for the rest of the name.
- **INFORMIX-ESQL/C** makes no distinction between uppercase and lowercase letters.
- The database name should be unique.

When you run **esqldemo5**, you are, as the creator of the database, the owner and Database Administrator (DBA) of that database.

If you installed your Informix database server product according to the installation instructions, the files that make up the demonstration database are protected so that you cannot make any changes to the original database.

You can run the **esqldemo5** script again whenever you want to work with a fresh demonstration database. The script prompts you when the creation of the database is complete, and asks if you would like to copy the demonstration programs to the current directory. Answer “N” to the prompt if you have made changes to the demonstration programs and do not want them replaced with the original versions. Answer “Y” to the prompt if you want to copy over the demonstration programs.

Creating the Demonstration Database on INFORMIX-OnLine

Use the following steps to create and populate the demonstration database in the **INFORMIX-OnLine** environment:

1. Set the **INFORMIXDIR** environment variable so that it contains the name of the directory in which your Informix products are installed. Set **SQLEXEC** to **\$INFORMIXDIR/lib/sqlturbo**. (For a full description of environment variables, see Chapter 4 of *The Informix Guide to SQL: Reference*.)

2. Create a new directory for the SQL demonstration programs. Create the directory by entering

```
mkdir dirname
```

3. Make the new directory the current directory by entering

```
cd dirname
```

4. Create the demonstration database and copy over the demonstration programs by entering

```
esqldemo5 dbname
```

The data for the database is put into the root dbspace.

To give someone else the SQL privileges to access the data, use the **GRANT** and **REVOKE** statements. The **GRANT** and **REVOKE** statements are described in Chapter 7 of *The Informix Guide to SQL: Reference*.

To use the demonstration programs that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **esqldemo5** script. To give someone else the permissions to access the demonstration programs in your directory, use the UNIX **chmod** command.

Creating the Demonstration Database on INFORMIX-SE

Use the following steps to create and populate the demonstration database in the **INFORMIX-SE** environment:

1. Set the **INFORMIXDIR** environment variable so that it contains the name of the directory in which your Informix products are installed. Set

SQLEXEC to **\$INFORMIXDIR/lib/sqlexec**. (For a full description of environment variables, see Chapter 4 of *The Informix Guide to SQL: Reference*.)

2. Create a new directory for the demonstration database. This directory will contain the demonstration programs included with the demonstration database. Create the directory by entering

```
mkdir dirname
```

3. Make the new directory the current directory by entering

```
cd dirname
```

4. Create the demonstration database and copy over the demonstration programs by entering

```
esqldemo5 dbname
```

When you run the **esqldemo5** script, it creates a subdirectory called ***dbname.dbs*** in your current directory and places the database files associated with **stores5** there. You will see both data and index files in the ***dbname.dbs*** directory.

To use the database and the demonstration programs that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **esqldemo5** script. To give someone else the permissions to access the demonstration programs in your directory, use the UNIX **chmod** command. Check with your system administrator for more information about operating system file and directory permissions. UNIX permissions are discussed in the *INFORMIX-SE Administrator's Guide*.

To give someone else access to the database that you have created, grant them the appropriate privileges using the SQL GRANT statement. To remove privileges, use the REVOKE statement. The GRANT and REVOKE statements are described in Chapter 7 of *The Informix Guide to SQL: Reference*.

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are compliant with ANSI Level 2 (published as ANSI X3.135-1989) on the **INFORMIX-OnLine** database server. They are compliant with ANSI Level 2 on the **INFORMIX-SE** database server with the following exceptions:

- Effective checking of constraints
- Serializable transactions

INFORMIX-TP/XA conforms to the *X/Open Preliminary Specification (April 1990), Distributed Transaction Processing: The XA Interface*.

New Features in INFORMIX-ESQL/C, Version 5.0

The following features have been added or modified for Version 5.0 of INFORMIX-ESQL/C:

Dynamic SQL in X/Open Mode

INFORMIX-ESQL/C has been modified to support Dynamic SQL in X/Open mode. Prior to this version, if you used dynamic SQL, you used the DESCRIBE statement to set an *sqlda-ptr* to an **sqlda** structure and describe the data retrieved when the described statement was executed. For Dynamic SQL in X/Open mode, you can allocate a system descriptor area and specify its size. You can retrieve the information stored in these system descriptor areas by using new SQL statements.

The modifications include changes to the following SQL statements:

- DESCRIBE
- EXECUTE
- FETCH
- OPEN
- PUT

In addition to the modifications to existing statements, four new statements have been added to support Dynamic SQL in X/Open mode. These statements are as follows:

- ALLOCATE DESCRIPTOR
- DEALLOCATE DESCRIPTOR
- GET DESCRIPTOR
- SET DESCRIPTOR

How and when to use Dynamic SQL in X/Open mode is discussed in Chapter 9 of this manual.

New Routines

Four new routines have been added to **INFORMIX-ESQL/C** to support **DATETIME** and **INTERVAL** data types. See Chapter 5 of this manual for information on these new routines. The names of the new routines are as follows:

- **dtcvfmtasc()** converts an ASCII string to a **DATETIME** value using a specified format string.
- **dttofmtasc()** converts a **DATETIME** value to an ASCII string value using a specified format string.
- **incvfmtasc()** converts an ASCII string to an **INTERVAL** value using a specified format string.
- **intofmtasc()** converts an **INTERVAL** value to an ASCII string value using a specified format string.

New Compile Options

New options have been added to the **esql** command. The new options and the full syntax of **esql** are discussed in Chapter 1.

- A new preprocessor option, **-log filename**, lets you send error messages and warnings to a specified file instead of to standard output.
- A new compiler flag, **-xopen**, can be passed to the compiler script to invoke the use of the X/Open data type codes.

New Compile Options for Backward Compatibility

New options have been added to the **esql** command for backward compatibility with applications created in previous versions of **INFORMIX-ESQL/C**. Do not use these options when compiling new applications. The new options and the full syntax of **esql** are discussed in Chapter 1.

- Cursor names and statement identifier names are global in scope. A new compiler flag, **-local**, lets you keep these cursors and statement identifiers local to the file in which they are defined.
- Statement ids (created with a **PREPARE** statement) and cursor names (created with a **DECLARE** statement) are not case sensitive, by default. If you want the **INFORMIX-ESQL/C** preprocessor to be case sensitive with respect to cursor names and statement ids, you can use the **-cs** option to the **esql** command when you preprocess your program.

In Version 4.1 of **INFORMIX-ESQL/C**, static cursor names and static statement ids were case sensitive by default. If you used uppercase and low-

uppercase letters to differentiate cursor names or statement ids, you must use the `-cs` option or rewrite your programs.

Other New Features

- You can declare multiple cursors using a single statement identifier. This feature is described in Chapter 9 of this manual and in the discussion of the DECLARE statement in Chapter 7 of *The Informix Guide to SQL: Reference*.
- A new preprocessor instruction, ELIF, provides an alternative to the macro statement defined by an IFDEF condition. This instruction is discussed in Chapter 1.

New Features in Informix Server Products, Version 5.0

This section highlights the major new features implemented in Version 5.0 of Informix server products:

- **Referential and Entity Integrity**
New data integrity constraints allow you to specify a column or columns as representing a *primary* or *foreign key* of a table upon creation, and to establish dependencies between tables. Once specified, a parent-child relationship between two tables is enforced by the database server. Other constraints allow you to specify a default value for a column, or to specify a condition for a column that an inserted value must meet.
- **Stored Procedures**
A stored procedure is a function written by a user using a combination of SQL statements and Stored Procedure Language (SPL). Once created, a procedure is stored as an object in the database in a compiled, optimized form, and is available to other users with the appropriate privileges. In a client/server environment, the use of stored procedures can significantly reduce network traffic.
- **Dynamic SQL**
Support is provided for the X/Open implementation of dynamic SQL using a system descriptor area. This support involves the new SQL statements ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, and SET DESCRIPTOR, as well as changes in the syntax of existing dynamic management statements.

- **Optimizer Enhancement**

You can use the new SET OPTIMIZATION statement to instruct the database server to select a high or low level of query optimization. The default level of HIGH causes the database server to examine and select the best of all possible optimization strategies. Since this level of optimization may result in a longer-than-desired optimization time for some queries, you have the option of setting an optimization level of LOW.
- **Relay Module (INFORMIX-NET only)**

The new Relay Module component of INFORMIX-NET resides on the client machine in a distributed data processing environment and *relays* messages between the application development tool and an **INFORMIX-OnLine** or **INFORMIX-SE** database server through a network interface. The Relay Module allows Version 5.0 application development tools to connect to a remote database server without the need to run an Informix database server process on the client.
- **Fast Indexing (INFORMIX-OnLine only)**

The Version 5.0 **INFORMIX-OnLine** database server uses a new method of creating large indexes when you execute the CREATE INDEX statement. In this method, index entries are sorted prior to their insertion into the B+ tree structure, resulting in faster index creation.
- **Two-Phase Commit (INFORMIX-STAR only)**

The new two-phase commit protocol allows you to manipulate data in multiple databases on multiple **OnLine** database servers within a single transaction. It ensures that transactions that span more than one **OnLine** database server are committed on an all-or-nothing basis.
- **Support for Optical Media (INFORMIX-OnLine/Optical only)**

A new product, **INFORMIX-OnLine/Optical**, allows you to store and retrieve blob data using an optical storage subsystem. The new SQL statements and functions that support optical storage are described in the *INFORMIX-OnLine/Optical User Manual*.
- **Support for Transaction Processing in the XA Environment (INFORMIX-TP/XA only)**

A new product, **INFORMIX-TP/XA**, allows you to use the **INFORMIX-OnLine** database server as a Resource Manager in conformance with the *X/Open Preliminary Specification (April 1990), Distributed Transaction Processing: The XA Interface*. The *INFORMIX-TP/XA User Manual* describes the changes in the behavior of existing SQL statements that manage transactions in an X/Open environment.

Programming with INFORMIX- ESQL/C

Chapter Overview	3
What Is INFORMIX-ESQL/C?	3
Embedding SQL Statements in C Routines	4
Case Sensitivity in ESQL/C Files	4
Inserting Comments	5
Header Files	5
ESQL/C Preprocessor Support	7
Include Files	7
The \$define and \$undef Statements	8
The ifdef , ifndef , else , elif , and endif Statements	9
Using Host Variables in SQL Statements	9
Declaring Host Variables	10
Initializing Host Variables	11
Scope of Host Variables	11
Defining a Block	12
Types of Host Variables	12
Arrays of Host Variables	14
Structures as Host Variables	14
typedef Expressions as Host Variables	15
Null Values in Host Variables	15
Character Pointers as Host Variables	16
Host Variables as Function Parameters	16

Indicator Variables	17
Declaring Indicator Variables	17
Values Returned in Indicator Variables	18
Using Indicator Variables	18
Compiling INFORMIX-ESQL/C Programs	20
Syntax of the esql Command	21
Preprocessing Without Compiling or Linking	22
Using the Preprocessing Options	23
Checking for ANSI-Standard Syntax	23
Checking for Missing Indicator Variables	23
Numbering Lines	24
Redirecting Errors and Warnings	24
Defining and undefining values while preprocessing	24
Using X/Open codes and the -xopen option	25
Setting the scope of cursor names and statement IDs	26
Case sensitivity in cursor names and statement IDs	26
Preprocessing, compiling, and linking with the esql command	26
Syntax of the compiling and linking options to the esql command	27
Passing other arguments to the cc compiler	27
Passing other C source and object files to the cc compiler	27
Using other libraries	27
A sample INFORMIX-ESQL/C program	28
Guide to demo1.ec	28

Chapter Overview

C programs that use **INFORMIX-ESQL/C** statements generally include the following elements, each of which is described in this chapter:

- Header files
- Include files
- Host variables
- Indicator variables
- SQL statements

Your program also can include dynamically defined statements, which are described in Chapter 9 of this manual.

This chapter provides details on each of the following topics:

- Using header files in **ESQL/C** programs
- Supplying preprocessor support with **ESQL/C**
- Embedding SQL statements in C programs
- Identifying C variables with database constructs
- Preprocessing and compiling your C program

What Is **INFORMIX-ESQL/C**?

INFORMIX-ESQL/C is an application development tool that enables you to embed SQL statements directly into C code. It consists of a code preprocessor, header files, and libraries of C routines.

You create an **ESQL/C** program by writing a C program, including special header files, and writing SQL statements into the program. You then run the **ESQL/C** preprocessor on your code. The **ESQL/C** preprocessor takes your code, reads all of the embedded SQL statements, and turns the embedded SQL statements into C code. Once the C code is created, it is compiled and linked. The process by which an **ESQL/C** program becomes an executable program is shown in Figure 1-1.

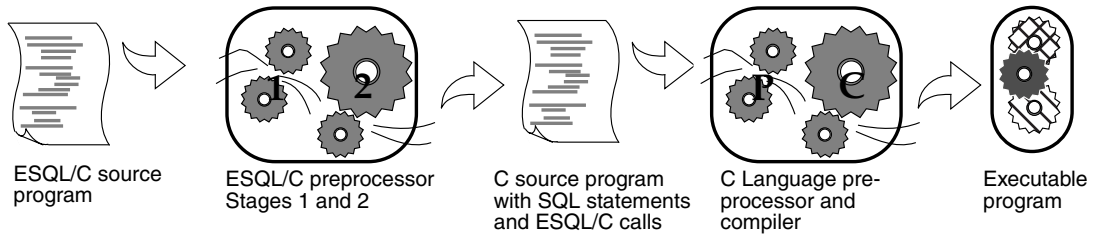


Figure 1-1 Relationship between INFORMIX-ESQL/C and C

Embedding SQL Statements in C Routines

SQL statements are embedded in a C routine using the dollar sign (\$) or the EXEC SQL keywords. Use either of the following conventions to embed an SQL statement in a C routine, replacing *SQL_statement* with the complete text of a valid statement:

```
$SQL_statement;
```

```
EXEC SQL SQL_statement;
```

Note: Use of the EXEC SQL keywords (in place of the dollar sign (\$)) conforms to ANSI standards.

ESQL/C statements can include host variables (and usually indicator variables, as well) in most places a constant can be used. See the syntax of individual statements in *The Informix Guide to SQL: Reference* for any exceptions.

Case Sensitivity in ESQL/C Files

The INFORMIX-ESQL/C preprocessor treats uppercase and lowercase letters as distinct identifiers in variable names.

Statement ids (created with a PREPARE statement) and cursor names (created with a DECLARE statement) are not case sensitive, by default.

The following example shows the creation of statement ids and cursor names. Since the preprocessor is not case sensitive, the following code produces errors.

```
$PREPARE st FROM "select * from tab1";
$PREPARE ST FROM "insert into tab2 values (1,2)";/*duplicate*/
$DECLARE curname CURSOR FOR st;
$DECLARE CURNAME CURSOR FOR ST;                /*duplicate*/
```

*Note: If you want the **ESQL/C** preprocessor to be case sensitive with respect to cursor names and statement ids, you can use the **-cs** option to the **esql** command when you preprocess your program.*

Inserting Comments

You can use a double dash (--) comment indicator on any **INFORMIX-ESQL/C** line (one that has been prefaced by \$ or EXEC SQL and terminated by a semicolon). The comment continues to the end of the line. For example, the following line of code includes a **DATABASE** statement that opens the **stores5** database and a comment to that effect:

```
$database stores5;    -- stores5 database is open now!

printf("\nDatabase opened\n"); /* This isn't an ESQL/C line*/
/* so it needs a regular C notation for a comment*/
$begin work;          /*You can also use a C comment here*/
```

You also can use a standard C comment on an **INFORMIX-ESQL/C** line, as shown in the last line in the preceding example.

Header Files

Several header files are provided with **INFORMIX-ESQL/C**, all of which are located in the **/incl** subdirectory of the **\$INFORMIXDIR** directory. An **INFORMIX-ESQL/C** program can include the following header files:

sqlca.h	contains the structure in which error status codes are stored. This file is automatically included when your program is preprocessed, to allow checking the success or failure of SQL statements.
sqlda.h	contains the structures that contain value pointers and descriptions of dynamically defined variables.

sqlstype.h	contains integer constants corresponding to SQL statements; used with the DESCRIBE statement.
sqltypes.h	contains the definitions of strings corresponding to C language and SQL data types; used with the DESCRIBE statement.
varchar.h	contains macros that you can use with VARCHAR data types.
locator.h	contains the structure in which information about the location of blobs (BYTE and TEXT data types) is stored.
sqlxtype.h	contains the definitions of strings corresponding to C language and SQL data types that are used when in X/Open mode.
decimal.h	contains the definition of the structure in which a DECIMAL data type is stored.
datetime.h	contains the definition of the structures in which DATETIME and INTERVAL data types are stored.

Use a dollar sign (\$) and the word “include” to add these files to your **ESQL/C** code. Examples of including these files follow:

```
$include sqlca;  
$include sqllda;  
$include sqlstype;
```

The contents of a file that you include using the preceding syntax is placed into the C code when you run the **ESQL/C** preprocessor on the **ESQL/C** code.

The **sqlca.h** file is automatically included by the preprocessor so that you can check the success or failure of your **ESQL/C** statements. You need not include any of the other header files (**sqllda.h**, **sqlstype.h**, **varchar.h**, or **locator.h**) unless your program makes reference to the structures or the definitions included in them.

The **sqlca.h** and **sqllda.h** files are described in detail in *The Informix Guide to SQL: Reference*. You also can read more about the **sqlca.h** header file later in this chapter, as well as in Chapter 7 of this manual. The **sqllda.h** header file also is discussed later in this chapter, as well as in Chapter 8 of this manual. The contents of **sqltype.h** is discussed in Chapter 9.

ESQL/C Preprocessor Support

The INFORMIX-ESQL/C preprocessor works in two stages:

- | | |
|---------|--|
| Stage 1 | acts as a preprocessor for ESQL/C. |
| Stage 2 | converts all of the embedded SQL code to C code. |

Stages 1 and 2 of the ESQL/C preprocessor mirror the C language preprocessor and compiler stage, as shown in Figure 1-1.

You can take advantage of Stage 1 of the ESQL/C preprocessor to incorporate other files into the source file or to define macros. You must include the files or define the macros necessary for the compilation of embedded SQL statements before Stage 2 of the preprocessor starts. It is not possible to use the C preprocessor to perform conditional compilation of ESQL/C statements because they were processed in Stage 2.

The ESQL/C preprocessor statements in Stage 1 have a similar syntax and effect as their counterparts in the C preprocessor, except that they take effect during input to the ESQL/C preprocessor.

You can use the following capabilities of the ESQL/C preprocessor when designing your embedded code:

- Include files
- The **define** and **undef** statements
- Conditional compilation statements (**ifdef**, **ifndef**, **else**, **elif**, **endif**)

Your ESQL/C source file also can contain commands for the C compiler preprocessor. These commands have no effect on ESQL/C statements but take effect in their usual way when the C compiler processes the source file.

Include Files

You can include other INFORMIX-ESQL/C files in your program in addition to header files. Do this by using the preprocessor statement **\$include** or **EXEC SQL include**. Use one of the following formats, replacing *filename* with the name of the file to be included in your program:

```
$include filename;  
  
EXEC SQL include filename;
```

Note: Use of the EXEC SQL keywords (in place of the dollar sign (\$)) conforms to ANSI standards.

Stage 1 of the **INFORMIX-ESQL/C** preprocessor reads the contents of *filename* into the current file at the position of the include statement before the **ESQL/C** preprocessing occurs. The standard `$include` of C includes the file after the **ESQL/C** preprocessor stage. You must use `$include` or **EXEC SQL include** if the specified file contains SQL statements.

You can write an `$include` statement with or without quotation marks around the filename.

```
$include filename;  
$include "pathname/filename";
```

For example, to include the file `constant_defs`, you can use either of the following statements. If you use a full pathname, you must enclose the pathname in quotes.

```
$include constant_defs;  
$include "/work/esqlcstuff/constant_defs";
```

When you use a `$include` statement with a filename, the preprocessor looks for the included file in this sequence:

1. In the current directory
2. In the directory `$INFORMIXDIR/incl/esql` (where `$INFORMIXDIR` represents the contents of the environment variable of that name)
3. In the directory `/usr/include`

The `$define` and `$undef` Statements

The remaining **ESQL/C** preprocessor statements have the same syntax and effect as their counterparts in the C preprocessor, except that they take effect during **INFORMIX-ESQL/C** preprocessing.

\$define assigns a compile-time value to a name.

\$undef removes a constant defined with `$define`.

The `$define` statement is limited to defining only symbols or integer constants. It does not support defining string constants or macros of statements that receive values at run time.

The following three examples show how to use the `$define` statement.

```
$define MAXROWS 25;
$define USETRANSACTIONS 1;
$define TRANS;
```

The *ifdef*, *ifndef*, *else*, *elif*, and *endif* Statements

The INFORMIX-ESQL/C preprocessor does not support a general `$if` statement; it supports only the `$ifdef` and `$ifndef` statements that test whether a name has been `$defined`.

The preprocessor handles the following statements:

<code>\$ifdef</code>	tests a name and executes subsequent statements if it has been defined with <code>\$define</code> .
<code>\$ifndef</code>	tests a name and executes subsequent statements if it has not been defined with <code>\$define</code> .
<code>\$elif</code>	begins an alternative section to an <code>\$ifdef</code> or <code>\$ifndef</code> condition that checks for the presence of another <code>\$ifdef</code> .
<code>\$else</code>	begins an alternative section to an <code>\$ifdef</code> or <code>\$ifndef</code> condition.
<code>\$endif</code>	closes an <code>\$ifdef</code> or <code>\$ifndef</code> condition.

In the following example, the `BEGIN WORK` statement only compiles if the name `USETRANSACTIONS` is defined:

```
$ifdef USETRANSACTIONS;
$begin work;
$endif;
```

Using Host Variables in SQL Statements

Host variables are normal C variables that you use in SQL statements. When you use a host variable in an SQL statement, precede its name with a dollar sign (\$) or a colon (:). The host variable **hostvar**, for example, appears in an SQL statement as `$hostvar` or `:hostvar`.

Note: Use of the colon (:) as a host-variable prefix conforms to ANSI standards.

Outside of an SQL statement, treat a host variable as you would a regular C variable.

You can define as many host variables as you need (up to the limit set for the symbol table of your C compiler).

Declaring Host Variables

Host variables are declared as ordinary C variables except that the declaration must either be prefaced by a dollar sign (\$) or contained within an EXEC SQL BEGIN DECLARE SECTION / EXEC SQL END DECLARE SECTION.

Note: Use of the EXEC SQL BEGIN/END DECLARE SECTION keywords conforms to ANSI standards.

Figure 1-2 shows an example of using the \$ format to declare host variables.

```
/* pointer to a character */
$ char  *hostvar;
/* integer */
$ int   hostint;
/* double */
$ double hostdbl;
/* character array */
$ char  hostarr[80];
/* structure */
$ struct {
    int svar1;
    int svar2;
    ...
} hoststruct;
```

Figure 1-2

Declaring host variables using the \$ syntax

Figure 1-3 shows an example of using the EXEC SQL format to declare host variables.

```
EXEC SQL BEGIN DECLARE SECTION
    char    *hostvar;
    int     hostint;
    double  hostdbl;
    char    hostarr[80];
EXEC SQL END DECLARE SECTION

EXEC SQL BEGIN DECLARE SECTION
    struct {
        int svar1;
        int svar2;
        ...
    } hoststruct;
EXEC SQL END DECLARE SECTION
```

Figure 1-3

Declaring host variables using the EXEC SQL syntax

Initializing Host Variables

INFORMIX-ESQL/C allows you to declare host variables with normal C initializer expressions. However, initializers containing character strings cannot contain embedded semicolons or ESQL/C keywords. Some valid examples of C initializers follow:

```
$int varname = 12;
$long cust_nos[8] = {0,0,0,0,0,0,0,9999};
```

Initializers are not checked for valid C syntax; they are simply copied to the output. The C compiler diagnoses any errors.

Scope of Host Variables

The rules governing the scope of a host variable are the same as those governing regular C variables:

- A host variable is an automatic variable unless you explicitly define it as external or static.
- A host variable declared in a function is local to that function and masks a definition by the same name outside of the function.
- You cannot define a host variable more than once in the same block of code.

Defining a Block

You can ensure that the host variables declared within a block of code are local to that block by using the combined symbol pair `${` and `}` to open and close the block. Alternatively, you can use a simple pair of braces, `{` and `}`. For example, the host variable `blk_int` in Figure 1-4 is valid only in the block of code between the braces, whereas `p_int` is valid in the block and outside of the block.

```
$int p_int
...
$select customer_num into $p_int from customer
    where lname = "Miller";
...
${
    $ int blk_int
    ...
    $blk_int = $p_int;
    select customer_num into blk_int from customer
        where lname = "Miller";
    ...
}$
...
```

Figure 1-4

Using braces to create a block of code

You can nest blocks up to 16 levels. The global level counts as level one.

Note: ANSI-standard syntax does not support the `${` and `}` symbols.

Types of Host Variables

Since host variables appear in SQL statements, they are associated with an SQL data type. In addition, a host variable must be declared as a C data type. The relationship between SQL data types and C data types is described in detail in Chapter 2. Figure 1-5 summarizes the relationship.

SQL Type	ESQL/C Predefined Data Type	C Language Type
CHAR(n) CHARACTER(n)	fixchar array[n], or string array [n+1]	char array [n + 1] or char *
BYTE TEXT	loc_t	
DATE		long int
DATETIME	datetime or dttime_t	
DECIMAL DEC NUMERIC MONEY	decimal or dec_t	
SMALLINT		short int
FLOAT DOUBLE PRECISION		double
INTEGER INT		long int
INTERVAL	interval or intrvl_t	
SERIAL		long int
SMALLFLOAT REAL		float
VARCHAR(m,x)	varchar[m+1] or string array[m+1]	char array[m+1]

Figure 1-5

Correspondence of SQL and C data types

If the host variable is not declared according to Figure 1-5, ESQL/C tries to convert data types, if the conversion is meaningful. See Chapter 2 for a discussion of data conversion.

Arrays of Host Variables

INFORMIX-ESQL/C understands and supports the declaration of arrays of variables. You must provide an integer value as the size of the array when you declare the array. An array of host variables can be either one or two dimensional.

You can use elements of an array within ESQL/C statements. For example, if you declare

```
$long customer_nos[10];
```

the following is possible:

```
for (i=1; i<10; i++)
{
$fetch customer_cursor into $customer_nos[i];
}
```

However, for data types other than CHAR, you cannot use the array name alone.

Structures as Host Variables

Structures can be declared as INFORMIX-ESQL/C host objects. In ESQL/C statements, you can name the structure variable as a whole or as its individual components. If a structure name is used, it is expanded into a list of component names. Structures can be nested.

```
$struct customer_t
{
int          c_no;
char         fname[32];
char         lname[32];
} cust_rec;
$struct customer_t cust2_rec;
```

With the preceding declaration,

```
$insert into customer values ($cust_rec);
```

is equivalent to

```
$insert into customer
  values ($cust_rec.c_no, $cust_rec.fname,
         $cust_rec.lname);
```

typedef Expressions as Host Variables

ESQL/C supports standard C **typedef** expressions and allows their use as host variables. For example, the following code creates the **smallint** type as a short integer and the **serial** type as a long integer. It then declares a **row_nums** variable as a **serial** type and a variable **counter** as a **smallint**.

```
$typedef short smallint;
$typedef long serial;
$serial row_nums [MAXROWS];
$smallint counter;
```

You cannot use a **typedef** that names a multidimensional array or a union as a host variable.

Null Values in Host Variables

The representation of null values depends on both the machine and the data type. Often, the representation does not correspond to a legal value for the C data type, and you should not attempt to perform arithmetic or other operations on a host variable that can have a null value.

INFORMIX-ESQL/C provides a function that enables you to test whether a host variable corresponds to a null value (**risnull**), and a function to set a host variable to a null value (**rsetnull**). See Chapter 2 for a description of these functions. You also can define indicator variables for host variables that correspond to database columns that allow null values.

Character Pointers as Host Variables

You can declare a character pointer as a host variable if the host variable is only used to input data to an SQL statement. For example, Figure 1-6 shows how you can associate a cursor with a statement and insert values into a table.

```
$char *s;
$char *i;
...
stcopy("select * from cust_calls",s);
stcopy("NS",i);
...
$prepare x from $s;
$insert into state values ($i, "New State");
```

Figure 1-6

Declaring a character pointer to input data

If you declare a character pointer as a host variable and use it to receive data from a SELECT statement, you receive a compile-time warning and your results may be truncated.

Host Variables as Function Parameters

You can use host variables as parameters to functions. You must precede the name of the host variable with the **parameter** keyword when you declare a host variable as a function parameter. For example, Figure 1-7 shows an example that expects three parameters, two of which are host variables.

```
f(s, id, s_size)
$parameter char s[20];
$parameter int id;
int s_size;
{
select fname into $s from customer
  where customer_num = $id;
...
}
```

Figure 1-7

Declaring host variables as parameters to functions

You cannot declare a parameter variable inside a block of C code.

You cannot use the **parameter** keyword in declarations of host variables that are not part of a function header. Doing so causes unpredictable errors.

ANSI C function parameter syntax is not currently supported. Therefore, you cannot use the **parameter** keyword in an ANSI C function header.

You can declare parameter variables within an EXEC SQL BEGIN DECLARE / EXEC SQL END DECLARE section. For example, the function header in Figure 1-7, when written using a DECLARE section, is shown in Figure 1-8.

```
f(s, id, s_size)
EXEC SQL BEGIN DECLARE SECTION
    parameter char s[20];
    parameter int id;
EXEC SQL END DECLARE SECTION
int s_size;
```

Figure 1-8

Using EXEC SQL BEGIN DECLARE in a function header

Indicator Variables

Since a null value is often not a definite value among other values, you must be able to determine whether an **INFORMIX-ESQL/C** statement returns a null value to a host variable. If a host variable corresponds to a database column that allows null values, you should define an indicator variable in association with a host variable. The associated host variable is called a main variable.

In addition to allowing a program to check for null values, an indicator variable can be used to check for truncated values that are returned by the database server.

Declaring Indicator Variables

To declare an indicator variable, declare the variable as an integer. For example, the following code declares the variable **nameind** as a short integer. It can then be used as an indicator variable.

```
$short nameind;
```

Indicator variables can be of any valid host variable data type except DATETIME or INTERVAL.

Values Returned in Indicator Variables

When an **INFORMIX-ESQL/C** statement returns a null value to a host variable (through the **INTO** clause of a **SELECT** or **FETCH** statement) and you defined an indicator variable, the indicator variable has a value of -1. The actual value in the host variable might not be a meaningful C value. If you did not assign an indicator variable to the host variable and a null value is returned, **ESQL/C** might generate an error, depending on how you compile the program:

- If you compile the program using the **-icheck** flag, **ESQL/C** generates an error and sets **sqlca.sqlcode** to a negative value when a null value is returned and no indicator variable is present. (See Chapter 7, “Error Handling.”)
- If you compile the program without using the **-icheck** flag, **ESQL/C** does not generate an error when a null value is returned and no indicator variable is present.

When a non-null SQL value is retrieved into a host variable character array, it can be truncated to fit. In this case, **ESQL/C** sets the associated indicator variable equal to the size in bytes of the SQL variable before truncation. The fact of truncation is signaled in the **sqlca** structure; the indicator variable tells you the length of the truncated value. If the returned value is neither null nor truncated, the indicator variable has the value 0.

Using Indicator Variables

You specify an indicator variable in an SQL statement in one of two ways:

- Place a colon (:) between the main (host) variable name and the indicator variable name. (You can use a dollar sign (\$) instead of a colon, but the colon makes the code easier to read.) Use the following format:

```
$hostvar:hostvarind
```

```
$hostvar$hostvarind
```

- Place the **INDICATOR** keyword between the main variable name and the indicator variable name. Use the following format:

```
$hostvar INDICATOR:hostvarind
```

```
$hostvar INDICATOR hostvarind
```

*Note: Use of the **INDICATOR** keyword conforms to ANSI standards.*

The code segments in Figure 1-9 and Figure 1-10 show examples of using indicator variables with host variables. In both examples, if **lname** is defined in the **customer** table as having a length longer than 15 characters, **nameind** contains the actual length of the **lname** column. The **name** host variable contains the first 15 characters of the **lname** value (the string name must be terminated with a null character). If the last name of the company representative with **customer_num** = 105 is shorter than 15 characters, only trailing blanks are truncated.

If company has a null value for this same customer, **compind** has a negative value. The contents of the character array **comp** cannot be predicted.

```

$char  name[16];
$char  comp[20];
$short nameind;
$short compind;
.
.
.
$select lname, company
        into $name:nameind, $comp:compind
        from customer
        where customer_num = 105;

```

Figure 1-9

Using indicator variables with the \$ and : symbols

```

EXEC SQL BEGIN DECLARE SECTION
        char  name[16];
        char  comp[20];
        short nameind;
        short compind;
EXEC SQL END DECLARE SECTION
.
.
.
EXEC SQL
        select lname, company
        into $name:nameind, $comp:compind
        from customer
        where customer_num = 105;

```

Figure 1-10

Using indicator variables with the EXEC SQL and : symbols

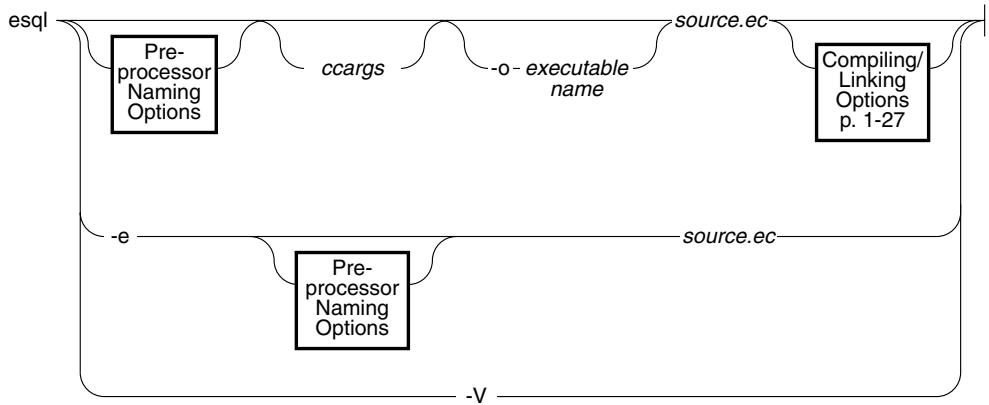
As an alternative to using the NULL keyword in an INSERT statement, you can use a host variable with a negative indicator variable.

Compiling INFORMIX-ESQL/C Programs

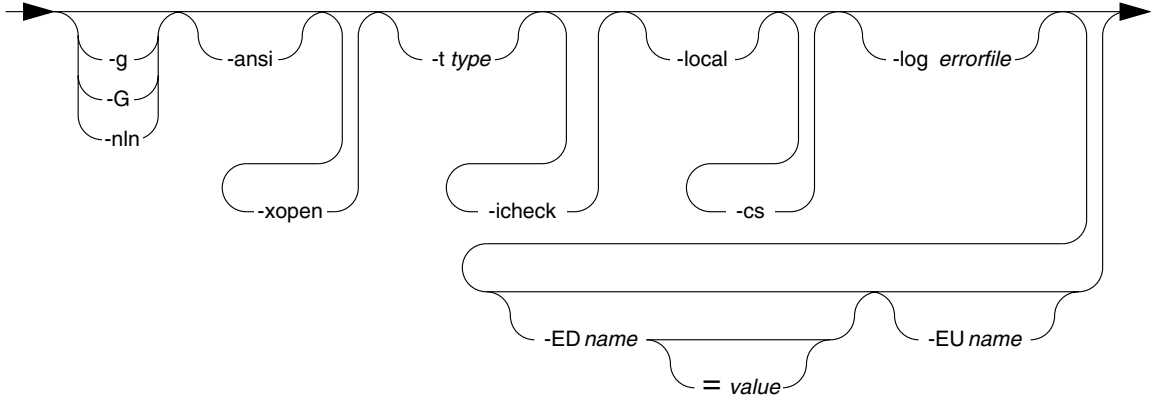
Before you can use the C compiler, you must preprocess your code that contains INFORMIX-ESQL/C statements. The ESQL/C preprocessor converts the embedded statements to C language code. You can then compile the resulting file with the C compiler to create an object file, which you can link with the ESQL/C libraries and your own libraries. You can use the **esql** command file that is installed with ESQL/C to perform all these tasks.

To preprocess and compile a C program that contains ESQL/C statements, give its filename the extension **.ec** and enter an **esql** command at your system prompt. The environment variable **INFORMIXDIR** must be set correctly for the **esql** command to work effectively. See Chapter 4 of *The Informix Guide to SQL: Reference* for a complete description of **INFORMIXDIR**.

Syntax of the *esql* Command



Pre-processor Naming Options



- ansi** checks for Informix extensions to ANSI standard syntax.
- `ccargs` arguments passed to the cc compiler.
- cs** indicates case sensitivity for cursor names and statement ids.
- e** preprocesses only, no compiling or linking.
- EDname** defines a user-supplied name to the preprocessor. This is the same as using a `$define` statement with `name` at the top of your **ESQL/C** program.

-E <i>name</i>	undefines a specified preprocessor name flag. This is the same as using a <code>\$undef</code> statement with <i>name</i> in your ESQL/C program.
-g	numbers every line (used by a debugger).
-G	no line number (used by a debugger; same as -nln).
-icheck	generates the code to check for a null value returned to a host variable that does not have an indicator variable associated with it; generates an error if such a case exists.
-local	specifies that the static cursor names and static statement ids that you declare in a file are local to that file. If you do not use the -local option, cursor names and statement ids, by default, are global entities.
-log <i>errorfile</i>	sends the error and warning messages to the specified file instead of to standard output.
-nln	no line number (used by a debugger; same as -G).
-o <i>executable name</i>	specifies the name of the executable file.
<i>source.ec</i>	specifies the name of the source file containing ESQL/C statements and C code. The file must have a .ec extension.
-V	prints preprocessor version information.
=val	lets you assign an initial value to the name, for example: <code>-EDMACNAME=62</code> . This is equivalent to the line: <code>\$ define MACNAME 62 ;</code> at the top of your ESQL/C program.
-xopen	indicates that the X/Open set of codes for the data types are used when a GET DESCRIPTOR or SET DESCRIPTOR statement is executed. It also generates warning messages for dynamic SQL statements that use Informix extensions to the X/Open standard.

Preprocessing Without Compiling or Linking

You can choose only to preprocess your ESQL/C program. To preprocess the code, use the **esql** command with the **-e** option and the appropriate parameters. The preprocessor creates a C program.

For example, to preprocess the program that resides in the file **demo1.ec**, you use the following command:

```
esql -e demo1.ec
```

If you want to preprocess **demo1.ec**, check for Informix extensions to ANSI-standard syntax, and not use line numbers, you use the following command:

```
esql -e -ansi -G demo1.ec
```

Using the Preprocessing Options

All of the preprocessor options described in the following sections can be used when either preprocessing only or when preprocessing, compiling, and linking.

Checking for Informix Extensions to ANSI-Standard Syntax

Use the **-ansi** argument to check for Informix extensions to ANSI-standard syntax. If you included Informix extensions to ANSI-standard syntax in your code, you receive warning messages when the file is preprocessed.

The following command preprocesses and compiles the **demo1.ec** program and verifies that it does not contain any Informix extensions to the ANSI-standard syntax:

```
esql -ansi demo1.ec
```

If you set the DBANSIWARN environment variable, your embedded SQL code is checked automatically for Informix extensions. You do not need to use the **-ansi** flag if you set DBANSIWARN. (See Chapter 4 of *The Informix Guide to SQL: Reference* for more information about DBANSIWARN.) If DBANSIWARN is set, a compiled user program can make run-time checks on Informix extensions to ANSI SQL syntax by using the **sqlca** (SQL Communications Area) structure, whether or not you compile with the **-ansi** parameter. (See Chapter 7 of this manual for details about error handling.)

Checking for Missing Indicator Variables

If you include the **-icheck** option, the preprocessor generates code in your program that returns an error if an SQL statement returns a null value to a host variable that does not have an associated indicator variable. If you do not use the **-icheck** option, no error is returned at run time if a null value is passed to a host variable without an indicator variable.

```
esql -icheck demo1.ec
```

Numbering Lines

By default, as the embedded SQL lines in your program are preprocessed, they are given a line number. If you want to include line numbers for every line (C and embedded SQL), use the **-g** option. If you do not want any line numbers, use the **-G** or **-nl** options.

Redirecting Errors and Warnings

By default, errors and warnings generated when you run **esql** are sent to standard output. If you want the errors and warnings to be put into a file, use the **-log** option with the filename. For example, the following command compiles the program **demo1.ec** and sends the errors to the **err.out** file:

```
esql -log err.out -o demorun demo1.ec
```

Defining and undefining Values While Preprocessing

You can use the **-ED** and **-EU** options to define or undefine values during preprocessing. Do not put a space between **ED** and the symbol name or between **EU** and the symbol name. Using **-EDname** is equivalent to using **\$define name**. The **-ED** option is processed before the code in your source file is preprocessed. The **-EU** option has the global effect of a **\$undef** statement over the whole file.

For example, if you use the following command line on the code in Figure 1-11, no code is generated because the `ENABLE_CODE` value was undefined from the command line:

```
esql -EENABLE_CODE define_ex.ec
```

```
/* define_ex.ec */
#include <stdio.h>
#include sqlca;
#define ENABLE_CODE;

main()
{...
#ifdef ENABLE_CODE;
    printf("First block enabled");
#endif ENABLE_CODE;
...
#ifdef ENABLE_CODE;
    $ define ENABLE_CODE;
#endif ENABLE_CODE;
...
#ifdef ENABLE_CODE;
    printf("Second block enabled");
#endif ENABLE_CODE;
}
```

Figure 1-11

ESQL/C excerpt that uses `ifdef`, `ifndef`, and `endif`

You can define a numeric constant using the `-ED` option as shown in the following example:

```
esql -EDMAXLENGTH 10 demo1.ec
```

Using X/Open Codes and the `-xopen` Option

If you include the `-xopen` option on the command line, your program is pre-processed using the X/Open codes for the SQL data types. If you use X/Open SQL in an **ESQL/C** program, all previous programs in the same application must be recompiled with the `-xopen` option. If you use the `-xopen` option, warning messages are generated for dynamic statements that use Informix extensions to the X/Open standard.

See Chapter 9 of this manual for more information about X/Open SQL.

Setting the Scope of Cursor Names and Statement Ids

If you use the **-local** option, static cursor names and static statement ids that you declare in a file are local to that file. If you do not use the **-local** option, cursor names and statement ids, by default, are global entities.

You cannot mix files compiled with and without the **-local** flag. If you mix them, you receive unpredictable results.

If you use the **-local** option, you must recompile the source files every time you rename them.

INFORMIX-ESQL/C adds a unique tag (two to nine characters long) to the cursor names and statement ids in an **ESQL/C** program. If the combined length of the cursor name (or statement id) and the unique tag exceeds 18 characters, you receive a warning message.

Case Sensitivity in Cursor Names and Statement Ids

You can use the **-cs** option to specify that cursor names and statement ids are case sensitive. If you do not use this option, cursor names and statement ids are not case sensitive. If you include statements such as those in the following example and do not use the **-cs** option, you receive an error; the preprocessor interprets this as a redeclaration of the same name:

```
$PREPARE st FROM "SELECT * FROM tab1";
$PREPARE ST FROM "INSERT INTO tab2 VALUES (1,2)";
$DECLARE curname CURSOR FOR st;
$DECLARE CURNAME CURSOR FOR ST;
```

If you use the **-cs** option with the previous example, no error is generated.

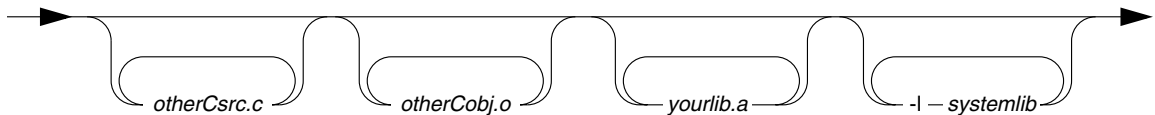
This option is included in this release for backward compatibility and its use is not recommended. The **-cs** option will not be available after Version 5.0.

Preprocessing, Compiling, and Linking with the *esql* Command

You can use the **esql** command to preprocess, compile, and link your program to other programs all in one step. You can specify any of the **ESQL/C** preprocessing options, as well as the compiling and linking options. The following illustration contains the syntax of the compiling and linking options to the **esql** command:

Syntax of the Compiling and Linking Options to the *esql* Command

Compiling/
Linking/
Options



<code>-l systemlib</code>	indicates other system libraries that you want to link.
<code>otherCobj.o</code>	indicates a C object file that you want to link with <i>source.ec</i> .
<code>otherCsrc.c</code>	indicates a C source file that you want to compile and link with <i>source.ec</i> .
<code>yourlib.a</code>	indicates your own special library that you want to link.

Passing Other Arguments to the *cc* Compiler

If you include options in the command line that are not supported by the **esql** command, the arguments are passed to the **cc** compiler. For example, the following command passes the `-c` argument to **cc**, which creates an object file:

```
esql -c demo1.ec
```

Passing Other C Source and Object Files to the *cc* Compiler

If you list other files with the `.c` extensions, the **esql** command passes them straight through to the C compiler **cc** to produce *source.o* and *otherCsrc.o*. These files are then linked with the appropriate **INFORMIX-ESQL/C** library routines, along with other C object files (*otherCobj.o*) that you include on the command line.

Using Other Libraries

If you want to use your own libraries or system libraries, you must explicitly include their names on the command line (for example, **libm.a** for mathematical functions).

A Sample INFORMIX-ESQL/C Program

The **demo1.ec** program illustrates most of the concepts presented in this chapter. It demonstrates how to use header files, declare and use host variables, and embed SQL statements.

Guide to *demo1.ec*

The sample INFORMIX-ESQL/C program, **demo1.ec**, uses a SELECT statement with no free parameters. That is, all of the information needed to run the SELECT statement is contained in the program and known at compile time.

The **demo1.ec** program reads a subset of the first and last names (those that start with C and later letters) from the **customer** table in the **stores5** database. Two host variables (**\$fname** and **\$lname**) are used to hold the data from the **customer** table. A cursor is declared to manage the information that is retrieved from the table. The rows are fetched one by one and the names are printed to standard output.

```
1 #include <stdio.h>
2 $include sqlca;
3
4 /* Uncomment the following line if the database has
5    transactions: */
6
7 /* $define TRANS; */
8
9 $define FNAME_LEN 15;
10 $define LNAME_LEN 15;
11
12 main()
13 {
14     {
15     $char fname[ FNAME_LEN + 1 ];
16     $char lname[ LNAME_LEN + 1 ];
17
18     printf( "\nDEMO1 Sample ESQL program running.\n\n");
19
20     $database stores5;
21
22     $declare democursor cursor for
23         select fname, lname
24             into $fname, $lname
25             from customer
26             where lname > "C";
```

Continued on page 1-30

Lines 1 to 3

The `#include <stdio.h>` statement includes the `stdio.h` UNIX header file from the `/usr/include` directory. The `stdio.h` file enables `demo1` to use the standard I/O library. The `sqlca.h` file is an `ESQL/C` header file that defines the structure that holds information about `ESQL/C` errors when they occur.

Lines 4 to 10

Lines 4 through 10 contain code that are processed by Stage 1 of the `ESQL/C` preprocessor. Lines 4 through 6 are comment lines for line 7, which, if uncommented, defines `TRANS`, a symbol that indicates that the database was created so that it uses transactions. Lines 9 and 10 define the constants `FNAME_LEN` and `LNAME_LEN` for use in host variable definitions later in the program.

Lines 15 and 16

These lines define host variables for the `fname` and `lname` columns of the `customer` table. A host variable receives data that is fetched from a table and supplies data that is written to a table. The length of the `fname` array is one greater than the length of the character column with which it is associated. The extra byte is necessary to hold the null-terminator.

Line 18

Line 18 simply lets the user of the program know that the program started to execute.

Line 20

Line 20 is the first SQL statement in the program. It opens the database named `stores5`. The `stores5` database must be created before it can be opened.

Lines 22 through 26

These lines contain a `DECLARE` statement that creates a cursor named `democursor` to manage the data that is read from the `customer` table. The type of data that is read from the table is determined by the `SELECT` statement contained in the `DECLARE` statement on lines 23 through 26. According to the `SELECT` statement, only the first and last names of the customers whose last name (`lname`) starts with a `C` or a later letter are read.

```
27 $ifdef TRANS;
28 $begin work;
29 $endif;
30
31 $open democursor;
32
33 for (;;)
34 {
35     $fetch democursor;
36     if (sqlca.sqlcode != 0) break;
37     printf("%s %s0, fname, lname");
38 }
39
40 $close democursor;
41
42 $ifdef TRANS;
43 $commit work;
44 $endif;
45
46 printf("\nProgram Over.\n");
47 }
```

Lines 27 to 29

Lines 27 and 29 are processed by Stage 1 of the **ESQL/C** preprocessor. If the **TRANS** symbol is set, the lines between the **\$ifdef** and **\$endif** are compiled. In this case, the **BEGIN WORK** statement on line 28 is compiled into the code.

Line 31

The **OPEN** statement opens the **democursor** cursor.

Line 33 to 38

This section of code is a **FOR** loop that contains a **FETCH** statement. For each iteration of the loop, the **FETCH** statement uses the **democursor** cursor to select a row from the table and put the data from that row into the host variables **fname** and **lname**. As long as **sqlca.sqlcode** equals zero, the data was fetched successfully and the loop continues. If an error occurs, the database server sets **sqlca.sqlcode** to a non-zero number. When all of the rows that matched the **SELECT** criteria are fetched, the database server sets **sqlca.sqlcode** to 100. So, when the value of **sqlca.sqlcode** is not equal to zero, the loop is exited. For more information about the **SQLCA** structure, see Chapter 5 of *The Informix Guide to SQL: Reference*.

Line 40

The CLOSE statement dissociates the cursor from the SELECT statement.

Line 42 to 4:

If the TRANS symbol is set, line 43 (COMMIT WORK) is compiled. In this case, the BEGIN WORK statement on line 28 would also have been compiled into the code.

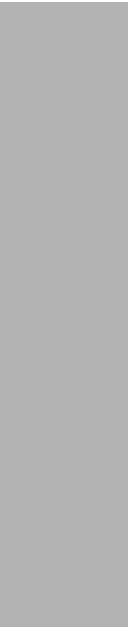
Line 44 and 45

The last two lines of the program simply tell the user that the program is over and close the main function.

INFORMIX-ESQL/C

Data Types

Chapter Overview	3
Choosing Data Types for Host Variables	3
Defined Integers for Data Types	5
Character Data Type Choices	5
Data Conversion	6
When Conversion Occurs	6
What Happens in a Conversion	7
Numbers to Strings	8
Numbers to Numbers	8
Operations on Numeric Values	8
Data Conversion When Fetching Rows	9
Converting Between DATETIME and DATE Data Types	9
Converting Between VARCHAR and Character Data Types	10
Data Type Function Descriptions	12
RISNULL	13
RSETNULL	16
RTYPALIGN	19
RTYPMSIZE	22
RTYPNAME	25
RTYPWIDTH	28
Numeric-Formatting Routines	31
Formatting Numeric Strings	31
Example Format String	33
Example Format String	34
Example Format String	35



Example Format String 36
RFMTDOUBLE 37
RFMTLONG 40

Chapter Overview

This chapter contains information about the SQL and C data types you can use to manipulate values in your **INFORMIX-ESQL/C** program. It also contains information about the routines available to you to determine and work with data types in **ESQL/C**. This chapter includes the following topics:

- Choosing the appropriate data type for a host variable
- Converting from one data type to another
- Working with routines that you can use with different data types
- Working with routines that you can use with null values

Choosing Data Types for Host Variables

Since host variables appear in SQL statements, they are associated with an SQL data type. The SQL data types that are available in a database are described in detail in Chapter 3 of *The Informix Guide to SQL: Reference*. For each column of a table in a database, you must declare a host variable of the appropriate C data type. The correspondence between SQL data types and host variable types is outlined in Figure 2-1.

SQL Type	ESQL/C Predefined Data Type	C Language Type
CHAR(n) CHARACTER(n)	fixchar array[n], or string array [n+1]	char array [n + 1] or char *
BYTE TEXT	loc_t	
DATE		long int
DATETIME	datetime or dttime_t	
DECIMAL DEC NUMERIC MONEY	decimal or dec_t	
SMALLINT		short int
FLOAT DOUBLE PRECISION		double
INTEGER INT		long int
INTERVAL	interval or intrvl_t	
SERIAL		long int
SMALLFLOAT REAL		float
VARCHAR(m,x)	varchar[m] or string array[m+1]	char array[m+1]

Figure 2-1 Correspondence between SQL and C data types

If you declare a host variable for a DATE, SMALLINT, FLOAT, INTEGER, or SERIAL database column, the host variable is simply a C-language variable of the type specified in Figure 2-1.

If you use BYTE, DECIMAL, DATETIME, INTERVAL, MONEY, TEXT, or VARCHAR columns, you must use the appropriate **INFORMIX-ESQL/C** predefined data type, or structure, as a host variable. Using these specialized

structures is described in detail in other chapters of this manual. The following list details where you can find the description of how to use the host data type:

BYTE	Chapter 6, “Working with Binary Large Objects”
DATETIME	Chapter 5, “Working with Time Data Types”
DECIMAL	Chapter 4, “Working with the DECIMAL Data Type”
INTERVAL	Chapter 5, “Working with Time Data Types”
MONEY	Chapter 4, “Working with the DECIMAL Data Type”
TEXT	Chapter 6, “Working with Binary Large Objects”
VARCHAR	Chapter 3, “Working with Character and String Data Types”

Defined Integers for Data Types

The `sqltypes.h` header file has a defined integer equivalent for each Informix database data type. For example, some of the entries in `sqltypes.h` are shown in Figure 2-2. (For brevity, only a few statement types are shown.)

```
#define SQLINT          2
#define SQLFLOAT       3
#define SQLSMFLOAT     4
#define SQLDECIMAL     5
#define SQLSERIAL      6
#define SQLDATE        7
#define SQLMONEY       8
.
.
.
```

Figure 2-2 Excerpt from `sqltype.h` file

These predefined data types are used as arguments for some of the routines in the `ESQL/C` library. When you need to use a definition, check the contents of the `sqltypes.h` file for the appropriate value.

Character Data Type Choices

If you use a character data type for your database column, you can choose the **char**, **string**, **fixchar**, or **varchar** data type for your host variable. The differences between the three data types are as follows:

char The **char** data type pads the value with trailing blanks up to the size the CHAR column returned. The **char** data type is

- null-terminated. It should be declared with a length of $[n+1]$, where n is the size of the column, to allow for the terminating null.
- string** The **string** data type differs from the **char** data type by truncating trailing blanks before inserting a null character to signal the end of the string. It should be declared with a length of $[n+1]$, where n is the size of the column, to allow for the terminating null.
- fixchar** The **fixchar** data type is the same as the **char** data type except that it does not add the trailing null byte to terminate the string. This means that you can declare a **fixchar** host variable corresponding to a CHAR(n) column as an array with n components.
- varchar** The **varchar** data type is implemented as an array of characters. The **varchar** data type is null-terminated. It should be declared with a length of $[m+1]$, where m is the maximum size of the column, to allow for the terminating null.

See Chapter 3, “Working with Character and String Data Types,” for more information about using character variables.

Data Conversion

When there is a discrepancy between the data type of a database variable and that of the host variable, or between the data type of two columns, **ESQL/C** attempts to convert one data type into the other. This includes the conversion of a CHAR data type into a number data type when the CHAR variable is a representation of a number. For example, when a comparison is made between a CHAR value and a number value, **ESQL/C** converts the CHAR value to a number value.

When Conversion Occurs

If you try to assign a value from a database table into a host variable that is not declared according to the correspondence shown in Figure 2-1, **ESQL/C** attempts to convert data types, if the conversion is meaningful.

Conversion can occur in many situations. The following list names a few common conversion situations:

- If you use a condition that compares two different types of values, such as comparing the contents of a zip code column to an integer value.
- If you insert values into a table using a host variable of one type and a receiving column of another type.
- If a numeric value of one type operates on a value of another type, both values are converted to a DECIMAL data type before the operation occurs.

What Happens in a Conversion

If a number type is converted to a character data type, a string is created for the character data type.

If conversion is not possible, either because it makes no sense or because the receiving variable is too small to accept the converted value, **ESQL/C** returns values as described in Figure 2-3, where N represents a number type and C represents a character type:

Conversion	Problem	Result
C → C	Does not fit	The string is truncated; sqlca.sqlwarn.sqlwarn1 is set to W ; the indicator variable is set to the size of the original string.
N → C	Does not fit	The string is filled with asterisks; sqlca.sqlwarn.sqlwarn1 is set to W ; the indicator variable is set to a positive integer.
C → N	Not a number	The number is undefined; sqlca.sqlcode is set to negative.
C → N	Overflow	The number is undefined; sqlca.sqlcode is negative.
N → N	Overflow	The number is undefined; sqlca.sqlcode is negative.

Figure 2-3 Number to character data type conversion problems and results

Numbers to Strings

The conversion of a number type to a character type occurs through creating a string. **INFORMIX-ESQL/C** uses an exponential format for very large or very small numbers.

Numbers to Numbers

If two values of different types operate on one another, they are converted to a **DECIMAL** value and then the operation occurs.

Operations on Numeric Values

INFORMIX-ESQL/C carries out all arithmetic in an arithmetic expression in type **decimal**. The type of the resulting variable determines the format of the stored or printed result. The following rules apply to the precision and scale of the **decimal** variable that results from an arithmetic operation on two numbers:

- All operands, if not already **decimal**, are converted to **decimal** and the resulting number is **decimal**.

Convert Type	To
FLOAT	decimal(16)
SMALLFLOAT	decimal(8)
INTEGER	decimal(10,0)
SMALLINT	decimal(5,0)

- The precision and scale of the result of an arithmetic operation depend on the precision and scale of the operands and on the type of arithmetic expression. The rules for arithmetic operations on operands with definite scale are summarized in Figure 2-4 (at the end of this section). When one of the operands has no scale (floating decimal), the result is a floating decimal.
- If the operation is addition or subtraction, **ESQL/C** adds trailing zeros to the operand with the smaller scale until the scales are equal.
- If the type of the result of an arithmetic operation requires the loss of significant digits, **ESQL/C** reports an error.
- Leading or trailing zeros are not considered significant digits and do not contribute to the determination of precision and scale.

In Figure 2-4, let p_1 and s_1 be the precision and scale of the first operand, and let p_2 and s_2 be the precision and scale of the second operand.

Operation	Precision and Scale of Result
Addition and Subtraction	Precision: $\text{MIN}(32, \text{MAX}(p_1 - s_1, p_2 - s_2) + \text{MAX}(s_1, s_2) + 1)$ Scale: $\text{MAX}(s_1, s_2)$
Multiplication	Precision: $\text{MIN}(32, p_1 + p_2)$ Scale: $s_1 + s_2$
Division	Precision: 32 Scale: $32 - p_1 + s_1 - s_2$ (cannot be negative)

Figure 2-4 Precision and scale of decimal results

Data Conversion When Fetching Rows

You can automatically convert DATETIME and INTERVAL values between database columns and host variables of character type **char**, **string**, or **fixchar**. The fields of the DATETIME or INTERVAL value in the database are converted to a character string, which is stored in the host variable. If the host variable is too short, the string is truncated, `sqlca.sqlwarn.sqlwarn1` is set to **W** and the indicator variable (if any) is set to the needed length.

Note that DATETIME and INTERVAL values cannot be fetched automatically into number host variables.

Converting Between DATETIME and DATE Data Types

No functions are provided to convert automatically between the DATETIME and DATE data types. You can perform these conversions using existing functions and intermediate strings.

To convert a DATETIME value to a DATE value, use these steps:

1. Use `dtextend` to adjust the DATETIME qualifier to *year to day*.
2. Apply `dttoasc`, creating a character string in the form *yyyy-mm-dd*.
3. Use `rdefmtdate` with a pattern argument of *yyyy-mm-dd* to convert the string to a DATE value.

To convert a DATE value into a DATETIME value, use these steps:

1. Declare a host variable with a qualifier of *year to day* (or, initialize the qualifier with the value returned by TU_DTENCODE(TU_YEAR,TU_DAY)).
2. Use **rfmtdate** with a pattern of *yyyy-mm-dd* to convert the DATE value to a character string.
3. Use **dtcvasc** to convert the character string to a value in the prepared DATETIME variable.
4. If necessary, use **dtextend** to adjust the DATETIME qualifier.

Converting Between VARCHAR and Character Data Types

Figure 2-5 shows the conversion of VARCHAR data to **char**, **string**, and **fix-char** character data types. When calculating the length of the source item, trailing spaces are not counted.

Source	Destination	Result
VARCHAR	char	If the source is longer, truncate, set indicator, and null terminate. If the destination is longer, pad with trailing spaces and null terminate.
VARCHAR	fixchar	If the source is longer, truncate and set indicator. If the destination is longer, pad with trailing spaces.
VARCHAR	string	If the source is longer, truncate, set indicator, and null terminate. If the destination is longer, null terminate.
char	VARCHAR	If the source is longer than the max VARCHAR, truncate, set indicator, and null terminate. If the max VARCHAR is longer than the source, the destination length = source "length" and null terminate.
fixchar	VARCHAR	If the source is longer than the max VARCHAR, truncate, set indicator, and null terminate. If the max VARCHAR is longer than the source, the destination length = source "length" and null terminate.

string	VARCHAR	If the source is longer than the max VARCHAR, truncate, set indicator, and null terminate. If the max VARCHAR is longer than the source, the destination length = source "length" and null terminate.
--------	---------	--

Figure 2-5

Converting VARCHAR data types to and from host data types

Figure 2-6 shows VARCHAR to character data type conversion examples. (A + represents a space character.)

Source			Destination		
Type	Contents	Length	Type	Contents	Indicator
VARCHAR(9)	Fairfield	9	char(5)	Fair\0	9
VARCHAR(9)	Fairfield	9	char(12)	Fairfield++\0	0
VARCHAR(12)	Fairfield+++	12	char(10)	Fairfield\0	12
VARCHAR(10)	Fairfield+	10	char(4)	Fai\0	10
VARCHAR(11)	Fairfield++	11	char(14)	Fairfield++++\0	0
VARCHAR(9)	Fairfield	9	fixchar(5)	Fairf	9
VARCHAR(9)	Fairfield	9	fixchar(10)	Fairfield+	0
VARCHAR(10)	Fairfield+	10	fixchar(9)	Fairfield	10
VARCHAR(10)	Fairfield+	10	fixchar(6)	Fairfi	10
VARCHAR(10)	Fairfield+	10	fixchar(11)	Fairfield++	0
VARCHAR(9)	Fairfield	9	string(4)	Fai\0	9
VARCHAR(9)	Fairfield	9	string(12)	Fairfield\0	0
VARCHAR(12)	Fairfield+++	12	string(10)	Fairfield\0	12
VARCHAR(11)	Fairfield++	11	string(6)	Fairf\0	11
VARCHAR(10)	Fairfield++	10	string(11)	Fairfield\0	0

Figure 2-6

Converting VARCHAR data types to character data types

Figure 2-7 shows character to VARCHAR data type conversion examples.

Source			Destination		
Type	Contents	Length	Type	Contents	Length
char(10)	Fairfield\0	10	VARCHAR(4)	Fair	4
char(10)	Fairfield\0	10	VARCHAR(11)	Fairfield	9
char(11)	Fairfield++\0	11	VARCHAR(9)	Fairfield	9
char(12)	Fairfield+++\0	12	VARCHAR(6)	Fairfi	6
char(10)	Fairfield+\0	10	VARCHAR(11)	Fairfield	9
fixchar(9)	Fairfield	9	VARCHAR(3)	Fai	3
fixchar(9)	Fairfield	9	VARCHAR(11)	Fairfield	9
fixchar(11)	Fairfield++	11	VARCHAR(9)	Fairfield	9
fixchar(13)	Fairfield++++	13	VARCHAR(7)	Fairfie	7
fixchar(10)	Fairfield+	10	VARCHAR(12)	Fairfield	9
string(9)	Fairfield\0	9	VARCHAR(4)	Fair'	4
string(9)	Fairfield\0	9	VARCHAR(11)	Fairfield	9

Figure 2-7

Converting character data types to VARCHAR data types

Data Type Function Descriptions

The following table lists the set of Informix library functions that act on more than one data type, work on null values, or format certain variable types. Those beginning with **rtyp** provide machine-independent size and alignment information for different data types. Those beginning with **rfmt** convert a value to a formatted string.

Function name	Description
rfmtdouble	Converts a double to a string
rfmtlong	Converts a long integer to a formatted string
risnull	Checks whether a C variable is null
rsetnull	Sets a C variable to null
rtypalign	Aligns data on proper type boundaries
rtypmsize	Gives byte size of SQL data types
rtypname	Converts data type to string
rtypwidth	Gives minimum conversion byte size

RISNULL

Purpose

The **risnull** function checks whether a C variable is null.

Syntax

```
int risnull(type, ptrvar)
           int type;
           char *ptrvar;
```

<i>ptrvar</i>	is a pointer to the C variable.
<i>type</i>	is an integer corresponding to the data type of a C variable. (See "Defined Integers for Data Types" on page 2-5.)

Return Codes

1	The variable is null.
0	The variable is not null.

Example

```

/*
 * risnull.ec *
 This program prints rows from the order table where the paid_date is NULL.
 */

#include <stdio.h>

#include sqltypes;

char errmsg[400];

main()
{
    char ans;
    $long order_num, order_date, ship_date, paid_date;

    $database stores5;          /* open stores5 database */
    err_chk("Open database");
    $declare c cursor for
        select order_num, order_date, ship_date, paid_date from orders;
    $open c;
    err_chk("Open cursor");
    printf("\n Order#\tPaid?\n");    /* print column hdgs */
    while(1)
    {
        $fetch c into $order_num, $order_date, $ship_date, $paid_date;
        if (SQLCODE == SQLNOTFOUND) /* if end of rows */
            break;                /* terminate loop */
        err_chk("Fetch");
        printf("%5d\t", order_num);
        if (risnull(CDATETYPE, (char *)&paid_date)) /* is price NULL ? */
            printf("NO\n");
        else
            printf("Yes\n");
    }
}

/*
 err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
 rgetmsg() to display the message for the error number in sqlca.sqlcode.
 */

err_chk(name)
char *name;
{
    if (sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}

```

Example Output

Order#	Paid?
1001	Yes
1002	Yes
1003	Yes
1004	NO
1005	Yes
1006	NO
1007	NO
1008	Yes
1009	Yes
1010	Yes
1011	Yes
1012	NO
1013	Yes
1014	Yes
1015	Yes
1016	NO
1017	NO
1018	Yes
1019	Yes
1020	Yes
1021	Yes
1022	Yes
1023	Yes

RSETNULL

Purpose

The `rsetnull` function sets a C variable to a value that corresponds to a database null value.

Syntax

```
int rsetnull(type, ptrvar)
    int type;
    char *ptrvar;
```

ptrvar is a pointer to the C variable.

type is an integer corresponding to the data type of a C variable. (See "Defined Integers for Data Types" on page 2-5.)

Example

```
/*
 * rsetnull.ec *

 This program fetches rows from the stock table for a chosen manufacturer
 and allows the user to set the unit_price to NULL.
 */

#include <stdio.h>
#include <ctype.h>
#include <decimal.h>
#include sqltypes.h;

#define LCASE(c) (isalpha(c) ? (isupper(c) ? tolower(c) : c) : c)

char format[] = "($$, $$$, $$$.&&)&";

char decdsply[20];
char errmsg[400];

$short stock_num;
$char description[16];
$dec_t unit_price;

main()
{
    $char manu_code[4];
    char ans;

    $database stores5; /* open stores5 database */
    err_chk("OPEN");
    $declare upcurs cursor for /* declare cursor */
```

```
        select stock_num, description, unit_price from stock
        where manu_code = $manu_code
        for update of unit_price;
printf("\n\tEnter Mfr. code: ");          /* prompt for mfr. code */
gets(manu_code);                          /* get mfr. code */
rupshift(manu_code);                       /* Make mfr code upper case */
$open upcurs;                              /* open select cursor */
err_chk("Open upcurs cursor");
/*
   Display Column Headings
*/
printf("\nStock # \tDescription \tUnit Price");
while(1)
{
    /* get a row */
    $fetch upcurs into $stock_num, $description, $unit_price;
    if(!err_chk("FETCH"))                  /* check result */
        break;
    if(risnull(CDECIMALTYPE, &unit_price)) /* if unit_price IS NULL */
        continue;                         /* skip to next row */
    rfmtdec(&unit_price, format, decdsply); /* format unit_price */
    /* display item */
    printf("\n\t%d\t\t%15s\t%s", stock_num, description, decdsply);
    ans = ' ';
    /* Set unit_price to NULL? y(es) or n(o) */
    while((ans = LCASE(ans)) != 'y' && ans != 'n')
    {
        printf("\n\t. . . Set unit_price to NULL ? (y/n) ");
        scanf("%1s", &ans);
    }
    if(ans == 'y')                          /* if yes, NULL to unit_price */
    {
        rsetnull(CDECIMALTYPE, &unit_price);
        $update stock set unit_price = $unit_price
            where current of upcurs;        /* and update current row */
        err_chk("UPDATE");
    }
}
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode < 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
    return((sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1);
}
```

Example Output

```
Enter Mfr. code: HRO

Stock #      Description      Unit Price
  1      baseball gloves      $250.00
. . . Set unit_price to NULL ? (y/n) n

  2      baseball              $126.00
. . . Set unit_price to NULL ? (y/n) y

  4      football              $480.00
. . . Set unit_price to NULL ? (y/n) n

  7      basketball           $600.00
. . . Set unit_price to NULL ? (y/n) y
```

RTYPALIGN

Purpose

The **rtypalign** function returns the position of the next proper boundary for a variable of the specified data type.

Syntax

```
int rtypalign(pos, type)
           int pos;
           int type;
```

pos is the current position in a buffer.

type is the integer code for a C or SQL data type. (See “Defined Integers for Data Types” on page 2-5.)

Usage

The **rtypalign** and **rtypmsize** functions are useful when setting up an **sqllda** structure to fetch data into a buffer; you can use the functions to provide machine independence.

The value of *type* is returned by the DESCRIBE statement into **sqllda.sqlvar->sqltype**.

You can see an application of the **rtypalign** function in the **unload.ec** demonstration program.

Return Code

>0 The offset of the next proper boundary for a variable of that type.

Example

```

/*
 * rtypalign.ec *

The following program prepares a select on all columns from the orders
table and then calculates the 'aligned' position in a buffer for each column.
*/

#include <decimal.h>

#include sqltypes;

char typnm[30], *rtypname();

char errmsg[400];

main()
{
    int i, pos;
    struct sqllda *sql_desc;
    struct sqlvar_struct *col;

    $ database stores5;          /* open stores5 database */
    err_chk("Open");
    $ prepare query_1 from "select * from orders"; /* prepare select */
    err_chk("Prepare");
    $ describe query_1 into sql_desc;          /* initialize sqllda */
    err_chk("Describe");

    col = sql_desc->sqlvar;
    printf("\n\ttype\t\tlen\t\next\taligned\n"); /* display column hdgs. */
    printf("\t\t\t\t\tposn\tposn\n\n");
    /*
    for each column in the orders table
    */
    i = 0;
    pos = 0;
    while(i++ < sql_desc->sqld)
    {
        /* Modify sqllen if SQL type is DECIMAL or MONEY */
        if(col->sqltype == SQLDECIMAL || col->sqltype == SQLMONEY)
        {
            col->sqltype = CDECIMALTYPE; /* change to DECIMAL */
            col->sqllen = sizeof(dec_t);
        }
        /*
        * display name of SQL type, length and un-aligned buffer position
        */
        printf("\t%s\t\t%d\t%d", rtypname(col->sqltype), col->sqllen, pos);

        pos = rtypalign(pos, col->sqltype); /* align pos. for type */
    }
}

```

```
    printf("\t%d\n", pos);

    pos += col->sqlen;      /* set next position */
    ++col;                 /* bump to next column */
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}
```

Example Output

type	len	next posn	aligned posn
serial	4	0	0
date	4	4	4
integer	4	8	8
char	40	12	12
char	1	52	52
char	10	53	53
date	4	63	64
byte	22	68	68
byte	22	90	90
date	4	112	112

RTYPMSIZE

Purpose

The **rtypmsize** function returns the number of bytes you must allocate in memory for the specified C or SQL type.

Syntax

```
int rtypmsize(sqltype, sqllen)
           int sqltype;
           int sqllen;
```

<i>sqllen</i>	is the number of bytes in the data file for the specified SQL type.
<i>sqltype</i>	is the integer code of the C or SQL type. (See “Defined Integers for Data Types” on page 2-5.)

Usage

The **rtypmsize** and **rtypalign** functions are useful when setting up an **sqlda** to fetch data into a buffer; you can use the functions to provide machine independence.

The **rtypmsize** function is designed to be used with the **sqlda** structure returned by a DESCRIBE statement. The *sqltype* and *sqllen* components correspond to the components of the same name in each **sqlda.sqlvar** structure.

For CCHARTYPE and CSTRINGTYPE, **rtypmsize** adds one byte to the number of characters for the null terminator. For CFIXCHARTYPE, there is no null terminator.

You can see an application of the **rtypmsize** function in the **unload.ec** demonstration program.

Return Codes

0	The <i>sqltype</i> is not a valid SQL type.
>0	The number of bytes required for data type is <i>n</i> .

Example

```

/*
 * rtypmsize.ec *

The following program prepares a select statement on all columns of the
catalog table and then displays the number of bytes needed to store each
column in memory.
*/

#include <stdio.h>

#include sqltypes;

char errmsg[400];

main()
{
    int i;
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    $ database stores5; /* open stores5 database */
    err_chk("Open");
    $ prepare query_1 from "select * from catalog"; /* prepare select */
    err_chk("Prepare");
    $ describe query_1 into sql_desc; /* setup sqlda */
    err_chk("Describe");

    printf("\n\tColumn          Type      Size\n\n"); /* column hdgs. */
    /*
     * For each column in the catalog table display the column name and
     * the number of bytes needed to store the column in memory.
     */
    for(i = 0, col = sql_desc->sqlvar; i < sql_desc->sqlld; i++, col++)
        printf("\t%-20s%-8s%3d\n", col->sqlname, rtypname(col->sqltype),
            rtypmsize(col->sqltype, col->sqlllen));
}

/*
 * err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
 * rgetmsg() to display the message for the error number in sqlca.sqlcode.
 */
err_chk(name)
char *name;
{
    if(sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}

```

Example Output

Column	Type	Size
catalog_num	serial	4
stock_num	smallint	2
manu_code	char	4
cat_descr	text	64
cat_picture	byte	64
cat_advert	varchar	256

RTYPNAME

Purpose

The **rtypname** function returns a null-terminated string containing the name of the specified SQL type.

Syntax

```
char *rtypname(sqltype)  
    int sqltype;
```

sqltype is an integer code for one of the SQL types. (See “Defined Integers for Data Types” on page 2-5.)

Return Codes

The following values are returned:

sqltype	Return String
SQLCHAR	"char"
SQLSMINT	"smallint"
SQLINT	"integer"
SQLFLOAT	"float"
SQLSMFLOAT	"smallfloat"
SQLDECIMAL	"decimal"
SQLSERIAL	"serial"
SQLDATE	"date"
SQLMONEY	"money"
SQLDTIME	"datetime"
SQLINTERVAL	"interval"
invalid type	"" (null string)

Example

```

/*
 * rtypename.ec *

The following program displays name of the columns
and the data type for each column for the 'orders' table.
*/

#include <stdio.h>

#include sqltypes;

char errmsg[400];

main()
{
    int i;
    char *rtypname();
    struct sqllda *sql_desc;
    struct sqlvar_struct *col;

    $ database stores5;          /* open stores5 database */
    err_chk("Open");
    $ prepare query_1 from "select * from orders"; /* prepare select */
    err_chk("Prepare");
    $ describe query_1 into sql_desc; /* initialize sqllda */
    err_chk("Describe");

    printf("\n\tColumn Name \t\tSQL type\n\n");
    /*
     * For each column in the orders table display the column name and
     * the name of the SQL data type
     */
    for (i = 0, col = sql_desc->sqlvar; i < sql_desc->sqllda; i++, col++)
        printf("\t%-15s\t\t%s\n", col->sqlname, rtypename(col->sqltype));
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if (sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}

```

Example Output

Column Name	SQL type
order_num	serial
order_date	date
customer_num	integer
ship_instruct	char
backlog	char
po_num	char
ship_date	date
ship_weight	decimal
ship_charge	money
paid_date	date

RTYPWIDTH

Purpose

The **rtypwidth** function returns the minimum number of characters required to avoid truncation when converting a value with an SQL type to a character data type.

Syntax

```
int rtypwidth(sqltype, sqllen)
           int sqltype;
           int sqllen;
```

sqllen is the number of bytes in the data file for the specified SQL type.

sqltype is the integer code of the SQL type. (See “Defined Integers for Data Types” on page 2-5.)

Usage

The **rtypwidth** function is designed to be used with the **sqlda** structure that is returned by a DESCRIBE statement. The *sqltype* and *sqllen* components correspond to the components of the same name in each **sqlda.sqlvar** structure.

Return Codes

0 The *sqltype* is not a valid SQL type.

> 0 A value of type *sqltype* requires a minimum of *n* characters to be expressed.

Example

```

/*
 * rtypwidth.ec *

The following program displays the name of columns in 'orders' table and
the number of characters required to store the data type for each column.
*/

#include <stdio.h>

char errmsg[400];

main()
{
    int i, numchars;
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    $ database stores5;          /* open stores5 database */
    err_chk("Open");
    $ prepare query_1 from "select * from orders"; /* prepare select */
    err_chk("Prepare");
    $ describe query_1 into sql_desc;          /* setup sqlda */
    err_chk("Describe");

    printf("\n\tColumn Name    \t# chars\n");
    /*
     * For each column in orders print the column name and the minimum
     * number of characters required to convert the SQL type to a character
     * data type
     */
    for (i = 0, col = sql_desc->sqlvar; i < sql_desc->sqlld; i++, col++)
    {
        numchars = rtypwidth(col->sqltype, col->sqllen);
        printf("\t%-15s\t%d\n", col->sqlname, numchars);
    }
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if (sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}

```

Example Output

Column Name	# chars
order_num	11
order_date	10
customer_num	11
ship_instruct	40
backlog	1
po_num	10
ship_date	10
ship_weight	10
ship_charge	9
paid_date	10

Numeric-Formatting Routines

You can use special run-time functions to format a numeric expression according to a specific pattern. These formatting routines let you line up decimal points, right or left justify numbers, enclose negative numbers in parentheses, and perform other formatting functions.

The following functions are included in the libraries for formatting numeric expressions in `INFORMIX-ESQL/C`:

Function Name	Description
<code>rfmtdec</code>	Converts a decimal to a string
<code>rfmtdouble</code>	Converts a double to a string
<code>rfmtlong</code>	Converts a long integer to a string

The formatting routines for double and long values follow the description of general formatting rules. The formatting routine for decimal values is described in Chapter 4.

Formatting Numeric Strings

The numeric expression format string consists of combinations of the * & # < , . - + () \$ characters. The characters - + () \$ float. When a character floats, multiple leading occurrences of the character appear as a single character as far to the right as possible, without interfering with the number that is being displayed.

- * This character fills with asterisks any positions in the display field that would otherwise be blank.
- & This character fills with zeros any positions in the display field that would otherwise be blank.
- # This character does not change any blank positions in the display field. Use this character to specify a maximum width for a field.
- < This character causes the numbers in the display field to be left justified.
- ,
- .
- This character is a literal. It is displayed as a minus sign when *expr1* is less than zero. When you group several in a row, a single minus

- sign floats to the right-most position without interfering with the number being printed.
- + This character is a literal. It is displayed as a plus sign when *expr1* is greater than or equal to zero and as a minus sign when *expr1* is less than zero. When you group several in a row, a single plus sign floats to the right-most position without interfering with the number being printed.
 - (This character is a literal. It is displayed as a left parenthesis before a negative number. It is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. When you group several in a row, a single left parenthesis floats to the right-most position without interfering with the number being printed.
 -) This is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. A single one of these characters generally closes a format string that begins with a left parenthesis.
 - \$ This character is a literal. It is displayed as a dollar sign. When you group several in a row, a single dollar sign floats to the right-most position without interfering with the number being printed.

The following pages show example format strings for numeric expressions.

Example Format String

Format String	Numeric Value	Formatted Result
"#####"	0	bbbbbb
"&&&&&&"	0	00000
"\$\$\$\$\$\$"	0	bbbb\$
"*****"	0	*****
"<<<<<<"	0	(null string)
"##,###"	12345	12,345
"##,###"	1234	b1,234
"##,###"	123	bbb123
"##,###"	12	bbbb12
"##,###"	1	bbbbb1
"##,###"	-1	bbbbb1
"##,###"	0	bbbbbb
"&&&, &&&&"	12345	12,345
"&&&, &&&&"	1234	01,234
"&&&, &&&&"	123	000123
"&&&, &&&&"	12	000012
"&&&, &&&&"	1	000001
"&&&, &&&&"	-1	000001
"&&&, &&&&"	0	000000
"\$\$\$\$,\$\$\$\$"	12345	*****
"\$\$\$\$,\$\$\$\$"	1234	(overflow)
"\$\$\$\$,\$\$\$\$"	123	\$1,234
"\$\$\$\$,\$\$\$\$"	123	bb\$123
"\$\$\$\$,\$\$\$\$"	12	bbb\$12
"\$\$\$\$,\$\$\$\$"	1	bbbb\$1
"\$\$\$\$,\$\$\$\$"	-1	bbbb\$1
"\$\$\$\$,\$\$\$\$"	0	bbbbbb\$
"** ***/	12345	12,345
"** ***/	1234	*1,234
"** ***/	123	***123
"** ***/	12	****12
"** ***/	1	*****1
"** ***/	0	*****

This table uses the character b to represent a blank or space.

Example Format String

Format String	Numeric Value	Formatted Result
"###,###.###"	12345.67	12,345.67
"###,###.###"	1234.56	b1,234.56
"###,###.###"	123.45	bbb123.45
"###,###.###"	12.34	bbbb12.34
"###,###.###"	1.23	bbbbb1.23
"###,###.###"	0.12	bbbbbb.12
"###,###.###"	0.01	bbbbbb.01
"###,###.###"	-0.01	bbbbbb.01
"###,###.###"	-1	bbbbbb1.00
"&&&.&&&.&&&"	.67	12,345.67
"&&&.&&&.&&&"	1234.56	01,234.56
"&&&.&&&.&&&"	123.45	000123.45
"&&&.&&&.&&&"	0.01	000000.01
"\$\$\$\$.\$\$\$"	12345.67	*****
"\$\$\$\$.\$\$\$"	(overflow)	
"\$\$\$\$.\$\$\$"	1234.56	\$1,234.56
"\$\$\$\$.###"	0.00	\$.00
"\$\$\$\$.###"	1234.00	\$1,234.00
"\$\$\$\$.&&&"	0.00	\$.00
"\$\$\$\$.&&&"	1234.00	\$1,234.00
"-###,###.###"	-12345.67	-12,345.67
"-###,###.###"	-123.45	-bbb123.45
"-###,###.###"	-12.34	-bbbb12.34
"-#,###.###"	-12.34	-bbb12.34
"---,###.###"	-12.34	-bb12.34
"---,###.###"	-12.34	-12.34
"---,-#.###"	-1.00	-1.00
"-###,###.###"	12345.67	12,345.67
"-###,###.###"	1234.56	1,234.56
"-###,###.###"	123.45	123.45
"-###,###.###"	12.34	12.34
"-#,###.###"	12.34	12.34
"---,###.###"	12.34	12.34
"---,###.###"	12.34	12.34
"---,---.###"	1.00	1.00
"---,---.##"	-0.01	-0.01
"---,---.&&&"	-0.01	-0.01

This table uses the character b to represent a blank or space.

Example Format String

Format String	Numeric Value	Formatted Result
"\$\$\$,\$\$\$.&&"	-12345.67	-\$12,345.67
"\$\$\$,\$\$\$.&&"	-1234.56	-b\$1,234.56
"\$\$\$,\$\$\$.&&"	-123.45	-bbb\$123.45
"-\$\$\$,\$\$\$.&&"	-12345.67	-\$12,345.67
"-\$\$\$,\$\$\$.&&"	-1234.56	-\$1,234.56
"-\$\$\$,\$\$\$.&&"	-123.45	-bb\$123.45
"-\$\$\$,\$\$\$.&&"	-12.34	-bbb\$12.34
"-\$\$\$,\$\$\$.&&"	-1.23	-bbbb\$1.23
"----,-\$.&&"	-12345.67	-\$12,345.67
"----,-\$.&&"	-1234.56	-\$1,234.56
"----,-\$.&&"	-123.45	-\$123.45
"----,-\$.&&"	-12.34	-\$12.34
"----,-\$.&&"	-1.23	-\$1.23
"----,-\$.&&"	-12	-\$12
"\$***,***.&&"	12345.67	\$*12,345.67
"\$***,***.&&"	1234.56	\$*1,234.56
"\$***,***.&&"	123.45	\$***123.45
"\$***,***.&&"	12.34	\$****12.34
"\$***,***.&&"	1.23	\$*****1.23
"\$***,***.&&"	.12	\$*****.12
"(\$\$\$,\$\$\$.&&)"	-12345.67	(\$12,345.67)
"(\$\$\$,\$\$\$.&&)"	-1234.56	(b\$1,234.56)
"(\$\$\$,\$\$\$.&&)"	-123.45	(bbb\$123.45)
"((\$\$\$,\$\$\$.&&)"	-12345.67	(\$12,345.67)
"((\$\$\$,\$\$\$.&&)"	-1234.56	(\$1,234.56)
"((\$\$\$,\$\$\$.&&)"	-123.45	(bb\$123.45)
"((\$\$\$,\$\$\$.&&)"	-12.34	(bbb\$12.34)
"((\$\$\$,\$\$\$.&&)"	-1.23	(bbbb\$1.23)
"(((,((\$.&&)"	-12345.67	(\$12,345.67)
"(((,((\$.&&)"	-1234.56	(\$1,234.56)
"(((,((\$.&&)"	-123.45	(\$123.45)
"(((,((\$.&&)"	-12.34	(\$12.34)
"(((,((\$.&&)"	-1.23	(\$1.23)
"(((,((\$.&&)"	-12	(\$12)

This table uses the character b to represent a blank or space.

Example Format String

Format String	Numeric Value	Formatted Result
"(\$\$,,\$\$.&&)"	12345.67	\$12,345.67
"(\$\$,,\$\$.&&)"	1234.56	\$1,234.56
"(\$\$,,\$\$.&&)"	123.45	\$123.45
"((\$\$,,\$\$.&&)"	12345.67	\$12,345.67
"((\$\$,,\$\$.&&)"	1234.56	\$1,234.56
"((\$\$,,\$\$.&&)"	123.45	\$123.45
"((\$\$,,\$\$.&&)"	12.34	\$12.34
"((\$\$,,\$\$.&&)"	1.23	\$1.23
"(((,((\$.&&)"	12345.67	\$12,345.67
"(((,((\$.&&)"	1234.56	\$1,234.56
"(((,((\$.&&)"	123.45	\$123.45
"(((,((\$.&&)"	12.34	\$12.34
"(((,((\$.&&)"	1.23	\$1.23
"(((,((\$.&&)"	.12	\$.12
"<<<<<<<"	12345	12,345
"<<<<<<<"	1234	1,234
"<<<<<<<"	123	123
"<<<<<<<"	12	12

RFMTDOUBLE

Purpose

The **rfmtdouble** function converts a **double** in internal format to a character string formatted according to a pattern.

Syntax

```
int rfmtdouble(dvalue, format, outbuf)
    double dvalue;
    char *format;
    char *outbuf;
```

<i>dvalue</i>	is the number to be formatted.
<i>format</i>	is the address of the formatted string.
<i>outbuf</i>	is the address of the buffer to receive the format string.

Return Codes

0	The conversion was successful.
-1211	The program ran out of memory—memory-allocation error.
-1217	The format string is too large.

Example

```

/*
 * rfimtdouble.ec *

The following program applies a series of format specifications to a series
of doubles and displays the result of each format.
*/

#include <stdio.h>

double dbls[] =
{
    210203.204,
    4894,
    443.334899312,
    -12344455,
    0
};

char *formats[] =
{
    "#####",
    "<, <<, <<<, <<<<",
    "$$$$$$.##",
    "({&&, &&&, &&&.})",
    "$*****.**",
    0
};

char result[40];

main()
{
    int x;
    int i = 0, f;

    while(dbls[i]) /* for each number in dbls */
    {
        f = 0;
        while(formats[f]) /* format with each of formats[] */
        {
            if (x = rfimtdouble(dbls[i], formats[f], result))
            {
                printf("Error %d in formatting %g using %s\n",
                    x, dbls[i], formats[f]);
                break;
            }
            /*
             * Display each result and bump to next format (f++)
             */
            printf("\n\t[%g] using format [%s]: %s",
                dbls[i], formats[f], result);
            f++;
        }
        ++i; /* bump to next double */
        printf("\n"); /* separate result groups */
    }
}

```

Example Output

```
[210203] using format [#####]:          210203
[210203] using format [<, <<<, <<<, <<<]: 210,203
[210203] using format [$$$$$$$$$.##]:     $210203.20
[210203] using format [( &&, &&&, &&&. )]: 000210,203.
[210203] using format [$*****.**]: $**210203.20

[4894] using format [#####]:           4894
[4894] using format [<, <<<, <<<, <<<]: 4,894
[4894] using format [$$$$$$$$$.##]:      $4894.00
[4894] using format [( &&, &&&, &&&. )]: 000004,894.
[4894] using format [$*****.**]: $****4894.00

[443.335] using format [#####]:         443
[443.335] using format [<, <<<, <<<, <<<]: 443
[443.335] using format [$$$$$$$$$.##]:    $443.33
[443.335] using format [( &&, &&&, &&&. )]: 000000443.
[443.335] using format [$*****.**]: $*****443.33

[-1.23445e+07] using format [#####]:    12344455
[-1.23445e+07] using format [<, <<<, <<<, <<<]: 12,344,455
[-1.23445e+07] using format [$$$$$$$$$.##]: $12344455.00
[-1.23445e+07] using format [( &&, &&&, &&&. )]: (12,344,455.)
[-1.23445e+07] using format [$*****.**]: $*12344455.00
```

RFMTLONG

Purpose

The **rfmtlong** function converts a **long** in internal format to a character string formatted according to a pattern.

Syntax

```
int rfmtlong(lvalue, format, outbuf)
    long lvalue;
    char *format;
    char *outbuf;
```

<i>format</i>	is the address of the formatted string.
<i>lvalue</i>	is the number to be formatted.
<i>outbuf</i>	is the address of the buffer to receive the format string.

Return Codes

0	The conversion was successful.
-1211	The program ran out of memory—memory-allocation error.
-1217	The format string is too large.

Example

```

/*
 * rfmtlong.ec *

The following program applies a series of format specifications to a series
of longs and displays the result of each format.
*/

#include <stdio.h>

long lngs[] =
{
    21020304L,
    3334899312L,
    -3334899312L,
    -12344455L,
    0
};

char *formats[] =
{
    "#####",
    "$$$$$$$$$$.##",
    "(&, &&&, &&&, &&&.)",
    "<<<<, <<<, <<<, <<<",
    "$*****.**",
    0
};

char result[40];

main()
{
    int x;
    int s = 0, f;

    while(lngs[s]) /* for each long in lngs[] */
    {
        f = 0;
        while(formats[f]) /* format with each of formats[] */
        {
            if (x = rfmtlong(lngs[s], formats[f], result))
            {
                printf("Error %d in formatting %ld using %s.\n",
                    x, lngs[s], formats[f]);
                break;
            }
            /*
             * Display result and bump to next format (f++)
             */
            printf("\n\t[%d] using format [%s]: %s",
                lngs[s], formats[f], result);
            f++;
        }
        ++s; /* bump to next long */
        printf("\n"); /* separate display groups */
    }
}

```

Example Output

```
[21020304] using format [#####]:          21020304
[21020304] using format [$$$$$$$$$.##]:      $21020304.00
[21020304] using format [(, &&&, &&&, &&&.)]:    00021,020,304.
[21020304] using format [<<<<, <<<, <<<, <<<]: 21,020,304
[21020304] using format [$*****.**: $****21020304.00

[-960067984] using format [#####]:          960067984
[-960067984] using format [$$$$$$$$$.##]:      $960067984.00
[-960067984] using format [(, &&&, &&&, &&&.)]:    (00960,067,984.)
[-960067984] using format [<<<<, <<<, <<<, <<<]: 960,067,984
[-960067984] using format [$*****.**: $***960067984.00

[960067984] using format [#####]:          960067984
[960067984] using format [$$$$$$$$$.##]:      $960067984.00
[960067984] using format [(, &&&, &&&, &&&.)]:    00960,067,984.
[960067984] using format [<<<<, <<<, <<<, <<<]: 960,067,984
[960067984] using format [$*****.**: $***960067984.00

[-12344455] using format [#####]:          12344455
[-12344455] using format [$$$$$$$$$.##]:      $12344455.00
[-12344455] using format [(, &&&, &&&, &&&.)]:    (00012,344,455.)
[-12344455] using format [<<<<, <<<, <<<, <<<]: 12,344,455
[-12344455] using format [$*****.**: $****12344455.00
```

Working with Character and String Data Types

Chapter Overview	3
Character and String Functions	4
BYCMPR	5
BYCOPY	7
BYFILL	9
BYLENG	11
LDCHAR	13
RDOWNSHIFT	15
RSTOD	16
RSTOI	18
RSTOL	20
RUPSHIFT	22
STCAT	23
STCHAR	25
STCMPR	27
STCOPY	29
STLENG	30
Programming with a VARCHAR Data Type	32
Declaring a Host Variable for a VARCHAR Data Type	32
VARCHAR Macros	33



Chapter Overview

This chapter describes the Informix library functions for working with character data types and string data types that are included with **ESQL/C**. You can use these functions in your C programs to manipulate characters and strings of bytes and characters, including variable-length expressions. When you use a compiler shell script (**esql**), these functions are linked automatically to your program.

This chapter also tells you how to declare host variables for the **VARCHAR** data type and how to use the macros that are available for working with **varchar** data types.

For information about all of the data types available for use in an **ESQL/C** program, see Chapter 2 of this manual. For information about SQL data types, see Chapter 3 of *The Informix Guide to SQL: Reference*.

Character and String Functions

The following functions are contained in the Informix library. Those beginning with **by** act on and return fixed-length strings of bytes. Those beginning with **rst** and **st** (except **stchar**) operate on and return null-terminated strings.

Function name	Description
bncmp	Compares two groups of contiguous bytes
bycopy	Copies bytes from one area to another
byfill	Fills the specified area with a character
byleng	Counts the number of bytes in a string
ldchar	Copies a fixed-length string to a null-terminated string
rdownshift	Converts all letters to lowercase
rstod	Converts a string to double
rstoi	Converts a string to short
rstol	Converts a string to long
rupshift	Converts all letters to uppercase
stcat	Concatenates one string to another
stchar	Copies a null-terminated string to a fixed-length string
stcmp	Compares two strings
stcopy	Copies one string to another string
stleng	Counts the number of bytes in a string

BYCMPR

Purpose

The **bycmpr()** function compares two groups of contiguous bytes for a given length. It returns the result of the comparison as its value.

Syntax

```
int bycmpr(byte1, byte2, length)
    char *byte1;
    char *byte2;
    int length;
```

byte1 is a pointer to the starting location of the first group of contiguous bytes.

byte2 is a pointer to the starting location of the second group of contiguous bytes.

length is the number of bytes to be compared.

Usage

The **bycmpr()** function performs a byte-by-byte comparison of the two groups of contiguous bytes until a difference is found or *length* number of bytes have been compared. It returns an integer whose value, relative to 0 (zero), indicates the difference in value between the two groups of bytes.

The **bycmpr()** function accomplishes the comparison by subtracting the bytes of the *byte2* group from those of the *byte1* group.

Return Codes

=0	The two groups are identical.
<0	The <i>byte1</i> group < <i>byte2</i> group.
>0	The <i>byte1</i> group > <i>byte2</i> group.

Example

```
/*
 * bycmp.c
 *
 * The following program performs four different byte comparisons with
 * bycmp() and displays the results.
 */

#include <stdio.h>

main()
{
    int x;

    static char string1[] = "abcdef";
    static char string2[] = "abcdeg";

    static int number1 = 12345;
    static int number2 = 12367;

    static char string3[] = {0x00, 0x07, 0x45, 0x32, 0x00};
    static char string4[] = {0x00, 0x07, 0x45, 0x31, 0x00};

    /* strings */
    x = bycmp(string1, string2, sizeof(string1));
    printf("\tResult #1: %d\n", x);

    /* ints */
    x = bycmp(&number1, &number2, sizeof(number1));
    printf("\tResult #2: %d\n", x);

    /* non printable */
    x = bycmp(string3, string4, sizeof(string3));
    printf("\tResult #3: %d\n", x);

    x = bycmp(&string1[2], &string2[2], 2); /* bytes */
    printf("\tResult #4: %d\n", x);
}
```

Example Output

```
Result #1: -1
Result #2: -1
Result #3: 1
Result #4: 0
```

BYCOPY

Purpose

The **bycopy()** function copies a given number of bytes from one location to another.

Syntax

```
void bycopy(from, to, length)
    char *from;
    char *to;
    int length;
```

<i>from</i>	is a pointer to the starting byte of the group of bytes to be copied.
<i>length</i>	is the number of bytes to be copied.
<i>to</i>	is a pointer to the starting byte of the destination group of bytes.

Note: Take care not to overwrite areas of memory adjacent to the destination area.

Example

```
/*
 * bycopy.ec *

The following program shows the results of bycopy() for three copy
operations.
*/

#include <stdio.h>

char dest[20];

main()
{
    int number1 = 12345;
    int number2 = 0;
    static char string1[] = "abcdef";
    static char string2[] = "abcdefghijklmn";

    bycopy(string1, dest, strlen(string1));
    printf("\tResult #1: %s\n", dest);

    bycopy(string2, dest, strlen(string2));
    printf("\tResult #2: %s\n", dest);

    bycopy(&number1, &number2, 3);
    printf("\tResult #3: number1(hex) %08x, number2(hex) %08x\n",
           number1, number2);
}
```

Example Output

```
Result #1: abcdef
Result #2: abcdefghijklmn
Result #3: number1(hex) 00003039, number2(hex) 00003000
```

BYFILL

Purpose

The `byfill()` function fills a specified area with one character.

Syntax

```
void byfill(to, length, ch)
    char *to;
    int length;
    char ch;
```

ch is the character that fills the area.

length is the number of times the character is repeated within the area.

to is the starting byte of the memory area to be filled.

Note: Take care not to overwrite areas of memory adjacent to the area to be filled.

Example

```
/*
 * byfill.ec *

The following program shows the results of three byfill() operations on an
area that is initialized to x's.
*/

#include <stdio.h>

main()
{
    static char area[20] = "xxxxxxxxxxxxxxxxxxxx";

    byfill(area, 5, 's');
    printf("\tResult #1: %s\n", area);

    byfill(&area[16], 2, 's');
    printf("\tResult #2: %s\n", area);

    byfill(area, sizeof(area)-1, 'b');
    printf("\tResult #3: %s\n", area);
}
```

Example Output

```
Result #1: sssssxxxxxxxxxxxxxx  
Result #2: sssssxxxxxxxxxxxxssx  
Result #3: bbbbbbbbbbbbbbbbbbb
```

BYLENG

Purpose

The **byleng()** function returns the number of significant characters in a string, not counting trailing blanks.

Syntax

```
int byleng(from, count)
    char *from;
    int count;
```

count is the number of bytes in the fixed-length string.
from is a pointer to a fixed-length string (not null-terminated).

Example

```
/*
 * byleng.ec *

The following program uses byleng() to count the significant characters in
an area.
*/

#include <stdio.h>

main()
{
    int x;
    static char area[20] = "xxxxxxxxxx          ";

    x = byleng(area, 15); /* initial length */
    printf("\tResult #1: %d, area: '%s'\n", x, area);

    bcopy("ss", &area[16], 2);
    x = byleng(area, 19); /* after copy */
    printf("\tResult #2: %d, area: '%s'\n", x, area);
}
```

Example Output

```
Result #1: 10, area: 'xxxxxxxxx  '\nResult #2: 18, area: 'xxxxxxxxx  ss '
```

LDCHAR

Purpose

The **ldchar()** function copies a fixed-length string into a null-terminated string with any trailing blanks removed.

Syntax

```
void ldchar(from, count, to)
    char *from;
    int count;
    char *to;
```

<i>count</i>	is the number of bytes in the fixed-length source string.
<i>from</i>	is a pointer to the fixed-length source string.
<i>to</i>	is a pointer to the first byte of a null-terminated destination string.

Example

```
/*
 * ldchar.ec *

The following program loads characters to specific locations in an array
that is initialized to z's. It displays the result of each ldchar()
operation.
*/

#include <stdio.h>

main()
{
    static char src1[] = "abcd  ";
    static char src2[] = "abcd g ";
    static char dest[40];

    ldchar(src1, stleng(src1), dest);
    printf("\tSource: [%s]\n\tDest  : [%s]\n\n", src1, dest);

    ldchar(src2, stleng(src2), dest);
    printf("\tSource: [%s]\n\tDest  : [%s]\n\n", src2, dest);
}
```

Example Output

```
Source: [abcd  ]  
Dest  : [abcd]
```

```
Source: [abcd g ]  
Dest  : [abcd g]
```

RDOWNSHIFT

Purpose

The `rdownshift()` function changes all of the characters within a null-terminated string to lowercase.

Syntax

```
void rdownshift(s)
    char *s;
```

`s` is a pointer to a null-terminated string.

Example

```
/*
 * rdownshift.ec *
 *
 * The following program uses rdownshift() on a string containing alpha,
 * numeric and punctuation characters.
 */
#include <stdio.h>

main()
{
    static char string[] = "123ABCDEFHGHIJK'.";

    printf("\tInput string...: [%s]\n", string);
    rdownshift(string);
    printf("\tAfter downshift: [%s]\n", string);
}
```

Example Output

```
Input string...: [123ABCDEFHGHIJK'.]
After downshift: [123abcdefghijklmnop'.]
```

RSTOD

Purpose

The **rstod()** function converts a null-terminated string into a double.

Syntax

```
int rstod(string, double_val)
    char *string;
    double *double_val;
```

double_val is a pointer to a double where the result of the function is held.

string is a pointer to a null-terminated string.

Return Codes

=0 The conversion was successful.

!=0 The conversion failed.

Example

```
/*
 * rstod.ec *

The following program tries to convert three strings to doubles.
It displays the result of each attempt.
*/

#include <stdio.h>

main()
{
    int errnum;
    char *string1 = "1234567887654321";
    char *string2 = "12345678.87654321";
    char *string3 = "zzzzzzzzzzzzzzzz";
    double d;

    if ((errnum = rstod(string1, &d)) == 0)
        printf("\n\tResult #1: %f", d);
    else
        printf("\n\tError %d in conversion\n", errnum);

    if ((errnum = rstod(string2, &d)) == 0)
        printf("\n\tResult #2: %.8f", d);
    else
        printf("\n\tError %d in conversion\n", errnum);

    if ((errnum = rstod(string3, &d)) == 0)
        printf("\n\tResult #3: %.8f\n", d);
    else
        printf("\n\tError %d in conversion of string #3\n", errnum);
}
```

Example Output

```
Result #1: 1234567887654321.000000
Result #2: 12345678.87654321
Error -1213 in conversion of string #3
```

RSTOI

Purpose

The **rstoi()** function converts a null-terminated string into an integer.

Syntax

```
int rstoi(string, ival)
    char *string;
    int *ival;
```

ival is a pointer to an integer where the result of the function is held.

string is a pointer to a null-terminated string.

Usage

The legal range of values is from -32767 to 32767.

If *string* corresponds to a null integer, *ival* points to the representation for a SMALLINT NULL. If you want to convert a string that corresponds to a long integer, use **rstol**. Failure to do so can result in corrupt data representation.

Return Codes

=0 The conversion was successful.

!=0 The conversion failed.

Example

```
/*
 * rstoi.ec *

The following program tries to convert three strings to integers.
It displays the result of each conversion.
*/

#include <stdio.h>

#include sqltypes;

main()
{
    int err;
    int i;
    short s;

    i = 0;
    if((err = rstoi("abc", &i)) == 0)
        printf("\n\tResult #1: %d", i);
    else
        printf("\n\tError %d in conversion of string #1\n", err);

    i = 0;
    if((err = rstoi("32766", &i)) == 0)
        printf("\n\tResult #2: %d", i);
    else
        printf("\n\tError %d in conversion of string #2\n", err);

    i = 0;
    if((err = rstoi("", &i)) == 0)
    {
        s = i; /* assign to a SHORT variable */
        if (risnull(CSHORTTYPE, (char *) &s)) /* and then test for NULL */
            printf("\n\tResult #3 is NULL\n");
        else
            printf("\n\tResult #3: %d\n", i);
    }
    else
        printf("\n\tError %d in conversion of string #3\n", err);
}
```

Example Output

```
Error -1213 in conversion of string #1

Result #2: 32766
Result #3 is NULL
```

RSTOL

Purpose

The **rstol()** function converts a null-terminated string into a long integer.

Syntax

```
int rstol(string, long_int)
    char *string;
    long *long_int;
```

long_int is a pointer to a long integer where the result of the function is held.

string is a pointer to a null-terminated string.

Usage

The legal range of values is from -2,147,483,647 to 2,147,483,647.

Return Codes

=0 The conversion was successful.

!=0 The conversion failed.

Example

```
/*
 * rstol.ec *

The following program tries to convert three strings to longs. It
displays the result of each attempt.
*/

#include <stdio.h>

#include sqltypes;

main()
{
    int err;
    long l;

    l = 0;
    if((err = rstol("abc", &l)) == 0)
        printf("\n\tResult #1: %ld", l);
    else
        printf("\n\tError %d in conversion of string #1\n", err);

    l = 0;
    if((err = rstol("2147483646", &l)) == 0)
        printf("\n\tResult #2: %ld", l);
    else
        printf("\n\tError %d in conversion of string #2\n", err);

    l = 0;
    if((err = rstol("", &l)) == 0)
    {
        if(risnull(CLONGTYPE, (char *) &l))
            printf("\n\tResult #3: NULL\n", l);
        else
            printf("\n\tResult #3: %ld\n", l);
    }
    else
        printf("\n\tError %d in conversion of string #3\n", err);
}
```

Example Output

```
Error -1213 in conversion of string #1

Result #2: 2147483646
Result #3: NULL
```

RUPSHIFT

Purpose

The **rupshift()** function changes all of the characters within a null-terminated string to uppercase.

Syntax

```
void rupshift(s)
    char *s;
```

`s` is a pointer to a null-terminated string.

Example

```
/*
 * rupshift.ec *
 *
 * The following program displays the result of rupshift() on a string
 * of numbers, letters and punctuation.
 */
#include <stdio.h>

main()
{
    static char string[] = "123abcdefghijklmnopq.;";

    printf("\tInput string: %s\n", string);
    rupshift(string);
    printf("\tAfter upshift: %s\n", string); /* Result */
}
```

Example Output

```
Input string: 123abcdefghijklmnopq.;
After upshift: 123ABCDEFGHIJKLMN.;
```

STCAT

Purpose

The **stcat()** function concatenates one null-terminated string to the end of another.

Syntax

```
void stcat(s, dest)
    char *s, *dest;
```

<i>dest</i>	is a pointer to the start of the null-terminated destination string.
<i>s</i>	is a pointer to the start of the string that is placed at the end of the destination string.

Example

```
/*
 * stcat.ec *

The following program illustrates the use of stcat() function.
It prepares and executes a dynamic select statement,
using a value entered from the terminal.
*/

#include <stdio.h>

/*
 * Declare a variable large enough to hold
 * the select statement + the value for customer_num entered from the terminal.
 */
$char selstmt[80] = "select fname, lname from customer where customer_num = ";
char errmsg[400];

main()
{
    $char fname[16], lname[16];
    char custno[11];

    printf("\n\tEnter Customer#: ");
    gets(custno);

    /*
     * Add custno to "select statement"
     */
    stcat(custno, selstmt);
```

```
$database stores5;
err_chk("Open database");

$prepare stmt_1 from $selstmt;
err_chk("Prepare");

$declare c cursor for stmt_1;
err_chk("Declare cursor");

$open c;
err_chk("Open cursor");

$fetch c into :fname, :lname;
if (SQLCODE == SQLNOTFOUND)
printf("\n\tCustomer# %s does not exist in customer table\n", custno);
else
{
err_chk("Fetch");
printf("Customer#: %s is '%s %s'\n", custno, fname, lname);
}

$close c;
err_chk("Close cursor");

$close database;
err_chk("Close database");
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n",sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}
```

Example Output

```
Enter Customer#: 104
Customer#: 104 is 'Anthony          Higgins
```

STCHAR

Purpose

The **stchar()** function stores a null-terminated string in a fixed-length string, padding the end with blanks, if necessary.

Syntax

```
void stchar(from, to, count)
    char *from;
    char *to;
    int count;
```

<i>count</i>	is the number of bytes in the fixed-length destination string.
<i>from</i>	is a pointer to the first byte of a null-terminated source string.
<i>to</i>	is a pointer to the fixed-length destination string.

Example

```
/*
 * stchar.ec *
 *
 * The following program shows the blank padded result produced by
 * stchar() function.
 */
#include <stdio.h>

main()
{
    static char src[] = "start";
    static char dst[25] = "123abcdefghijklm.";

    printf("Source string: [%s]\n", src);
    printf("Destination string before stchar: [%s]\n", dst);

    stchar(src, dst, sizeof(dst) - 1);

    printf("Destination string after stchar: [%s]\n", dst);
}
```

Example Output

```
Source string: [start]
Destination string before stchar: [123abcdefghijklmnopq;.]
Destination string after stchar: [start ]
```

STCMPR

Purpose

The `stcmp()` function compares two null-terminated strings.

Syntax

```
int stcmp(s1, s2)
    char *s1, *s2;
```

s1 is a pointer to the first null-terminated string.

s2 is a pointer to the second null-terminated string.

Note: s1 is greater than s2 when s1 appears after s2 in the ASCII collating sequence.

Return Codes

=0 The two strings are identical.

<0 The first string is less than the second string.

>0 The first string is greater than the second string.

Example

```
/*
 * stcmp.ec *

The following program displays the results of stcmp() on three string
comparisons.
*/

#include <stdio.h>

main()
{
    printf("\n\tResult #1: %d", stcmp("aaa", "aaa")); /* equal */
    printf("\n\tResult #2: %d", stcmp("aaa", "aaaa")); /* less */
    printf("\n\tResult #3: %d\n", stcmp("bbb", "aaaa")); /* greater */
}
```

Example Output

```
Result #1: 0  
Result #2: -1  
Result #3: 1
```

STCOPY

Purpose

The `stcopy()` function copies a null-terminated string from one location in memory to another location.

Syntax

```
void stcopy(from, to)
    char *from, *to;
```

from is a pointer to the null-terminated string to be copied.
to is a pointer to a location in memory where the string is copied.

Example

```
/*
 * stcopy.ec *
 *
 * The program copies "John Doe" to an address and displays the result.
 */
#include <stdio.h>

main()
{
    static char string[] = "abcdefghijklmnopqrstuvwxy";

    printf("\n\tstring: [%s]", string); /* display dest */
    stcopy("John Doe", &string[15]); /* copy */
    printf("\n\tstring: %s\n", string); /* display string */
}
```

Example Output

```
string: [abcdefghijklmnopqrstuvwxy]
string: abcdefghijklmnoJohn Doe
```

STLENG

Purpose

The **stleng()** function returns the length, in bytes, of a specified null-terminated string.

Syntax

```
int stleng(string)
    char *string;
```

string is a pointer to a null-terminated string.

Usage

The length does not include the terminating null.

Example

```
/*
 * stleng.ec *

The following program uses stleng to display cat_advert from the catalog
table where cat_advert is less than 35 characters.
*/

#include <stdio.h>

char errmsg[400];

$varchar cat_advert[256];
$long cat_num;

main()
{
    int length;

    printf("\nPrinting advertisement text less than 35 characters long\n");

    $database stores5; /* open the stores5 database */
    err_chk("Open database");

    $declare curs cursor for /* setup cursor for select */
    select catalog_num, cat_advert from catalog;
    err_chk("Declare cursor");

    $open curs;
    err_chk("Open cursor");
```

```
while(getrow())
{
if ((length = stleng(cat_advert)) < 35)
    printf("\n\tAdvertisement for %ld (%d bytes):\n\t'%s'\n",
        cat_num, length, cat_advert);
}
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n",sqlca.sqlcode, name, errmsg);
        exit(1);
    }
}

/* fetch the next row for selected items */

getrow()
{
    $fetch curs into $cat_num, $cat_advert;
    if (SQLCODE == SQLNOTFOUND)
        return(0);
    err_chk("FETCH");
    return(1);
}
```

Example Output

Printing advertisement text less than 35 characters long

```
Advertisement for 10001 (34 bytes):
'Your First Season's Baseball Glove'

Advertisement for 10025 (31 bytes):
'ProCycle Stem with Pearl Finish'

Advertisement for 10047 (28 bytes):
'Long Drive Golf Balls: White'

Advertisement for 10058 (29 bytes):
'Athletic Watch w/4-Lap Memory'

Advertisement for 10068 (22 bytes):
'High-Quality Kickboard'

Advertisement for 10072 (27 bytes):
'Team Logo Silicone Swim Cap'
```

Programming with a VARCHAR Data Type

This section covers the following topics:

- Declaring host variables for VARCHAR data types
- Using the VARCHAR macros

See Chapter 2 of this manual for a description of the VARCHAR data type and information on data type conversion.

Declaring a Host Variable for a VARCHAR Data Type

Use the following syntax to declare a host variable for a VARCHAR data type:

```
$varchar host_name[n];
```

host_name is an identifier, the name of the variable.

n specifies the maximum length of the variable. It should be the *max-size* from the column declaration plus one to account for the null terminator in the array.

varchar specifies the data type.

To conform to ANSI standards, use the EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION statements instead of the dollar sign (\$). For example:

```
EXEC SQL BEGIN DECLARE SECTION
    varchar host_name[n];
EXEC SQL END DECLARE SECTION
```

INFORMIX-ESQL/C represents a VARCHAR value in storage as an array of characters. If the contents of a **varchar** host variable are stored in a table, the data is not padded with extra blanks. If the array contains explicit trailing blanks, they are retained when the value is stored in the database.

Alternatively, you can use the following syntax to declare a host variable for a VARCHAR data type:

```
$string host_name[n];
```

If you declare a host variable of type **string**, it is also an array of characters. However, unlike values that are read into host variables declared as type **varchar**, if the column contains explicit (user-entered) trailing blanks, the blanks are stripped when the value is read into the **string** host variable.

If you read values from a VARCHAR database column into a **char** or **fixchar** host variable, user-entered spaces are retained as they are with the **varchar** host variable.

When you read a value from any type of host variable into a VARCHAR database column, all user-entered blanks are retained.

VARCHAR Macros

VARCHAR macros in the **varchar.h** include file allow you to use encoded size information about a **varchar**. The size information is stored in the **LENGTH** field of the system descriptor area (or the **sqlen** field of an **sqlda** structure) after a describe. The size information for a VARCHAR column is stored in the **syscolumns** system catalog table. The names of the macros and their use are as follows:

VLENGTH(size)	Determines how large to make the host array for a VARCHAR data type.
VCMIN(size)	Determines the minimum length of a VARCHAR data type from the encoded value in the syscolumns system catalog table.
VCMAX(size)	Determines the maximum length of a VARCHAR data type from the encoded value in the syscolumns system catalog table.
VCSIZ(max, min)	Encodes the maximum and minimum sizes.

The macros are defined as follows:

```
#define VLENGTH(len)      (VCMAX(len)+1)
#define VCMIN(size)      (((size) >> 8) & 0x00ff)
#define VCMAX(size)      ((size) & 0x00ff)
#define VCSIZ(max, min)  (((min) << 8) & 0xff00) + ((max) & 0x00ff)
```

The following example obtains **collength** from the **syscolumns** system catalog table for the **cat_advert** column. It then uses the macros from **varchar.h** to display the maximum size (VCMAX), the minimum size (VCMIN), the host variable length(VLENGTH), and the encoded value of the combined maximum size and minimum size (VCSIZ) for **cat_advert**.

Example

```
$include varchar.h;

$int   vc_size;
int    vc_code;
int    max, min;
int    hv_length;

char  errmsg[512];

main()
{
    $database stores5;
    err_chk("database");
    $select collength into $vc_size from syscolumns
        where colname = "cat_advert";
    err_chk("select");

    max = VCMAX(vc_size);
    printf("\n\tmax: %d", max);

    min = VCMIN(vc_size);
    printf("\n\tmin: %d", min);

    hv_length = VCLENGTH(vc_size);
    printf("\n\tlv_length: %d", hv_length);

    vc_code = VCSIZ(max, min);
    printf("\n\tvc_code: %d\n", vc_code);
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode < 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
        exit(1);
    }
    return((sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1);
}
```

Example Output

```
max: 255
min: 65
lv_length: 256
vc_code: 16895
```

Working with the DECIMAL Data Type

Chapter Overview 3

The DECIMAL Data Type 3

Decimal Type Functions 5

DECCVASC 6

DECTOASC 9

DECCVINT 12

DECTOINT 14

DECCVLONG 16

DECTOLONG 18

DECCVDBL 20

DECTODBL 22

DECADD, DECSUB, DECMUL, and DECDIV 25

DECCMP 31

DECCOPY 33

DECECVT and DECFCVT 35

DECROUND 41

DECTRUNC 43

RFMTDEC 45



Chapter Overview

This chapter contains the following information on how to use DECIMAL data type values in an **INFORMIX-ESQL/C** program:

- An overview of the DECIMAL data type
- A sample program that uses the DECIMAL data type functions
- The syntax for functions that you can use to manipulate DECIMAL data types.

For information about all of the data types available for use in an **ESQL/C** program, see Chapter 2 of this manual. For information about SQL data types, see Chapter 3 of *The Informix Guide to SQL: Reference*.

The DECIMAL Data Type

The DECIMAL data type is a machine-independent method for representing numbers of up to 32 significant digits, with or without a decimal point, and with exponents in the range -128 to +126.

INFORMIX-ESQL/C provides routines that facilitate conversion of DECIMAL data type numbers to and from every data type allowed in the C language.

When you define a column as having the **DECIMAL(*m*,*n*)** data type, it has a total of *m* (≤ 32) significant digits (the precision) and *n* ($\leq m$) digits to the right of the decimal point (the scale). For a complete description of the DECIMAL data type, see Chapter 3 of *The Informix Guide to SQL: Reference*.

When used within a program, DECIMAL type numbers are stored in a C structure of the following type:

```
#define DECSIZE 16

struct decimal
{
    short dec_exp;
    short dec_pos;
    short dec_ndgts;
    char  dec_dgts [DECSIZE];
};

typedef struct decimal dec_t;
```

The **decimal** structure and the typedef **dec_t** are found in the **decimal.h** header file. Include this file in all C source files that use any of the decimal functions:

```
#include <decimal.h>
```

The **decimal** structure has four parts:

dec_exp	holds the exponent of the normalized decimal type number. This exponent represents a power of 100.
dec_pos	holds the sign of the decimal type number (1 when the number is zero or greater; 0 when the number is less than zero).
dec_ndgts	contains the number of base 100 significant digits of the decimal type number.
dec_dgts	is a character array that holds the significant digits of the normalized decimal type number, which is dec_dgts[0] != 0 . Each character in the array is a one-byte binary number in base 100. The dec_ndgts array contains the number of significant digits in dec_dgts .

All operations on **decimal** type numbers take place through the functions provided in **ESQL/C** and are described in the following section. Any other operations, modifications, or analyses can produce unpredictable results.

Decimal Type Functions

The following C function calls are available in **INFORMIX-ESQL/C** to treat **DECIMAL** data type numbers:

Function name	Description
deccvasc	Converts C char type values to decimal type values
dectoasc	Converts decimal type values to C char type values
deccvint	Converts C int type values to decimal type values
dectoint	Converts decimal type values to C int type values
deccvlong	Converts C long type values to decimal type values
dectolong	Converts decimal type values to C long type values
deccvdbl	Converts C double type values to decimal type values
dectodbl	Converts decimal type values to C double type values
decadd	Adds two decimal numbers
decsub	Subtracts two decimal numbers
decmul	Multiplies two decimal numbers
decdiv	Divides two decimal numbers
deccmp	Compares two decimal numbers
deccopy	Copies a decimal number
decevt	Converts a DECIMAL value to an ASCII string
decfcvt	Converts a DECIMAL value to an ASCII string
decround	Rounds a decimal number
dectrunc	Truncates a decimal number
rfmtdec	Converts a decimal type value to a formatted character string.

The syntax of each of these functions is described in the remainder of this chapter.

DECCVASC

Purpose

The **deccvasc** function converts a value held as printable characters in a C **char** type into a **decimal** type number.

Syntax

```
int deccvasc(cp, len, np)
    char *cp;
    int len;
    dec_t *np;
```

<i>cp</i>	is a pointer to a string that holds the value to be converted.
<i>len</i>	is the length of the string.
<i>np</i>	is a pointer to the decimal structure where the result of the conversion is placed.

Usage

Leading spaces in the character string are ignored.

The character string can have a leading sign, either a plus (+) or minus (-), a decimal point, and digits to the right of the decimal point.

The character string can contain an exponent preceded by either *e* or *E*. The exponent can be preceded by a sign, either a plus (+) or minus (-).

Return Codes

0	The conversion was successful.
-1200	The number is too large to fit into a decimal type (overflow).
-1201	The number is too small to fit into a decimal type (underflow).
-1213	The string has non-numeric characters.
-1216	The string has a bad exponent.

Example

```

/*
 * deccvasc.ec *

The following program converts two strings to decimal numbers and displays
the values stored in each field of the decimal structures.
*/

#include <stdio.h>

#include decimal;

char string1[] = "-12345.6789";
char string2[] = "480";

main()
{
    int x;
    dec_t num1, num2;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    /*
     * Display the exponent, sign value and number of digits in num1.
     */
    printf("\tstring1: %s\n", string1);
    disp_dec("num1", &num1);

    /*
     * Display the exponent, sign value and number of digits in num2.
     */
    printf("\tstring2: %s\n", string2);
    disp_dec("num2", &num2);
    exit(0);
}

disp_dec(s, num)
dec_t *num;
{
    int n;

    printf("\t%s: dec_exp: %d, dec_pos: %d, dec_ndgts: %d, dec_dgts: ",
s, num->dec_exp, num->dec_pos, num->dec_ndgts);
    n = 0;
    while(n < num->dec_ndgts)
        printf("%02d ", num->dec_dgts[n++]);
    printf("\n\n");
}

```

Example Output

```
string1: -12345.6789
num1: dec_exp: 3, dec_pos: 0, dec_ndgts: 5, dec_dgts: 01 23 45 67 89

string2: 480
num2: dec_exp: 2, dec_pos: 1, dec_ndgts: 2, dec_dgts: 04 80
```

DECTOASC

Purpose

The **dectoasc** function converts a **decimal** type number to an ASCII string.

Syntax

```
int dectoasc(np, cp, len, right)
    dec_t *np;
    char *cp;
    int len;
    int right;
```

<i>cp</i>	is a pointer to the beginning of the character buffer to hold the ASCII string.
<i>len</i>	is the maximum length in bytes of the string buffer.
<i>np</i>	is a pointer to the decimal structure whose associated decimal value is converted into an ASCII string.
<i>right</i>	is an integer indicating the number of decimal places to the right of the decimal point.

Usage

If *right* = -1, the number of decimal places is determined by the decimal value of **np*.

If the number does not fit into a character string of length *len*, **dectoasc** converts the number to an exponential notation. If the number still does not fit, the string is filled with asterisks. If the number is shorter than the string, it is left justified and padded on the right with blanks.

Because the ASCII string returned by **dectoasc** is not null-terminated, your program must add a null character to the string before printing it.

Return Codes

0	The conversion was successful.
-1	The conversion failed.

Example

```
/*
/*
 * dectoasc.ec *

The following program converts decimal numbers to strings of varying sizes.
*/

#include <stdio.h>

#include decimal;

#define END sizeof(result)

char string1[] = "-12345.038782";
char string2[] = "480";
char result[40];

main()
{
    int x;
    dec_t num1, num2;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }

    if (x = dectoasc(&num1, result, 5, -1))
        printf("Error %d in converting decimal1 to string\n", x);
    else
    {
        result[5] = '\0'; /* null terminate */
        printf("\tResult #1: '%s'\n", result);
    }

    if (x = dectoasc(&num1, result, 10, -1))
        printf("Error %d in converting decimal1 to string\n", x);
    else
    {
        result[10] = '\0'; /* null terminate */
        printf("\tResult #2: '%s'\n", result);
    }

    if (x = dectoasc(&num2, result, END, 3))
        printf("Error %d in converting decimal2 to string\n", x);
    else
    {
        result[END-1] = '\0'; /* null terminate */
        printf("\tResult #3: '%s'\n", result);
    }
}
```

Example Output

```
Error -1 in converting decimal1 to string
Result #2: '-12345.039'
Result #3: '480.000'
```

DECCVINT

Purpose

The **deccvint** function converts a C type **int** into a **decimal** type number.

Syntax

```
int deccvint(integer, np)
           int integer;
           dec_t *np;
```

integer is the integer to be converted.

np is a pointer to a **decimal** structure where the result is placed.

Return Codes

0 The conversion was successful.

<0 The conversion failed.

Example

```
/*
 * deccvint.ec *

The following program converts two integers to decimal numbers and displays
the results.
*/

#include <stdio.h>

#include decimal;

char result[40];

main()
{
    int x;
    dec_t num;

    if (x = deccvint(129449233, &num))
    {
        printf("Error %d in converting int1 to decimal\n", x);
        exit(1);
    }
    if (x = dectoasc(&num, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal to string\n", x);
        exit(1);
    }
    printf("\tResult #1: %s\n", result);

    if (x = deccvint(33, &num))
    {
        printf("Error %d in converting int2 to decimal\n", x);
        exit(1);
    }
    if (x = dectoasc(&num, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal to string\n", x);
        exit(1);
    }
    printf("\tResult #2: %s\n", result);
    exit(0);
}
```

Example Output

```
Result #1: 129449233.0
Result #2: 33.0
```

DECTOINT

Purpose

The **dectoint** function converts a **decimal** type number into a C type **int**.

Syntax

```
int dectoint(np, ip)
            dec_t *np;
            int *ip;
```

<i>ip</i>	is a pointer to the integer.
<i>np</i>	is a pointer to a decimal structure whose value is converted to an integer.

Return Codes

0	The conversion was successful.
<0	The conversion failed.
-1200	The magnitude of the decimal type number > 32767.

Example

```
/*
 * dectoint.ec *

The following program converts two decimal numbers to ints and displays
the result of each conversion.
*/

#include <stdio.h>

#include decimal;

char string1 [] = "32767";
char string2 [] = "32768";

main()
{
    int x;
    int n = 0;
    dec_t num;

    if (x = deccvasc(string1, strlen(string1), &num))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = dectoint(&num, &n))
    {
        printf("Error %d in converting decimal to int\n", x);
        exit(1);
    }
    printf("\tResult #1: %d\n", n);

    if (x = deccvasc(string2, strlen(string2), &num))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = dectoint(&num, &n))
    {
        printf("Error %d in converting decimal to int\n", x);
        exit(1);
    }
    printf("\tResult #2: %d\n", n);
    exit(0);
}
```

Example Output

```
Result #1: 32767
Error -1200 in converting decimal to int
```

DECCVLONG

Purpose

The **deccvlong** function converts a C type **long** value into a **decimal** type number.

Syntax

```
int deccvlong(lng, np)  
    long lng;  
    dec_t *np;
```

<i>lng</i>	is the long value that is converted into a decimal type value.
<i>np</i>	is a pointer to a decimal structure that holds the decimal type number.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * deccvlong.ec *

The following program converts two longs to decimal numbers and displays
the results.
*/

#include <stdio.h>

#include decimal;

char result[40];

main()
{
    int x;
    dec_t num;
    long n;

    if (x = deccvlong(129449233L, &num))
    {
        printf("Error %d in converting long to decimal\n", x);
        exit(1);
    }
    if (x = dectoasc(&num, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal to string\n", x);
        exit(1);
    }
    printf("\tResult #1: %s\n", result);

    n = 2147483646; /* set n */
    if (x = deccvlong(n, &num))
    {
        printf("Error %d in converting long to decimal\n", x);
        exit(1);
    }
    if (x = dectoasc(&num, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal to string\n", x);
        exit(1);
    }
    printf("\tResult #2: %s\n", result);
    exit(0);
}
```

Example Output

```
Result #1: 129449233.0
Result #2: 2147483646.0
```

DECTOLONG

Purpose

The **dectolong** function converts a **decimal** type number into a C type **long**.

Syntax

```
int dectolong(np, lngp)  
    dec_t *np;  
    long *lngp;
```

lngp is a pointer to a **long** integer where the result of the conversion is placed.

np is a pointer to a **decimal** structure.

Return Codes

0 The conversion was successful.

-1200 The magnitude of the **decimal** type number > 2,147,483,647.

Example

```
/*
 * dectolong.ec *

The following program converts two decimal numbers to longs and displays
the return value and the result for each conversion.
*/

#include <stdio.h>

#include decimal;

char string1[] = "2146382012";
char string2[] = "3238299493";

main()
{
    int r = 0;
    long n = 0;
    dec_t num;

    deccvasc(string1, strlen(string1), &num);          /* string to decimal */
    r = dectolong(&num, &n);                          /* decimal to long */
    /*
    display result
    */
    printf("\n\tReturn #1: %05.d, Result #1: %ld", r, n);
    n = 0;                                           /* clear n */
    deccvasc(string2, strlen(string2), &num);        /* string to decimal */
    r = dectolong(&num, &n);                          /* decimal to long */
    /*
    display result
    */
    printf("\n\tReturn #2: %05.d, Result #2: %ld", r, n);
}

```

Example Output

```
Return #1: 00000, Result #1: 2146382012
Return #2: -1200, Result #2: 0

```

DECCVDBL

Purpose

The `deccvdbl` function converts a C type **double** into a **decimal** type number.

Syntax

```
int deccvdbl(dbl, np)
           double dbl;
           dec_t *np;
```

<i>dbl</i>	is the double value that is converted into a decimal type value.
<i>np</i>	is a decimal structure that contains the decimal type number.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * deccvdbl.ec *

The following program converts two double type numbers to decimal numbers
and displays the results.
*/

#include <stdio.h>

#include decimal;

char result[40];

main()
{
    int x;
    dec_t num;
    double d = 2147483647;

    if (x = deccvdbl((double)1234.5678901234, &num))
    {
        printf("Error %d in converting double1 to decimal\n", x);
        exit(1);
    }
    if (x = dectoaasc(&num, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal1 to string\n", x);
        exit(1);
    }
    printf("\tResult #1: %s\n", result);

    if (x = deccvdbl(d, &num))
    {
        printf("Error %d in converting double2 to decimal\n", x);
        exit(1);
    }
    if (x = dectoaasc(&num, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal2 to string\n", x);
        exit(1);
    }
    printf("\tResult #2: %s\n", result);
    exit(0);
}
```

Example Output

```
Result #1: 1234.5678901234
Result #2: 2147483647.0
```

DECTODBL

Purpose

The **dectodbl** function converts a **decimal** type number into a **double**.

Syntax

```
int dectodbl(np, dbl)  
    dec_t *np;  
    double *dbl;
```

dbl is a pointer to a **double** where the result of the conversion is placed.

np is a pointer to a **decimal** structure.

Usage

Depending on the floating-point format of the host machine, the conversion of a **decimal** type number to a **double** can result in the loss of precision.

Return Codes

0 The conversion was successful.
<0 The conversion failed.

Example

```
/*
 * dectodbl.ec *

The following program converts two decimal numbers to doubles and displays
the results.
*/

#include <stdio.h>

#include decimal;

char string1[] = "2949.3829398204382";
char string2[] = "3238299493";
char result[40];

main()
{
    int x;
    double d = 0;
    dec_t num;

    if (x = deccvasc(string1, strlen(string1), &num))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = dectodbl(&num, &d))
    {
        printf("Error %d in converting decimal1 to double\n", x);
        exit(1);
    }
    printf("\t String 1: %s\n", string1);
    printf("\tDouble val: %.15f\n", d);

    if (x = deccvasc(string2, strlen(string2), &num))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = dectodbl(&num, &d))
    {
        printf("Error %d in converting decimal2 to double\n", x);
        exit(1);
    }
    printf("\t String 2: %s\n", string2);
    printf("\tDouble val: %f\n", d);
    exit(0);
}
```

Example Output

```
String 1: 2949.3829398204382
Double val: 2949.382939820438423
String 2: 3238299493
Double val: 3238299493.000000
```

DECADD, DECSUB, DECMUL, and DECDIV

Purpose

The **decadd**, **decsub**, **decmul**, and **decdiv** arithmetic functions take pointers to three **decimal** structures as parameters. The first two **decimal** structures hold the operands of the arithmetic function. The third **decimal** structure is where the result is placed.

Syntax

```
int decadd(n1, n2, result)/* result = n1 + n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;
```

```
int decsub(n1, n2, result)/* result = n1 - n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;
```

```
int decmul(n1, n2, result)/* result = n1 * n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;
```

```
int decdiv(n1, n2, result)/* result = n1 / n2 */
    dec_t *n1;
    dec_t *n2;
    dec_t *result;
```

<i>n1</i>	is a pointer to the decimal structure of the first operand.
<i>n2</i>	is a pointer to the decimal structure of the second operand.
<i>result</i>	is a pointer to the decimal structure of the result of the operation.

Usage

The result can be the same as either *n1* or *n2*.

Return Codes

0	The operation was successful.
-1200	The operation resulted in overflow.
-1201	The operation resulted in underflow.
-1202	The operation attempted to divide by zero.

DECADD Example

```
/*
 * decadd.ec *

The following program obtains the sum of two decimal numbers.
*/

#include <stdio.h>

#include decimal;

char string1[] = " 1000.6789"; /* leading spaces will be ignored */
char string2[] = "80";
char result[40];

main()
{
    int x;
    dec_t num1, num2, sum;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = decadd(&num1, &num2, &sum))
    {
        printf("Error %d in adding decimals\n", x);
        exit(1);
    }
    if (x = dectoasc(&sum, result, sizeof(result), -1))
    {
        printf("Error %d in converting decimal result to string\n", x);
        exit(1);
    }
    printf("\t%s + %s = %s\n", string1, string2, result); /* display result */
    exit(0);
}
```

DECADD Example Output

1000.6789 + 80 = 1080.6789

DECSUB Example

```
/*
 * decsub.ec *

The following program subtracts two decimal numbers and displays the result.
*/

#include <stdio.h>

#include decimal;

char string1[] = "1000.038782";
char string2[] = "480";
char result[40];

main()
{
    int x;
    dec_t num1, num2, diff;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = decsub(&num1, &num2, &diff))
    {
        printf("Error %d in subtracting decimals\n", x);
        exit(1);
    }
    if (x = dectoaasc(&diff, result, sizeof(result), -1))
    {
        printf("Error %d in converting result to string\n", x);
        exit(1);
    }
    printf("\t%s - %s = %s\n", string1, string2, result);
    exit(0);
}
```

DECSUB Example Output

```
1000.038782 - 480 = 520.038782
```

DECMUL Example

```
/*
 * decmul.ec *

The decmul.ec program multiplies two decimal numbers and displays the result.
*/

#include <stdio.h>

#include decimal;

char string1[] = "80.2";
char string2[] = "6.0";
char result[40];

main()
{
    int x;
    dec_t num1, num2, mpx;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = decmul(&num1, &num2, &mpx))
    {
        printf("Error %d in converting multiply\n", x);
        exit(1);
    }
    if (x = dectasc(&mpx, result, sizeof(result), -1))
    {
        printf("Error %d in converting mpx to display string\n", x);
        exit(1);
    }
    printf("\t%s * %s = %s\n", string1, string2, result);
    exit(0);
}
```

DECMUL Example Output

```
80.2 * 6.0 = 481.2
```

DECDIV Example

```
/*
 * decdiv.ec *

The following program divides two decimal numbers and displays the result.
*/

#include <stdio.h>

#include decimal;

char string1[] = "480";
char string2[] = "80";
char result[40];

main()
{
    int x;
    dec_t num1, num2, dvd;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = decdiv(&num1, &num2, &dvd))
    {
        printf("Error %d in converting divide num1 by num2\n", x);
        exit(1);
    }
    if (x = dectasc(&dvd, result, sizeof(result), -1))
    {
        printf("Error %d in converting dividend to string\n", x);
        exit(1);
    }
    printf("\t%s / %s = %s\n", string1, string2, result);
    exit(0);
}
```

DECDIV Example Output

480 / 80 = 6.0

DECCMP

Purpose

The **deccmp** function compares two **decimal** type numbers.

Syntax

```
int deccmp(n1, n2)
    dec_t *n1;
    dec_t *n2;
```

n1 is a pointer to a **decimal** structure of the first number.

n2 is a pointer to a **decimal** structure of the second number.

Return Codes

-1	The first value is less than the second value.
0	The two values are identical.
1	The first value is greater than the second value.
DECUNKNOWN	Either value is null.

Example

```
/*
 * deccmp.ec *

The following program compares decimal numbers and displays the results.
*/

#include <stdio.h>

#include decimal;

char string1[] = "-12345.6789"; /* leading spaces will be ignored */
char string2[] = "12345.6789";
char string3[] = "-12345.6789";
char string4[] = "-12345.6780";

main()
{
    int x;
    dec_t num1, num2, num3, num4;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string2, strlen(string2), &num2))
    {
        printf("Error %d in converting string2 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string3, strlen(string3), &num3))
    {
        printf("Error %d in converting string3 to decimal\n", x);
        exit(1);
    }
    if (x = deccvasc(string4, strlen(string4), &num4))
    {
        printf("Error %d in converting string4 to decimal\n", x);
        exit(1);
    }

    printf("\tResult #1: %d\n", deccmp(&num1, &num2));
    printf("\tResult #2: %d\n", deccmp(&num2, &num3));
    printf("\tResult #3: %d\n", deccmp(&num1, &num3));
    printf("\tResult #4: %d\n", deccmp(&num3, &num4));
    exit(0);
}
```

Example Output

```
Result #1: -1
Result #2: 1
Result #3: 0
Result #4: -1
```

DECCOPY

Purpose

The **deccopy** function copies one **decimal** structure to another.

Syntax

```
void deccopy(n1, n2)  
    dec_t *n1;  
    dec_t *n2;
```

n1 is a pointer to the value held in the source **decimal** structure.

n2 is a pointer to the destination **decimal** structure.

Example

```
/*
 * deccopy.ec *

 The following program copies one decimal number to another.
*/

#include <stdio.h>

#include decimal;

char string1[] = "12345.6789";
char result[40];

main()
{
    int x;
    dec_t num1, num2;

    if (x = deccvasc(string1, strlen(string1), &num1))
    {
        printf("Error %d in converting string1 to decimal\n", x);
        exit(1);
    }
    deccopy(&num1, &num2);
    if (x = dectoasc(&num2, result, sizeof(result), -1))
    {
        printf("Error %d in converting num2 to string\n", x);
        exit(1);
    }
    printf("\t%s\n", result);
    exit(0);
}
```

Example Output

```
12345.6789
```

DECECVT and DECFCVT

Purpose

The **dececvt** and **decfcvt** functions are analogous to the subroutines under ECVT(3) in section three of the *UNIX Programmer's Manual*. The **dececvt** function works in the same fashion as **ecvt(3)**, and the **decfcvt** function works in the same fashion as **fcvt(3)**. They both convert a DECIMAL value to an ASCII string.

Syntax

```
char *dececvt(np, ndigit, decpt, sign)
    dec_t *np;
    int ndigit;
    int *decpt;
    int *sign;
```

```
char *decfcvt(np, ndigit, decpt, sign)
    dec_t *np;
    int ndigit;
    int *decpt;
    int *sign;
```

<i>decpt</i>	is a pointer to an integer that is the position of the decimal point relative to the beginning of the string. A negative value for <i>*decpt</i> means to the left of the returned digits.
<i>ndigit</i>	is the length of the ASCII string for dececvt . It is the number of digits to the right of the decimal point for decfcvt .
<i>np</i>	is a pointer to a decimal structure containing the DECIMAL value to be converted.
<i>sign</i>	is a pointer to the sign of the result. If the sign of the result is negative, <i>*sign</i> is nonzero; otherwise, it is zero.

Usage

The **dececvt** function converts the DECIMAL value pointed to by *np* into a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string.

The low-order digit is rounded.

The **decfcvt** function is identical to **dececv**, except that *ndigit* specifies the number of digits to the right of the decimal point instead of the total number of digits.

Let *np* point to 12345.67 and suppress all arguments except *ndigit*:

dececv(4)	=	"1235"	*decpt = 5
dececv(10)	=	"1234567000"	*decpt = 5
decfcvt(1)	=	"123457"	*decpt = 5
decfcvt(3)	=	"12345670"	*decpt = 5

Now let *np* point to .001234:

dececv(4)	=	"1234"	*decpt = -2
dececv(10)	=	"1234000000"	*decpt = -2
decfcvt(1)	=	" "	*decpt = -2
decfcvt(3)	=	"1"	*decpt = -2

DECECVT Example

```
/*
 * dececv.t.ec *

The following program converts a series of DECIMAL numbers to fixed
strings of 20 ASCII digits. For each conversion it displays the resulting
string, the decimal position from the beginning of the string and the
sign value. The display illustrates how the results of dececv.t should
be interpreted.
*/

#include <stdio.h>

#include decimal;

char *strings[] =
{
    "210203.204",
    "4894",
    "443.334899312",
    "-12344455",
    0
};

char result[40];

main()
{
    int x;
    int i = 0, f, sign;
    dec_t num;
    char *dp, *dececv.t();

    while(strings[i])
    {
        if (x = decconvasc(strings[i], strlen(strings[i]), &num))
        {
            printf("Error %d in converting string [%s] to Decimal\n",
                x, strings[i]);
            break;
        }
        dp = dececv.t(&num, 20, &f, &sign); /* to ASCII string */

        /* display result */
        printf("\tInput string[%d]: %s\n", i, strings[i]);
        printf("\tOutput of dececv.t: %c%*.s.%s decpt: %d sign: %d\n\n",
            (sign ? '-' : '+'), f, f, dp, dp+f, f, sign);
        ++i;
        /* next string */
    }
}
```

DECECVT Example Output

```
Input string[0]: 210203.204
Output of dececv: +210203.2040000000000000 decpt: 6 sign: 0

Input string[1]: 4894
Output of dececv: +4894.0000000000000000 decpt: 4 sign: 0

Input string[2]: 443.334899312
Output of dececv: +443.334899312000000000 decpt: 3 sign: 0

Input string[3]: -12344455
Output of dececv: -12344455.000000000000 decpt: 8 sign: 1
```

DECFCVT Example

```

/*
 * decfcvt.ec *

The following program converts a series of DECIMAL numbers to strings of
ASCII digits with 3 digits to the right of the decimal point. For each
conversion it displays the resulting string, the position of the decimal
point from the beginning of the string and the sign value. The display
illustrates how the results of decfcvt should be interpreted.
*/

#include <stdio.h>

#include decimal;

char *strings[] =
{
    "210203.204",
    "4894",
    "443.334899312",
    "-12344455",
    0
};

char result[40];

main()
{
    int x;
    dec_t num;
    int i = 0, f, sign;
    char *dp, *decfcvt();

    while(strings[i])
    {
        if (x = deccvasc(strings[i], strlen(strings[i]), &num))
        {
            printf("Error %d in converting string [%s] to Decimal\n",
                x, strings[i]);
            break;
        }

        dp = decfcvt(&num, 3, &f, &sign); /* to ASCII string */

        /* display result */
        printf("\tInput string[%d]: %s\n", i, strings[i]);
        printf("\tOutput of decfcvt: %c%.3s decpt: %d sign: %d\n\n",
            (sign ? '-' : '+'), f, f, dp, dp+f, f, sign);
        ++i; /* next string */
    }
}

```

DECFCVT Example Output

Input string[0]: 210203.204
Output of decfcvt: +210203.204 decpt: 6 sign: 0

Input string[1]: 4894
Output of decfcvt: +4894.000 decpt: 4 sign: 0

Input string[2]: 443.334899312
Output of decfcvt: +443.335 decpt: 3 sign: 0

Input string[3]: -12344455
Output of decfcvt: -12344455.000 decpt: 8 sign: 1

DECROUND

Purpose

The **decround** function rounds a **decimal** type number to fractional digits.

Syntax

```
void decround(d, s)
    dec_t *d;
    int s;
```

d is a **dec_t** structure for a **decimal** number whose value is rounded.

s is the number of fractional digits to which the number in *d* is rounded.

Usage

The rounding factor is $5 \times 10^{-s-1}$. Rounding is performed by adding the factor to a positive number or by subtracting it from a negative number, and then truncating to *s* digits, as follows:

unrounded	<i>s</i>	rounded	truncated
1.4	0	1.0	1.0
1.5	0	2.0	1.0
1.684	2	1.68	1.68
1.685	2	1.69	1.68
1.685	1	1.7	1.6
1.685	0	2.0	1.0

Example

```
/*
 * decround.ec *

The following program rounds a decimal type number six times and displays
the result of each operation.
*/

#include <stdio.h>

#include decimal;

char string[] = "-12345.038572";
char result[40];

main()
{
    int x;
    int i = 6; /* number of decimal places to start with */
    dec_t num1;

    while(i)
    {
        if (x = deccvasc(string, strlen(string), &num1))
        {
            printf("Error %d in converting string to decimal\n", x);
            break;
        }
        decround(&num1, i);
        if (x = dectoasc(&num1, result, sizeof(result), -1))
        {
            printf("Error %d in converting result to string\n", x);
            break;
        }
        printf("\tRound %d: %s\n", i--, result);
    }
}
```

Example Output

```
Round 6: -12345.038572
Round 5: -12345.03857
Round 4: -12345.0386
Round 3: -12345.039
Round 2: -12345.04
Round 1: -12345.0
```

DECTRUNC

Purpose

The **detrunc** function truncates to fractional digits a **decimal** type number that has been rounded.

Syntax

```
void detrunc(d, s)
    dec_t *d;
    int s;
```

d is a **dec_t** structure for a rounded **decimal** number whose value is truncated.

s is the number of fractional digits to which the number is truncated.

Usage

The following examples show the outcome of using **detrunc** with various inputs:

unrounded	<i>s</i>	rounded	truncated
1.4	0	1.0	1.0
1.5	0	2.0	1.0
1.684	2	1.68	1.68
1.685	2	1.69	1.68
1.685	1	1.7	1.6
1.685	0	2.0	1.0

Example

```
/*
 * dectrunc.ec *

The following program truncates a decimal number six times and displays each
result.
*/

#include <stdio.h>

#include decimal;

char string[] = "    -12345.038572";
char result[40];

main()
{
    int x;
    int i = 6; /* number of decimal places to start with */
    dec_t num1;

    while(i)
    {
        if (x = deccvasc(string, strlen(string), &num1))
        {
            printf("Error %d in converting string to decimal\n", x);
            break;
        }
        dectrunc(&num1, i);
        if (x = dectoasc(&num1, result, sizeof(result), -1))
        {
            printf("Error %d in converting result to string\n", x);
            break;
        }
        printf("\tTruncate to %d: %s\n", i--, result);
    }
}
```

Example Output

```
Truncate to 6: -12345.038572
Truncate to 5: -12345.03857
Truncate to 4: -12345.0385
Truncate to 3: -12345.038
Truncate to 2: -12345.03
Truncate to 1: -12345.0
```

RFMTDEC

Purpose

The `rfmtdec` function converts a `dec_t` in internal format to a character string formatted according to a pattern.

Syntax

```
int rfmtdec(dec, format, outbuf)
    dec_t *dec;
    char *format;
    char *outbuf;
```

<i>dec</i>	is the address of the decimal number to be formatted.
<i>format</i>	is the address of the format string. The formatting possibilities are described in the section “Numeric-Formatting Routines” on page 2-31.
<i>outbuf</i>	is the address of the buffer to receive the formatted string.

Return Codes

0	The conversion was successful.
-1211	The program ran out of memory—memory-allocation error.
-1217	The format string is too large.

Example

```
/*
 * rfmtdec.ec *

The following program applies a series of format specifications to each of
a series of decimal numbers and displays each result.
*/

#include <stdio.h>

#include decimal;

char *strings[] =
{
    "210203.204",
    "4894",
    "443.334899312",
    "-12344455",
    0
};

char *formats[] =
{
    "*****",
    "$$$$$$.##",
    "(&&, &&&, &&&.)",
    "<, <<, <<<, <<<<",
    "$*****.**",
    0
};

char result[40];

main()
{
    int x;
    int s = 0, f;
    dec_t num;

    while(strings[s])
    {
```



```
/*
 * Convert each string to decimal
 */
if (x = deccvasc(strings[s], strlen(strings[s]), &num))
{
    printf("Error %d in converting string [%s] to decimal\n",
        x, strings[s]);
    break;
}
f = 0;
while(formats[f])
{
    /*
     * Format DecimalDecimal num for each of formats[f]
     */
    rfmtdec(&num, formats[f], result);
    /*
     * Display result and bump to next format (f++)
     */
    printf("\n\tstrings[%d], formats[%d]: %s", s, f++, result);
}
++s; /* bump to next string */
printf("\n"); /* separate result groups */
}
```

Example Output

```
strings[0], formats[0]: **      210203
strings[0], formats[1]:  $210203.20
strings[0], formats[2]:  000210,203.
strings[0], formats[3]:  210,203
strings[0], formats[4]:  $***210203.20

strings[1], formats[0]: **      4894
strings[1], formats[1]:  $4894.00
strings[1], formats[2]:  000004,894.
strings[1], formats[3]:  4,894
strings[1], formats[4]:  $*****4894.00

strings[2], formats[0]: **      443
strings[2], formats[1]:  $443.33
strings[2], formats[2]:  0000000443.
strings[2], formats[3]:  443
strings[2], formats[4]:  $*****443.33

strings[3], formats[0]: **     12344455
strings[3], formats[1]:  $12344455.00
strings[3], formats[2]:  (12,344,455.)
strings[3], formats[3]:  12,344,455
strings[3], formats[4]:  $*12344455.00
```

Working with Time Data Types

Chapter Overview	3
The DATE Data Type	3
DATE Functions	4
RDATESTR	5
RDAYOFWEEK	7
RDEFMTDATE	9
RFMTDATE	12
RJULMDY	15
RLEAPYEAR	17
RMDYJUL	19
RSTRDATE	21
RTODAY	23
DATETIME and INTERVAL Data Types	24
DATETIME and INTERVAL Columns	24
Declaring DATETIME and INTERVAL Host Variables	25
Fetching DATETIME and INTERVAL Values	26
Fetching DATETIME Values	26
Fetching INTERVAL Values	26
Implicit Data Conversion When Fetching	26
Storing DATETIME and INTERVAL Values	27
Implicit Data Conversion When Storing DATETIME and INTERVAL Values	27
Converting Between DATETIME and DATE Data Types	27
DATETIME and INTERVAL Data Type Functions	28
DTCURRENT	30

DTCVASC	32
DTCVFMTASC	35
DTEXTEND	37
DTTOASC	40
DTTOFMTASC	42
INCVASC	44
INCVFMTASC	46
INTOASC	48
INTOFMTASC	50

Chapter Overview

This chapter contains information on how to use DATETIME, INTERVAL, and DATE data type values in an **INFORMIX-ESQL/C** program. Specifically, it contains the following information:

- An overview of the DATE data type
- A list of functions you can use with the DATE data type
- The syntax for functions that you can use to manipulate the DATE data type
- An overview of the DATETIME and INTERVAL data types and how to use them
- A discussion of converting DATETIME and DATE data types
- The syntax for functions that you can use to manipulate the DATETIME and INTERVAL data types

For information about all of the data types available for use in an **ESQL/C** program, see Chapter 2 of this manual. For information about SQL data types, see Chapter 3 of *The Informix Guide to SQL: Reference*.

The DATE Data Type

INFORMIX-ESQL/C stores dates as four-byte integers whose value is the number of days since December 31, 1899. Dates before December 31, 1899, are negative numbers, while dates after December 31, 1899, are positive numbers. For a complete description of the DATE data type, see Chapter 3 of *The Informix Guide to SQL: Reference*.

DATE Functions

The following DATE manipulation functions are included in the libraries distributed with **INFORMIX-ESQL/C** for converting dates written in string form to and from this internal format. These functions are described on the next several pages.

Function Name	Description
rdatestr	Converts an internal format to a string format
rdayofweek	Returns day of the week
rdefmtdate	Converts a string format to an internal format
rfmtdate	Converts an internal format to a string format
rjulmdy	Returns month, day, and year from an internal format
rleapyear	Determines whether it is leap year
rmdyjul	Returns an internal format from month, day, and year
rstrdate	Converts a string format to an internal format
rtoday	Returns a system date in internal format

RDATESTR

Purpose

The **rdatestr** function converts a date in internal format to a character string date of the form *mm/dd/yyyy*.

Syntax

```
int rdatestr(jdate, str)  
    long jdate;  
    char *str;
```

<i>jdate</i>	is the internal representation of a date as a long integer.
<i>str</i>	is a pointer to the area where the results are to be stored.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * rtoday.ec *

The following program obtains today's date from the system.
It then converts it to ASCII for displaying the result.
*/

#include <stdio.h>

main()
{
    int errnum;
    char today_date[20];
    long i_date;

    /* Get today's date in the internal format */
    rtoday(&i_date);

    /* Convert date from internal format into a mm/dd/yyyy string */
    if ((errnum = rdatestr(i_date, today_date)) == 0)
        printf("\n\tToday's date is %s.\n", today_date);
    else
        printf("\n\tError %d in converting date to mm/dd/yyyy\n", errnum);
}
```

Example Output

```
Today's date is 10/26/1991.
```

RDAYOFWEEK

Purpose

The **rdayofweek** function returns the day of the week represented as an integer, given an internal date as an argument.

Syntax

```
int rdayofweek(jdate)  
    long jdate;
```

jdate is the internal representation of the date as a long integer.

Return Values

0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Example

```
/*
 * rdayofweek.ec *

The following program obtains today's date from the system
and determines the what day of the week it is.
*/

#include <stdio.h>

main()
{
    char today_date[11];
    long i_date;
    char *day_name;

    /* Put today's date into the internal format for a date */
    rtoday(&i_date);

    /* Put i_date into a mm/dd/yyyy string */
    rdatestr(i_date, today_date);

    printf("\nToday's date is %s.\n", today_date);

    /* Figure out what day of the week i_date is */
    switch (rdayofweek(i_date))
    {
        case 0: day_name = "Sunday"; break;
        case 1: day_name = "Monday"; break;
        case 2: day_name = "Tuesday"; break;
        case 3: day_name = "Wednesday"; break;
        case 4: day_name = "Thursday"; break;
        case 5: day_name = "Friday"; break;
        case 6: day_name = "Saturday"; break;
    }
    printf("\nIt is %s.\n", day_name);
}
```

Example Output

```
Today's date is 10/26/1991.

It is Saturday.
```

RDEFMTDATE

Purpose

The **rdefmtdate** function creates a long integer whose value is the number of days since December 31, 1899, for a string date whose format is provided.

Syntax

```
int rdefmtdate(jdate, fmtstring, input)
    long *jdate;
    char *fmtstring;
    char *input;
```

<i>fmtstring</i>	is a pointer to a character array containing the format pattern for the date supplied in <i>input</i> .
<i>input</i>	is a pointer to the string containing the date to be converted to a long integer.
<i>jdate</i>	is a long integer, the internal representation of the date expressed as <i>input</i> .

Usage

The string *fmtstring* uses the same formatting characters as **rfmtdate**, as shown on page 5-12.

The *input* string and the *fmtstring* must be in the same sequential order in terms of month, day, and year. They need not, however, have the same literals nor the same representation for month, day, and year.

The following combinations of *fmtstring* and *input* are valid:

fmtstring	input
"mmdyy"	"Dec. 25th, 1991"
"mmm. dd. yyyy"	"dec 25 1991"
"mmm. dd. yyyy"	"DEC-25-1991"
"mmm. dd. yyyy"	"122591"
"mmm. dd. yyyy"	"12/25/91"
"yy/mm/dd"	"91/12/25"
"yy/mm/dd"	"1991, December 25th"
"yy/mm/dd"	"In the year 1991, the month of December, its 25th day"
"dd-mm-yy"	"This 25th day of December, 1991"

Return Codes

0	The operation was successful.
-1204	There is an invalid year component in the <i>input</i> parameter.
-1205	There is an invalid month component in the <i>input</i> parameter.
-1206	There is an invalid day component in the <i>input</i> parameter.
-1209	Since <i>*input</i> does not contain delimiters between the year, month, and day components, the length of <i>*input</i> must be exactly six or eight bytes.
-1212	<i>fmtstring</i> does not contain a year, a month, and a day component.

Example

```
/*
 * rdefmtdate.ec *

The following program accepts a date entered from the console,
converts it into the internal date format using rdefmtdate().
It checks the conversion by finding the day of the week.
*/

#include <stdio.h>

main()
{
    int x;
    char date[20];
    long i_date;
    char *day_name;

    static char fmtstr[9] = "mddyyyy";

    printf("Enter a date as a single string, month.day.year\n");
    gets(date);

    printf("\nThe date is %s.\n", date);

    if (x = rdefmtdate(&i_date, fmtstr, date))
        printf("Error %d on rdefmtdate conversion\n", x);
    else
    {
        /* Figure out what day of the week i_date is */
        switch (rdayofweek(i_date))
        {
            case 0: day_name = "Sunday"; break;
            case 1: day_name = "Monday"; break;
            case 2: day_name = "Tuesday"; break;
            case 3: day_name = "Wednesday"; break;
            case 4: day_name = "Thursday"; break;
            case 5: day_name = "Friday"; break;
            case 6: day_name = "Saturday"; break;
        }
        printf("\nThe day of the week is %s.\n", day_name);
    }
}
```

Example Output

```
Enter a date as a single string, month.day.year
020192

The date is 020192.

The day of the week is Saturday.
```

RFMTDATE

Purpose

The **rfmtdate** function converts a date in internal format to a string formatted according to a pattern.

Syntax

```
int rfmtdate(jdate, fmtstring, result)
    long jdate;
    char *fmtstring;
    char *result;
```

fmtstring is a pointer to the character array containing the format pattern for the date returned in *result*.

jdate is the internal representation of a date as a long integer.

result is a pointer to the character array that receives the formatted date.

Usage

The *fmtstring* date string consists of combinations of the characters *m*, *d*, and *y*, as shown in the following example:

Format	Meaning
dd	Day of the month as a 2-digit number (01-31)
ddd	Day of the week as a 3-letter abbreviation (Sun through Sat)
mm	Month as a 2-digit number (01-12)
mmm	Month as a 3-letter abbreviation (Jan through Dec)
yy	Year as a 2-digit number in the 1900s (00-99)
yyyy	Year as a 4-digit number (0001-9999)

Any other characters in *fmtstring* are reproduced literally in *result*.

The examples that follow convert the integer *jdate* that corresponds to December 25, 1992, to a string *result* using the format in *fmtstring*:

fmtstring	result
"mmdyy"	122592
"ddmmyy"	251292
"yymmdd"	921225
"yy/mm/dd"	92/12/25
"yy mm dd"	92 12 25
"yy-mm-dd"	92-12-25
"mmm. dd, yyyy"	Dec. 25, 1992
"mmm dd yyyy"	Dec 25 1992
"yyyy dd mm"	1992 25 12
"mmm dd yyyy"	Dec 25 1992
"ddd, mmm. dd, yyyy"	Mon, Dec. 25, 1992
"(ddd) mmm. dd, yyyy"	(Mon) Dec. 25, 1992

Return Codes

0	The conversion was successful.
-1210	The internal date cannot be converted to month-day-year format.
-1211	The program ran out of memory—memory-allocation error.

Example

```
/*
 * rfmdtdate.ec *

The following program converts a date from internal format to
a specified format using rfmdtdate().
*/

#include <stdio.h>

main()
{
    char today_date[11];
    long i_date;
    int x;

    /* Put today's date into the internal format for a date */
    rtoday(&i_date);

    /*
     * Convert date to "mm-dd-yyyy" format
     */
    if (x = rfmdtdate(i_date, "mm-dd-yyyy", today_date))
        printf("rfmdtdate call failed with error %d\n", x);
    else
        printf("\tToday's date is %s.\n", today_date);

    /*
     * Convert date to "mm.dd.yy" format
     */
    if (x = rfmdtdate(i_date, "mm.dd.yy", today_date))
        printf("rfmdtdate call failed with error %d\n",x);
    else
        printf("\tToday's date is %s.\n", today_date);

    /*
     * Convert date to "mmm ddth, yyyy" format
     */
    if (x = rfmdtdate(i_date, "mmm ddth, yyyy", today_date))
        printf("rfmdtdate call failed with error %d\n", x);
    else
        printf("\tToday's date is %s.\n", today_date);
}
```

Example Output

```
Today's date is 10-26-1991.
Today's date is 10.26.91.
Today's date is Oct 26th, 1991.
```

RJULMDY

Purpose

The **rjulmdy** function creates an array of three short integers containing the month, day, and year components corresponding to an internal date.

Syntax

```
int rjulmdy(jdate, mdy)  
    long jdate;  
    short mdy[3];
```

<i>jdate</i>	is the internal representation of the date as a long integer.
<i>mdy</i>	is an array of short integers, where <i>mdy</i> [0] is the month (1 to 12), <i>mdy</i> [1] is the day (1 to 31), and <i>mdy</i> [2] is the year (1 to 9999).

Return Codes

= 0	The operation was successful.
< 0	The operation failed.

Example

```
/*
 * rjulmdy.ec *

The following program illustrates the conversion of date in internal format
to an array of three short integers, one each for month, day and year.
*/

#include <stdio.h>

main()
{
    long i_date;
    short mdy_array[3];
    int errnum;

    /* Get today's date in the internal format */
    rtoday(&i_date);

    /* Convert from internal format to MDY array */
    if ((errnum = rjulmdy(i_date, mdy_array)) == 0)
    {
        printf("The month component is: %d\n", mdy_array[0]);
        printf("The day component is: %d\n", mdy_array[1]);
        printf("The year component is: %d\n", mdy_array[2]);
    }
    else
        printf("rjulmdy call failed with error %d", errnum);
}
```

Example Output

```
The month component is: 10
The day component is: 26
The year component is: 1991
```

RLEAPYEAR

Purpose

The **rleapyear** function returns TRUE when the argument passed to it is a leap year and FALSE when it is not.

Syntax

```
int rleapyear(year)  
    int year;
```

year is an integer.

Usage

The argument *year* must be the year component of a date and not the date itself.

The year must be expressed in full (1992) and not abbreviated (92).

Return Codes

1	The year is a leap year.
0	The year is not a leap year.

Example

```
/*
 * rleapyear.ec *

The following program obtains the system date into a long integer in
the internal format.
It then converts the internal format into an array of three short integers
that contain the month, day, and year portions of the date.
It then tests the year value to see if the year is a leap year.
*/

#include <stdio.h>

main()
{
    long i_date;
    int errnum;
    short mdy_array[3];

    /* Get today's date in the internal format */
    rtoday(&i_date);

    /* Convert internal format into a MDY array */
    if ((errnum = rjulmdy(i_date, mdy_array)) == 0)
    {
        /* Check if it is a leap year */
        if (rleapyear(mdy_array[2]))
            printf("%d is a leap year\n", mdy_array[2]);
        else
            printf("%d is not a leap year\n", mdy_array[2]);
    }
    else
        printf("rjulmdy call failed with error %d", errnum);
}
```

Example Output

```
1991 is not a leap year
```

RMDYJUL

Purpose

The **rmidyjul** function creates an internal date from three short integers that contain the numeric values for the month, day, and year.

Syntax

```
int rmidyjul(mdy, jdate)
    short mdy[3];
    long *jdate;
```

<i>jdate</i>	is a pointer to the internal representation of the returned date as a long integer.
<i>mdy</i>	is an array of short integers, where <i>mdy</i> [0] is the month (1 to 12), <i>mdy</i> [1] is the day (1 to 31), and <i>mdy</i> [2] is the year (1 to 9999).

Usage

The year must be expressed in full (1992) and not abbreviated (92).

Return Codes

0	The operation was successful.
-1204	There was an invalid year component in <i>mdy</i> [2].
-1205	There was an invalid month component in <i>mdy</i> [0].
-1206	There was an invalid day component in <i>mdy</i> [1].

Example

```
/*
 * rmdyjul.ec *

The following program converts an array of three short integers
(containg values for month, day and year) into internal format for date.
*/

#include <stdio.h>

main()
{
    long i_date;
    int errnum;
    static short mdy_array[3] = { 12, 21, 1985 };

    /* Convert MDY array into internal format */
    if ((errnum = rmdyjul(mdy_array, &i_date)) == 0)
        printf("12/21/1985 converted to internal format\n");
    else
        printf("rmdyjul call failed with errnum = %d\n", errnum);
}
```

Example Output

```
12/21/1985 converted to internal format
```

RSTRDATE

Purpose

The **rstrdate** function converts a character string date to a date in internal format.

Syntax

```
int rstrdate(str, jdate)
    char *str;
    long *jdate;
```

jdate is a pointer to a long integer that receives the converted date.

str is a pointer to the string to be converted.

Usage

The *str* should contain a numeric month, day, and year in that order. Any non-numeric character can be used as a separator between the month, day, and year.

The year must be expressed in full (1992) and not abbreviated (92).

Return Codes

= 0 The conversion was successful.

< 0 The conversion failed.

Example

```
/*
 * rstrdate.ec *
 * The following program converts a character string
 * in "mmddyyyy" format to an internal date format.
 */

#include <stdio.h>

main()
{
    long i_date;
    int errnum;

    /* Convert Sept. 6th, 1989 into i_date */
    if ((errnum = rstrdate("9.6.1989", &i_date)) == 0)
        printf("9/6/1989 converted to internal format\n");
    else
        printf("rstrdate call failed with error %d\n", errnum);
}
```

Example Output

```
9/6/1989 converted to internal format
```

RTODAY

Purpose

The `rtoday` function puts the system date into internal format.

Syntax

```
void rtoday(today)
    long *today;
```

today is a pointer to a long integer that receives the current system date in internal format.

Example

```
/*
 * rtoday.ec *

The following program obtains today's date from the system.
It then converts it to ASCII for displaying the result.
*/

#include <stdio.h>

main()
{
    int errnum;
    char today_date[20];
    long i_date;

    /* Get today's date in the internal format */
    rtoday(&i_date);

    /* Convert date from internal format into a mm/dd/yyyy string */
    if ((errnum = rdatestr(i_date, today_date)) == 0)
        printf("\n\tToday's date is %s.\n", today_date);
    else
        printf("\n\tError %d in converting date to mm/dd/yyyy\n", errnum);
}
```

Example Output

```
Today's date is 10/26/1991.
```

DATETIME and INTERVAL Data Types

The DATETIME data type encodes a time to a particular precision. The precision is expressed by a qualifier, and the qualifier is an integral part of the data type. As a host variable, a DATETIME value is represented in a structure of type **dttime_t**:

```
typedef struct dttime {
    short dt_qual;
    dec_t dt_dec;
} dttime_t;
```

The qualifier of the value is represented in the **dt_qual** field, and the digits of the fields of the value are stored in **dt_dec**.

The INTERVAL data type encodes an interval of time to a particular precision. The precision is expressed by a qualifier and, as with DATETIME, the qualifier is an integral part of the data type. As a host variable, an INTERVAL value is represented in a structure of type **intrvl_t**:

```
typedef struct intrvl {
    short in_qual;
    dec_t in_dec;
} intrvl_t;
```

These structures, along with a number of macro definitions for use in composing qualifier values, are contained in the *include* file **datetime.h**. The type **dec_t** that is a component of these structures is defined in the file **decimal.h**.

A DATETIME or INTERVAL data type is stored as a decimal number with a scale factor of zero and a precision equal to the number of digits implied by its qualifier. Once you know the precision and scale, you know the storage format. For example, a table column defined as DATETIME YEAR TO DAY contains four digits for year, two digits for month, and two digits for day, for a total of eight digits. It is thus stored as if it were DECIMAL(8,0).

DATETIME and INTERVAL Columns

A DATETIME column holds a value that represents a moment in time. For information about creating a column of the DATETIME data type, see Chapter 3 of *The Informix Guide to SQL: Reference*.

Use the INTERVAL data type for columns in which you plan to store values that represent a span or period of time. For information about using the INTERVAL data type in a column, see Chapter 3 of *The Informix Guide to SQL: Reference*.

Declaring DATETIME and INTERVAL Host Variables

Declare a host variable for a DATETIME column by using the data type DATETIME followed by an optional DATETIME qualifier:

```
$datetime year to day holidays[10];
$datetime hour to second wins, places, shows;
$datetime column6;
```

If you omit the qualifier from the declaration of the **datetime** host variable, as in the last example, your program must explicitly initialize the qualifier using the macros shown on page 5-29.

Declare a host variable for an INTERVAL column by using the data type INTERVAL followed by an optional INTERVAL qualifier.

```
$interval day(3) to day accrued_leave, leave_taken;
$interval hour to second race_length;
$interval scheduled;
```

If you omit the qualifier from the declaration of the **interval** host variable, as in the last example, your program must explicitly initialize the qualifier using the macros shown on page 5-29.

Because of the multiword nature of these data types, it is not possible to declare an uninitialized **datetime** or **interval** host variable named **year**, **month**, **day**, **hour**, **minute**, **second**, or **fraction**. You should avoid the following declarations:

```
$datetime year; /* will cause an error */
$datetime year to day year, today; /* ambiguous */
```

Fetching DATETIME and INTERVAL Values

Fetching DATETIME Values

When a DATETIME value is fetched into a host variable of type **dttime_t**, one of two events occurs:

- When the **dt_qual** field contains a valid qualifier, the database value is extended to match the qualifier. (*Extending* is the operation of adding or dropping fields of a DATETIME value to make it match a given qualifier. Extending is done in SQL statements with the EXTEND function and in INFORMIX-ESQL/C with the **dtextend** function.)
- When the **dt_qual** field does not contain a valid qualifier, the database value and its qualifier are both fetched, thus initializing the host variable. Zero is an invalid qualifier, so zero can be stored into the **dt_qual** field whenever the database value is to be fetched, without extending it.

Fetching INTERVAL Values

When an INTERVAL value is fetched into an **intrvl_t** host variable, the **in_qual** field of the variable is tested. If it contains zero (or any invalid qualifier) both the database value and the database qualifier are fetched, initializing the variable.

When the variable qualifier is valid, it is checked for compatibility with the database qualifier. (INTERVAL qualifiers are compatible provided that, if one contains a field of **month** or **year**, the other contains only fields of **month** and **year**.)

Implicit Data Conversion When Fetching

You can automatically convert DATETIME and INTERVAL values between database columns and host variables of character type (**char**, **string**, or **fixchar**). The fields of the DATETIME or INTERVAL value in the database are converted to a character string, which is stored in the host variable. If the host variable is too short, the string is truncated, **sqlca.sqlwarn.sqlwarn1** is set to **W**, and the indicator variable (if any) is set to the needed length.

Note that DATETIME and INTERVAL values cannot be fetched automatically into number host variables.

Storing DATETIME and INTERVAL Values

When a host variable is used to store a DATETIME or INTERVAL value in the database, it must contain a valid qualifier.

When storing a DATETIME value, the qualifier in the host variable can be different from the qualifier of the database column. The host variable value is extended to match the database qualifier. When storing an INTERVAL value, the host variable qualifier must be compatible with that of the database column.

If the host qualifier is invalid or INTERVAL qualifiers are incompatible, a negative error code is set in `sqlca.sqlcode` and the update or insert operation fails.

Implicit Data Conversion When Storing DATETIME and INTERVAL Values

When a host variable of **character** type is used to update or insert a DATETIME or INTERVAL value, the database server tries to convert the characters using the qualifier and type of the database column. If the conversion fails, `sqlca.sqlcode` is set to a negative value and the update or insert operation fails.

Automatic conversion from **number** and **date** host variables is not supported.

Converting Between DATETIME and DATE Data Types

No functions are provided to convert automatically between the DATETIME and DATE data types. You can perform these conversions using existing functions and intermediate strings.

To convert a DATETIME value to a DATE value, follow these steps:

1. Use `dtextend` to adjust the DATETIME qualifier to *year to day*.
2. Apply `dttoasc`, creating a character string in the form *yyyy-mm-dd*.
3. Use `rdefmtdate` with a pattern argument of *yyyy-mm-dd* to convert the string to a DATE.

To convert a DATE value to a DATETIME value, follow these steps:

1. Declare a host variable with a qualifier of *year to day* (or, initialize the qualifier with the value returned by `TU_DTENCODE(TU_YEAR,TU_DAY)`).
2. Use `rfmtdate` with a pattern of *yyyy-mm-dd* to convert the DATE value to a character string.

3. Use **dtcvasc** to convert the character string to a value in the prepared **datetime** variable.
4. If necessary, use **dtextend** to adjust the DATETIME qualifier.

DATETIME and INTERVAL Data Type Functions

The following C function calls are available to treat **datetime** and **interval** host variables:

Function Name	Description
dtcurrent	Gets the current date and time
dtextend	Changes the qualifier of datetime
dtcvasc	Converts an ANSI-compliant character string to datetime
dtcvfmtasc	Converts a character string to datetime
dttoasc	Converts a datetime to an ANSI-compliant character string
dttofmtasc	Converts a datetime to a character string
incvasc	Converts an ANSI-compliant character string to interval
incvfmtasc	Converts a character string to interval
intoasc	Converts an interval to an ANSI-compliant character string
intofmtasc	Converts an interval to a character string

These functions are shown alphabetically on the following pages.

In addition to these functions, an *include* file named **datetime.h** is included with the **INFORMIX-ESQL/C** libraries. This file, which defines the data structures, also defines the following names and macro functions (which are required only when working directly with qualifiers in binary form):

Name of macro	Description
TU_YEAR	The name for a qualifier field
TU_MONTH	The name for a qualifier field
TU_DAY	The name for a qualifier field
TU_HOUR	The name for a qualifier field
TU_MINUTE	The name for a qualifier field
TU_SECOND	The name for a qualifier field
TU_FRAC	The name for the leading qualifier field of FRACTION
TU_Fn	The names for datetime ending fields of "FRACTION(n)", for n from 1 to 5
TU_START(q)	Returns the leading field number from qualifier q
TU_END(q)	Returns the trailing field number from qualifier q
TU_DTENCODE(f,t)	Composes a DATETIME qualifier from the first field number f and trailing field number t
TU_IENCODE(p,f,t)	Composes an INTERVAL qualifier from the first field number f with precision p and trailing field number t

Figure 5-1 *DATETIME and INTERVAL qualifier macros*

DTCURRENT

Purpose

The **dtcurrent** function assigns the current date and time to a **datetime** variable.

Syntax

```
void dtcurrent(d)
    dtime_t *d;
```

d is the address of an initialized **dtime_t** host variable.

Usage

If the variable qualifier is set to zero (or any invalid qualifier), it is initialized to *year to fraction(3)*.

If the variable contains a valid qualifier, the current date and time are extended to agree with the qualifier.

Calls

These statements set the variable *timewarp* to the current date:

```
$datetime year to day timewarp;
dtcurrent (&timewarp);
```

These statements set the variable *now* to the current time, to the nearest millisecond:

```
now.dt_qual = TU_DTENCODE (TU_HOUR, TU_F3);
dtcurrent (&now);
```

Example

```
/*
 * dtcurrent.ec *

The following program obtains the current date from the system
in an internal format. It then converts it to ASCII and prints it.
*/

#include <stdio.h>

#include datetime;

main()
{
    int x;
    char out_str[16];
    $datetime year to hour dt1;

    /* Get today's date */
    dtcurrent(&dt1);

    /* Convert to ASCII for displaying */
    dttoasc(&dt1, out_str);
    printf("\tThe value of dt1 (year to hour) is %s\n", out_str);
}
```

Example Output

```
The value of dt1 (year to hour) is 1991-10-26 14
```

DTCVASC

Purpose

The **dtcvasc** function converts a string that conforms to ANSI SQL standards to a **datetime** value.

Syntax

```
int dtcvasc(str,d)
    char *str;
    dttime_t *d;
```

d is the address of an initialized **dttime_t** variable.

str is the address of a string of digits and field delimiters.

Usage

The variable must be initialized with the desired qualifier.

The input string can have leading and trailing spaces. However, from the first significant digit to the last, the only characters accepted are digits and delimiters appropriate to the fields implied by the qualifier.

If a year value is given as one or two digits, *1900* is added to it.

If the input string is acceptable, the value is set in the variable and the function returns zero. Otherwise, it does not change the variable and returns a negative error code.

Return Codes

-1260	It is not possible to convert between the specified types.
-1261	There are too many digits in the first datetime or interval field.
-1262	There is a non-numeric character in datetime or interval .
-1263	A field in a datetime or interval is out of range.
-1264	There are extra characters at the end of a datetime or interval .
-1265	Overflow occurred on a datetime or interval operation.

- 1266 A **datetime** or **interval** is incompatible for the operation.
- 1267 The result of a **datetime** computation is out of range.
- 1268 There is an invalid **datetime** qualifier.

Example Call

Here, the variable *columbus* is initialized to Columbus's birthday, 1989:

```
$datetime year to day columbus;  
dctvasc("89-10-9",&columbus);
```

Example

```
/*  
 * dctvasc.ec *  
  
The following program converts ASCII datetime strings in ANSI SQL format  
into datetime (dttime_t) structure.  
*/  
  
#include <stdio.h>  
  
$include datetime;  

```

Example Output

```
Result #1: successful conversion  
Error -1263 in converting 2nd datetime string
```

DTCVFMTASC

Purpose

The **dtcvfmtasc** function converts a string with a specified format to a **datetime** value.

Syntax

```
int dtcvfmtasc(str, fmtstr, d)
    char *str;
    char *fmtstr;
    dtime_t *d;
```

<i>d</i>	is the address of a dtime_t variable with an initialized qualifier.
<i>fmtstr</i>	is the address of the format string, using the directives defined for the DBTIME environment variable. If this argument is null, the function uses the format specified by DBTIME.
<i>str</i>	is the address of a string of digits and field delimiters.

Usage

The output variable must be initialized with the desired qualifier.

The input string can have leading and trailing spaces. However, from the first significant digit to the last, the only characters accepted are digits and delimiters appropriate to the fields implied by the format string.

If the input string and the format specification are acceptable, the value is set in the variable and the routine returns zero. Otherwise, an error code is returned and the output variable contains an unpredictable value.

The output qualifier does not need to be identical to the input qualifier as specified by the format string. When the output qualifier is different from the input, **dtcvfmtasc** performs extensions as if **dtextend** had been called.

If *fmtstr* is null and DBTIME is not defined, the standard ANSI SQL format is used. The input must have values for *year to second* in the ANSI SQL format.

All fields in the **datetime/interval** input string must be contiguous when calling **dtcvfmtasc()**. In other words, if the qualifier is *hour-to-second*, all values for *hour*, *minute*, and *second* must be specified somewhere in the string, or an error results.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

Here the variable *birthday* is initialized to a fictional birthday party:

```
/*
 *The input and output qualifiers are the same (month
 *to minute) The variable 'birthday' will be set to
 * "06-09 13:30"

 *Note the absence of field-width and precision specification
 * in the input format string.
 */
$datetime month to minute birthday;
dtcvfmtasc("June 9 at 01:30pm",
           "%B %d at %I:%M%p",
           &birthday );

/*
 * The input and output qualifiers are different:
 * input qual: month to minute
 * output qual: year to minute
 *The variable 'birthday' will be set to "XXXX-06-09 13:30"
 *Notice that the output has been extended, and the year
 *is set to the current year.
 */
$datetime year to minute birthday;
dtcvfmtasc("June 9 at 01:30pm", "%B %d at %I:%M%p",
           &birthday );
```

DTEXTEND

Purpose

The **dtextend** function copies a DATETIME value under a different qualifier.

Syntax

```
int dtextend(id,od)
           dtime_t *id, *od;
```

id is the address of a variable to be copied.

od is the address of a variable with a valid qualifier.

Usage

The field digits of *id* are copied to *od*, with the copy controlled by the qualifier of *od*.

Fields in *id* that are not included by the *od* qualifier are disregarded.

Fields in *od* that are not present in *id* are filled in as follows:

- Fields to the left of the most significant field in *id* are filled in from the current time and date.
- Fields to the right of the least significant field in *id* are filled in with zeros.

Return Code

-1268 There is an invalid DATETIME qualifier.

Example Call

In these statements, a variable *xmas* is set up with the date of Christmas for the current year. The **dtextend** function is used to generate the current year.

```
$datetime work, xmas;
work.dt_qual = TU_DTENCODE(TU_MONTH, TU_DAY);
dtcvasc("12-25", &work);
xmas.dt_qual = TU_DTENCODE(TU_YEAR, TU_DAY);
dtextend(&work, &xmas);
```

Example

```
/*
 * dtextend.ec *

The following program illustrates the results of datetime extension.
The fields to the right are filled with zeros,
and the fields to the left are filled in from current date and time.
*/

#include <stdio.h>

#include datetime;

main()
{
    int x;
    char month_str[20], year_str[20];
    $datetime month to day month_dt;
    $datetime year to minute year_min;

    /* Assign value to month_dt and extend */

    if(x = dtcvasc("12-07", &month_dt))
    {
        printf("Error %d in dtcvasc\n", x);
    }
    else
    {
        if (x = dtextend(&month_dt, &year_min))
            printf("Error %d in datetime extension\n", x);
        else
        {
            dttoasc(&month_dt, month_str);
            dttoasc(&year_min, year_str);
            printf("Value of month_dt is: %s\n", month_str);
            printf("Value of year_min (month_dt extended)");
            printf(" from year to minute) is: %s\n", year_str);
        }
    }
}
```

Example Output

```
Value of month_dt is: 12-07  
Value of year_min (month_dt extended from year to minute) is: 1991-12-07 00:00
```

DTTOASC

Purpose

The **dttoasc** function converts the field values of a **datetime** variable to an ASCII string that conforms to ANSI SQL standards.

Syntax

```
int dttoasc(d,str)
    datetime_t *d;
    char *str;
```

d is the address of an initialized **datetime_t** variable.

str is the address of space for a string.

Usage

The digits of the variable fields are converted to ASCII and copied to the output with delimiters (hyphen, space, colon, or period) between them.

The output does not include the qualifier or the parentheses that are used to delimit a DATETIME literal in an SQL statement.

The output includes one byte for each delimiter (hyphen, space, colon, or period) plus the fields with the following sizes:

year	four digits
fraction of DATETIME	as specified by precision
all other fields	two digits

The maximum length of output is produced from a DATETIME qualified as *year to fraction(5)*. It contains 19 digits, 6 delimiters, and the terminating null, for a total of 26 bytes.

If the variable has not been initialized, the function returns an unpredictable value, but one not exceeding 26 bytes.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * dttoasc.ec *

The following program illustates the conversion of a datetime value
into an ASCII string in ANSI SQL format
*/

#include <stdio.h>

#include datetime;

main()
{
    char out_str[16];
    $datetime year to hour dt1;

    /* Initialize dt1 */
    dtcurrent(&dt1);

    /* Convert the internal format to ascii for displaying */
    dttoasc(&dt1, out_str);

    /* Print it out*/
    printf("\tThe value of dt1 (year to hour) is: %s\n", out_str);
}
```

Example Output

```
The value of dt1 (year to hour) is: 1991-10-26 14
```

DTTOFMTASC

Purpose

The **dttofmtasc** function converts a **datetime** variable to an ASCII string of a specified format.

Syntax

```
int dttofmtasc(d,str,strlen,fmtstr)
    dtime_t *d;
    char *str;
    int strlen;
    char *fmtstr;
```

<i>d</i>	is the address of an initialized dtime_t variable.
<i>fmtstr</i>	is the address of the format string, using the directives defined for the DBTIME environment variable. If this argument is null, the function uses the format specified by DBTIME.
<i>str</i>	is the address of space for the output string.
<i>strlen</i>	is the length of <i>str</i> .

Usage

If the variable is not initialized, the function returns an unpredictable value.

The output does not include the qualifier or the parentheses that are used to delimit a DATETIME literal in an SQL statement. The output format does not need to be identical to the input qualifier. When the output qualifier is different from the input qualifier, **dttofmtasc** performs extensions as if **dtextend** were called.

If *fmtstr* is null and DBTIME is not defined, the standard ANSI SQL format is used. In this instance, the **dttofmtasc()** routine sets the output to contain values for *year to second* in ANSI SQL format.

Return Codes

0	The conversion was successful.
<0	The conversion failed; check the text of the error message.

Example

```
/*
 * The input and output qualifiers are same (hour to second)
 */
$datetime hour to second x;
char buff[50];

/*
 * assume x is initialized to "01:30:20"
 * 'buff' will be set to "01 h 30 m 20 s"
 *
 * Use field-width specification to avoid leading zeros (E.g. %1H).
 */
dttofmtasc(&x, buff, sizeof(buff), "%H h %M m %S s");

/*
 * The input and output qualifiers are different
 * input qual : hour to second
 * output qual: year to second (ANSI SQL default qualifier)
 */
$datetime hour to second x;
char buff[50];

/*
 * assume x is initialized to "01:30:20"
 * 'buff' will be set to "XXXX-XX-XX 01:30:20"
 * Notice that the output has been extended, and year-month-day fields
 * are set to current year, month and day.
 */
dttofmtasc(&x, buff, sizeof(buff), (char *)0);
```

INCVASC

Purpose

The **incvasc** function converts a string that conforms to the ANSI SQL standard to an **interval** value.

Syntax

```
int incvasc(str,i)
    char *str;
    intrvl_t *i;
```

i is the address of an initialized **intrvl_t** variable.
str is the address of a string of digits and field delimiters.

Usage

The variable must be initialized to the desired qualifier.

The input string can have leading and trailing spaces. However, from the first significant digit to the last, the only characters accepted are digits and delimiters appropriate to the fields implied by the qualifier.

If the input string is acceptable, the value is set in the variable and the function returns zero. Otherwise, the function does not change the variable and returns a negative error code.

Return Codes

-1260	It is not possible to convert between the specified types.
-1261	There are too many digits in the first datetime or interval field.
-1262	There is a non-numeric character in datetime or interval .
-1263	A field in a datetime or interval is out of range.
-1264	There are extra characters at the end of a datetime or interval .
-1265	Overflow occurred on a datetime or interval operation.
-1266	A datetime or interval is incompatible for the operation.
-1267	The result of a datetime computation is out of range.
-1268	There is an invalid datetime qualifier.

Example

```
/*
 * incvasc.ec *

The following program converts ASCII strings into interval (intvl_t)
structure. It also illustrates error conditions involving invalid qualifiers
for interval values.
*/

#include <stdio.h>

#include datetime;

main()
{
    int x;
    $interval day to second in1;

    if(x = incvasc("20 3:10:35", &in1))
        printf("Result #1 failed with conversion error:%d\n",x);
    else
        printf("Result #1: successful conversion\n");

    /*
     * Note that the following literal string has a 26 in the hours field
     */
    if(x = incvasc("20 26:10:35", &in1))
        printf("Error %d in coverting interval #2\n", x);
    else
        printf("Result #2: successful conversion\n");

    /*
     * Try to convert using an invalid qualifier (YEAR to SECOND)
     */
    in1.in_qual = TU_IENCODE(4, TU_YEAR, TU_SECOND);
    if(x = incvasc("1991-02-11 3:10:35", &in1))
        printf("Error %d in coverting interval #3\n", x);
    else
        printf("Result #3: successful conversion\n");
}
```

Example Output

```
Result #1: successful conversion
Error -1263 in coverting interval #2
Error -1268 in coverting interval #3
```

INCVFMTASC

Purpose

The **incvfmtasc()** function converts a string with a specified format to an **interval** value.

Syntax

```
int incvfmtasc(str,fmtstr, i)
    char *str;
    char *fmtstr;
    intrvl_t *i;
```

<i>fmtstr</i>	is the address of the format string, using the directives defined for the DBTIME environment variable. If this argument is null, the function uses the format specified by DBTIME.
<i>i</i>	is the address of an initialized intrvl_t variable.
<i>str</i>	is the address of a string containing the interval.

Usage

The output variable must be initialized with the desired qualifier.

If the input string is acceptable, the value is set in the variable and the function returns zero. Otherwise, the function returns an error code and the output variable can contain unpredictable results.

If *fmtstr* is null, the function returns an error.

All fields in the **datetime/interval** input string must be contiguous when calling **incvfmtasc**. In other words, if the qualifier is *hour-to-second*, all values for *hour*, *minute*, and *second* must be specified somewhere in the string, or an error results.

The output qualifier need not be identical to the input qualifier, as specified by the format string. When the output qualifier is different from the input qualifier, **incvfmtasc** converts the result to appropriate units. However, both the input and the output must represent an interval with a span of *year-to-month* or *day-to-fraction*.

The input string can have leading and trailing spaces. However, from the first significant digit to the last, the only characters accepted are digits and delimiters appropriate to the fields implied by the format string.

The directives `%B`, `%b` and `%p` are not applicable in `cvfmtasc()`, since *month name* and *A.M./P.M.* information is not useful for representing intervals of time. Use the `%Y` directive if the interval is more than 99 years (`%y` can handle only two digits). Use `%H` for hours (not `%I`, since `%I` can handle only 12 hours).

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * The input and output qualifiers are same (day to minute).
 * 'x' will be set to "20 03:40"
 *
 * Note the absence of field-width and precision specification
 * in the input format string.
 */
$interval day to minute x;
incvfmtasc("20 days, 3 hours, 40 minutes",
           "%d days, %H hours, %M minutes", &x);

/*
 * The input and output qualifiers are different
 * input qual : day to minute
 * output qual: hour to second
 */

/*
 * Since the expected number of digits in the first field is more than 2
 * declare the variable 'in' with some maximum width for hours [5].
 */
$interval hour(5) to second x;

/*
 * 'x' will be set to "483:40:00".
 * Notice that "20 days and 3 hours" have become "483 hours" and
 * seconds field has been set to "00".
 */
incvfmtasc("20 days, 3 hours, 40 minutes",
           "%d days, %H hours, %M minutes", &x);
```

INTOASC

Purpose

The **intoasc** function converts the field values of an **interval** variable to an ASCII string that conforms to the ANSI SQL standard.

Syntax

```
int intoasc(i,str)
    intrvl_t *i;
    char *str;
```

i is the address of an initialized **intrvl_t** variable.

str is the address of space for a string.

Usage

The digits of the variable fields are converted to ASCII and copied to the output with delimiters (hyphen, space, colon, or period) between them.

The output does not include the qualifier or the parentheses that are used to delimit an interval literal in an SQL statement.

The output includes one byte for each delimiter (hyphen, space, colon, or period) plus the fields with the following sizes:

leading field	as specified by precision
fraction	as specified by precision
all other fields	two digits

The maximum length of output is produced from an **interval** qualified as *day(5) to fraction(5)*. It contains 16 digits, 4 delimiters, and the terminating null, for a total of 21 bytes.

If the variable is not initialized, the function returns an unpredictable value, but one not exceeding 21 bytes.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * intoasc.ec *

The following program illustrates the conversion of an interval (intvl_t)
into an ASCII string.
*/

#include <stdio.h>

#include datetime;

main()
{
    int x;
    char out_str[10];
    $interval day(3) to day in1;

    if(x = incvasc("3", &in1))
        printf("Initial conversion failed with error: %d\n",x);
    else
    {
        /* Convert the internal format to ascii for displaying */
        intoasc(&in1, out_str);
        printf("The value of in1 is '%s'\n", out_str);
    }
}
```

Example Output

```
The value of in1 is '  3'
```

INTOFMTASC

Purpose

The **intofmtasc()** function converts an **interval** variable to an ASCII string of a specified format.

Syntax

```
intofmtasc(i,str,strlen,fmtstr)
    intrvl_t *i;
    char *str;
    int strlen;
    char *fmtstr;
```

<i>i</i>	is the address of an initialized input intrvl_t variable.
<i>fmtstr</i>	is the address of the format string, using the directives defined for the DBTIME environment variable. If this argument is null, the function uses the format specified by DBTIME.
<i>str</i>	is the address of space for the output string.
<i>strlen</i>	is the length of <i>str</i> .

Usage

The output does not include the qualifier or the parentheses that are used to delimit an INTERVAL literal in an SQL statement.

The output qualifier need not be identical to the input qualifier, as specified by the format string. When the output qualifier is different from the input qualifier, **intofmtasc()** converts the result to appropriate units. However, both the input and the output must represent an interval with a span of *year-to-month* or *day-to-fraction*.

If the variable is not initialized, the function returns an unpredictable value.

If *fmtstr* is null, the **intofmtasc()** function returns an error.

The directives *%B*, *%b* and *%p* are not applicable in **intofmtasc**, since *month name* and *A.M./P.M.* information is not useful for representing intervals of time. Use the *%Y* directive if the interval is more than 99 years (*%y* can handle only two digits). Use *%H* for hours (not *%I*, since *%I* can handle only 12 hours).

If the input value and the format specification are acceptable, the output string is set and the function returns zero. Otherwise, the function returns an error code and the output string can contain unpredictable results.

Return Codes

0	The conversion was successful.
<0	The conversion failed.

Example

```
/*
 * The input and output qualifiers are same (day to minute)
 */
$interval day to minute x;
charbuff[50];

/*
 * Assume that 'x' has been initialized to "20 3:40".
 * 'buff' will be set to "20 days, 3 hours and 40 minutes to go"
 */
intofmtasc(&x, buff, sizeof(buff),
           "%1d days, %1H hours and %1M minutes to go");

/*
 * The input and output qualifiers are different
 * input qual : day to minute
 * output qual: hour to second
 */
$interval day to minute x;
charbuff[50];

/*
 * Assume that 'x' has been initialized to "20 3:40".
 * 'buff' will be set to "483 hours 40 minutes and 0 seconds to go"
 *
 * Notice that "20 days and 3 hours" have become "483 hours" and
 * the seconds field has been set to zero.
 */
intofmtasc(&x, buff, sizeof(buff),
           "%1H hours, %1M minutes and %1S seconds to go");
```

Working with Binary Large Objects

Chapter Overview	3
Programming with Blobs	3
Fields Common to All Data Locations	5
Locating Blobs in Memory	6
Reading a Blob into Memory	7
Writing a Blob from Memory	9
Locating Blobs in Open Files	10
Reading a Blob into an Open File	11
Writing a Blob from an Open File	12
Locating Blobs in Named Files	14
Reading a Blob into a Named File	15
Writing a Blob from a Named File	16
User-Programmed Location	17
User-Programmed Open Function	17
User-Programmed Close Function	18
User-Programmed Read Function	18
User-Programmed Write Function	19
LOC_DESCRIPTOR	19
Guide to <code>dispcat_pic</code>	22
Before Using <code>dispcat_pic</code>	22
Using the Conditional Display Logic	23
Loading the <code>cat_picture</code> Column	23
Using <code>blobload</code>	24
The <code>dispcat_pic</code> Program	26



Chapter Overview

This chapter covers the following topics:

- Programming with binary large objects (blobs)
- The locator structure
- Locating blobs in memory
- Locating blobs in an open file
- Locating blobs in a named file
- Locating blobs with user-written code

This chapter also contains an annotated example program that reads the **cat_descr** and **cat_picture** blob columns from the **catalog** table of the **stores5** database.

For information about the data types available in an **INFORMIX-ESQL/C** program, see Chapter 2 of this manual. For information about the **TEXT** and **BYTE** blob data types, as well as other SQL data types, see Chapter 3 of *The Informix Guide to SQL: Reference*.

Programming with Blobs

In an **ESQL/C** program, you use a locator structure to read or write blobs—columns having a **TEXT** or **BYTE** data type. The locator structure does not contain the blob data; it contains information about the size and location of the blob data. It is the host variable for **TEXT** and **BYTE** columns when they are stored in or retrieved from the database. It describes the source location when blob data is inserted into the database and it describes the destination when blob data is fetched.

The locator structure is defined in the **locator.h** header file. The following comments in the **locator.h** file specify the use of fields in the locator structure:

- USER indicates that the field is set by the user and inspected by the database server.
- SYSTEM indicates that the field is set by the database server and inspected by the user.
- INTERNAL indicates that the field is a work area for the database server.

Note that a portion of the locator structure is a **union** (overlapping variant structures). The variant in use depends on whether the object is located in memory or a file. You specify whether the object is located in memory or in a file by specifying the contents of the **loc_loctype** field.

Figure 6-1 shows the definition of the locator structure as it appears in the **locator.h** file. Notice the additional comments in the file itself.

```
typedef struct
{
    short loc_loctype;          /* USER: type of locator - see below */
    union                      /* variant on 'loc' */
    {
        struct                /* case LOCMEMORY */
        {
            long  lc_bufsize; /* USER: buffer size */
            char *lc_buffer;  /* USER: memory buffer to use */
            char *lc_currdata_p; /* INTERNAL: current memory buffer */
            int   lc_mflags;  /* INTERNAL: memory flags (see below) */
        } lc_mem;

        struct                /* cases LOCFNAME & LOCFILE */
        {
            char *lc_fname;   /* USER: file name */
            int   lc_mode;    /* USER: perm. bits used if creating */
            int   lc_fd;      /* USER: os file descriptor */
            long  lc_position; /* INTERNAL: seek position */
        } lc_file;
    } lc_union;

    long  loc_indicator;      /* USER SYSTEM: indicator */
    long  loc_type;           /* SYSTEM: type of blob */
    long  loc_size;           /* USER SYSTEM: num bytes in blob or -1 */
    int   loc_status;         /* SYSTEM: status return of locator ops */
    char *loc_user_env;       /* USER: for the user's PRIVATE use */
    long  loc_xfercount;      /* INTERNAL/SYSTEM: Transfer count */

    int (*loc_open)();        /* USER: open function */
    int (*loc_close)();       /* USER: close function */
    int (*loc_read)();        /* USER: read function */

    int (*loc_write)();       /* USER: write function */

    int  loc_oflags;          /* USER/INTERNAL: see flag definitions below */
} loc_t;

#define loc_fname      lc_union.lc_file.lc_fname
```

```

#define loc_fd          lc_union.lc_file.lc_fd
#define loc_position    lc_union.lc_file.lc_position
#define loc_bufsize     lc_union.lc_mem.lc_bufsize
#define loc_buffer      lc_union.lc_mem.lc_buffer
#define loc_currdata_p  lc_union.lc_mem.lc_currdata_p
#define loc_mflags      lc_union.lc_mem.lc_mflags

/* Enumeration literals for loc_loctype */

#define LOCMEMORY      1          /* memory storage */
#define LOCFNAME      2          /* File storage with file name */
#define LOCFILE       3          /* File storage with fd */
#define LOCUSER       4          /* User define functions */

/* passed to loc_open and stored in loc_oflags */
#define LOC_RDONLY    0x1        /* read only */
#define LOC_WRONLY    0x2        /* write only */

/* LOC_APPEND can be set when the locator is created
 * if the file is to be appended to instead of created
 */
#define LOC_APPEND    0x4        /* write with append */
#define LOC_TEMPFILE  0x8        /* 4GL tempfile blob */

/* LOC_USEALL can be set to force the maximum size of the blob to always be
 * used when the blob is an input source. This is the same as setting the
 * loc_size field to -1. Good for LOCFILE or LOCFNAME blobs only.
 */
#define LOC_USEALL    0x10       /* ignore loc_size field */
#define LOC_DESCRIPTOR 0x20      /* BLOB is optical descriptor */

/* passed to loc_open and stored in loc_mflags */
#define LOC_ALLOC     0x1        /* free and alloc memory */

#endif /* LOCATOR_INCL */

```

Figure 6-1 *Locator structure*

Fields Common to All Data Locations

The following fields are common to all data locations:

loc_indicator

A value of -1 in the **loc_indicator** field indicates a null value. The program can set it when storing a null; the database server sets it on a fetch.

loc_status

The database server sets the **loc_status** field to zero when an operation is successful and returns a negative value when an error occurs.

loc_type

The **loc_type** field specifies whether the variable is TEXT (SQLTEXT) or BYTE (SQLBYTES) type.

Locating Blobs in Memory

If you set **loc_loctype** to LOCMEMORY, the TEXT or BYTE data is stored in primary memory. The memory buffer is addressed by **loc_buffer** and **loc_bufsize** gives its size.

The **loc_size** field contains the size of the data in bytes or the value -1. The program sets **loc_size** when storing a blob to the database; the database server sets **loc_size** after fetching.

If **loc_bufsize** is set to -1 when the locator is used for a fetch, the database server uses **malloc()** to obtain memory to hold the data and sets **loc_buffer** and **loc_bufsize** in addition to **loc_size**. If you perform subsequent fetches and the size of the data increases, the existing buffer is freed and the necessary memory is allocated. This alters the memory address at which the blob is stored, so if you reference the address in your programs, your program logic must account for the address change.

If the data does not fit in a size of **loc_bufsize** on a fetch or select, **loc_status** is set to a negative return code and the actual size of the data is set in **loc_indicator**. If **loc_bufsize** is less than **loc_size** when a value is stored in the database, an error is returned.

Initialize **loc_oflags** with the proper flags, usually 0. Figure 6-1 lists possible values.

Reading a Blob into Memory

The following code excerpt from `getcd_me` reads the `cat_descr` TEXT column of the `catalog` table into memory and then displays it:

```

$long cat_num;
$loc_t cat_descr;
-
-
/*
    Prepare locator structure for select of cat_descr
*/
cat_descr.loc_loctype = LOCMEMORY;      /* set loctype for in memory */
cat_descr.loc_bufsize = -1;             /* let db get buffer */
cat_descr.loc_oflags = 0;               /* clear loc_oflags */
$select catalog_num, cat_descr         /* look up catalog number */
    into $cat_num, $cat_descr from catalog
    where catalog_num = $cat_num;
if(!err_chk("Select"))                 /* if not found */
{
    printf("\n\t** Cat_num %ld not found in catalog table **", cat_num);
    if(!more())                         /* More to do? */
        break;                          /* no, terminate loop */
    else
        continue;                       /* yes */
}
prdesc();                               /* if found, print cat_descr */

```

The program sets `loc_loctype` to `LOCMEMORY` so that the database server returns the `cat_descr` text in a memory buffer. The program sets `loc_bufsize` to `-1` so that the database server allocates the memory for the buffer. The program also sets `loc_oflags` to `0` because it does not use a file. The program calls `prdesc()` to display the text returned by the `SELECT` statement. The `prdesc()` function in the following example sets a pointer, `p`, to the address that is returned in `loc_buffer`. The size of the buffer is returned in `loc_bufsize`.

```

/* prdesc() prints cat_desc for a row in the catalog table */
prdesc()
{
    long size;
    char shdesc[81], *p;

    size = cat_descr.loc_size;           /* get size of data */
    printf("\nDescription for %ld:\n", cat_num);
    p = cat_descr.loc_buffer;           /* set p to buffer addr */
    /*
    print buffer 80 characters at a time
    */
    while(size >= 80)
    {
        ldchar(p, 80, shdesc);           /* mv from buffer to shdesc */
        printf("\n%80s", shdesc);       /* display it */
        size -= 80;                       /* decrement length */
        p += 80;                           /* bump p by 80 */
    }
    strncpy(shdesc, p, size);
    shdesc[size] = '\0';
    printf("%-s\n", shdesc);           /* display last segment */
}

```

The following command runs `getcd_me` for the `stores5` database. It displays the `cat_descr` column for a catalog number that the user inputs. The user's input and the resulting output from `cat_descr` are shown in the following example:

```

% getcd_me

        database stores5 . . .
        Enter cat_num: 10004

Description for 10004:
Jackie Robinson signature glove. Highest professional quality, used by National
League.

        More? (y/n) . . .

```

Writing a Blob from Memory

The `upcd_me` program updates the `cat_descr` TEXT column of the `catalog` table from a memory buffer containing text that the user inputs. You can run the `upcd_me` program and input text to update the `cat_descr` column as shown in the following example:

```
% upcd_me

      database: stores5 ...
      Enter cat_num: 10004

Description for 10004:

Jackie Robinson signature ball. Highest professional quality, used by National
League.

Update? (y/n) . . .y

Enter description (255 chars):
Jackie Robinson home run ball, signed, 1955.

More? (y/n) . . .n
```

The following code excerpt illustrates how `upcd_me` uses the locator structure to update `cat_descr` from the text that is stored in memory:

```
$long cat_num;
$loc_t cat_descr;
-
-
-
/*
    Update?
*/
while((ans[0] = LCASE(ans[0])) != 'y' && ans[0] != 'n')
{
    printf("\n\tUpdate? (y/n) . . ."); /* update description? */
    getans(ans, 1);
}
if(ans[0] == 'y') /* if yes */
{
    printf("\n\tEnter description (%d chars):\n\t", BUFFSZ - 1);
    /* Enter description */
    getans(ans, BUFFSZ - 1);
    cat_descr.loc_loctype = LOCMEMORY; /* set loctype for in memory */
    cat_descr.loc_buffer = ans; /* set buffer addr */
    cat_descr.loc_bufsize = BUFFSZ; /* set buffer size */
    /* set size of data */
    cat_descr.loc_size = strlen(ans) + 1;
    /* Update */
    $update catalog set cat_descr = $cat_descr
    where catalog_num = $cat_num;
```

The program calls `getans()` to copy the user's input into an `ans` array and then moves the address of the array to `loc_buffer`. The program moves the size of the buffer (`BUFSZ`) into `loc_bufsize`, the size of the blob (`strlen(ans)`) into `loc_size`, and then performs the update.

The following code excerpt illustrates the use of a locator structure in an INSERT statement:

```
$char name[20];
$loc_t photo;

photo.loc_loctype = LOCMEMORY; /* Photo resides in memory */
photo.loc_buffer = photo_ptr; /* pointer to where it is */
photo.loc_size = photo_len; /* length of image*/

$ INSERT INTO employee (name, badge_pic)
  VALUES ($name, $photo);
```

Locating Blobs in Open Files

If you set `loc_loctype` to `LOCFILE`, the blob data is located in a file that your program opened. The file descriptor of the open file is specified in `loc_fd`. The database server reads or writes the data from the current location in the file. When using an open file, you must place the host system file-open mode flags in `loc_oflags` using `LOC_ONLY`, `LOC_WONLY`, or `LOC_APPEND`.

When the database server stores data to the database from an open file, it reads `loc_size` bytes or to the end of the file, if `loc_size` contains `-1`. When fetching from the database, the database server writes all the data and sets the length in `loc_size`.

Reading a Blob into an Open File

The following code excerpt from the `getcd_of` program selects the `cat_descr` column into the file named by `argv[2]`:

```

$char db_stmnt[250];

$long cat_num;
$short stock_num;
$char manu_code[4];
$loc_t cat_descr;
$loc_t cat_picture;
$vvarchar cat_advert[256];
-
-
-
if ((fd = open(argv[2], O_WRONLY)) < 0) /* open output file */
{
    printf("\nCan't open file: %s\n", argv[1]);
    exit(1);
}
if (rstol(argv[1], &cat_num) /* cat_num string to long */
    {
    printf("\nUsage: prog dbname cat_num file_name\n-Illegal cat_num");
    exit(1);
    }

/*
Prepare locator structure for select of cat_descr
*/
cat_descr.loc_loctype = LOCFILE; /* set type to open file */
cat_descr.loc_fd = fd; /* supply file descriptor*/
cat_descr.loc_oflags = LOC_APPEND; /* set file-open mode write */
$select catalog_num, cat_descr /* verify catalog number */
into $cat_num, $cat_descr from catalog
where catalog_num = $cat_num;
if (!err_chk("Select")) /* if error, display and quit */
{
    printf("\n\t** Cat_num %ld not found in catalog table **", cat_num);
    exit();
}
exit();

```

The `getcd_of` program opens the file that is named by the third argument on the command line. It then converts the catalog number, given as `argv[1]`, to a long integer to match it against the `catalog_num` column of the `catalog` table. To prepare the locator structure for the `SELECT` statement, `getcd_of` moves `LOCFILE` to `loc_loctype` to tell the database server to place the text for `cat_descr` in the open file. It moves `LOC_APPEND` to `loc_oflags` to specify that the data should be appended to any existing data in the file.

Writing a Blob from an Open File

The **updc_d_of** program updates the **cat_descr** column from text located in an open file. The input file for **updc_d_of** contains a series of records, each one consisting of a catalog number and the associated text to update the **cat_descr** column. The input file is organized as follows:

```
\10001\  
Dark brown leather first baseman's mit. Specify right-handed or left-handed.  
  
\10002\  
Babe Ruth signature glove. Black leather. Infield/outfield style. Specify right-  
or left-handed.  
-  
-
```

The following code excerpt from `upcd_of` illustrates the use of the locator structure to update the `cat_descr` column of the `catalog` table from an open file:

```

$long cat_num;
$short stock_num;
$char manu_code[4];
$loc_t cat_descr;
$loc_t cat_picture;
$vvarchar cat_advert[256];
-
-
-
if ((fd = open(argv[1], O_RDONLY)) < 0)    /* open input file */
{
    printf("\nCan't open file: %s\n", argv[1]);
    exit(1);
}
while(getcat_num(fd, line, sizeof(line))) /* get cat_num line from file */
{
    line[6] = '\0';                        /* replace / with null */
    rstol(&line[1], &cat_num);             /* cat_num string to long */
    flpos = lseek(fd, 0L, 1);
    length = getdesc_len(fd);
    flpos = lseek(fd, flpos, 0);
    /*
        lookup cat_num in catalog table
    */
    $select catalog_num into $cat_num from catalog
        where catalog_num = $cat_num;
    if(!err_chk("Select"))                /* if not found */
    {
        printf("\n\t** Cat_num %ld not found in catalog table **", cat_num);
        -
    }
    -
    -
    /*
        if found
    */
    cat_descr.loc_loctype = LOCFILE;        /* update from open file */
    cat_descr.loc_fd = fd;                 /* load file descriptor */
    cat_descr.loc_oflags = LOC_RDONLY;     /* set file-open mode (read) */
    cat_descr.loc_size = length;           /* set size of blob */
    /*
        update cat_descr column of catalog table
    */
    $update catalog set cat_descr = $cat_descr
        where catalog_num = $cat_num;
    err_chk("Update");
}
$close database;

```

The `upcd_of` program opens the input file named by `argv[1]`, calls `getcat_num()` to read a catalog number, and then calls `getdesc()` to determine the length of the text that updates `cat_descr`. The program then performs a select to verify that the catalog number exists in the `catalog` table. If it does, `upcd_of` prepares the locator structure to update `cat_descr` from the text in

the open file. It sets **loc_loctype** to LOCFILE to inform the database server that **cat_descr** is to be updated from an open file. The program then moves **fd**, the file descriptor for the input file, to **loc_fd**, and sets **loc_oflags** to LOC_RDONLY, the file-open-mode flag for read-only. Finally, it moves **length**, the length of the incoming text for **cat_descr**, to **loc_size**, and performs the update.

Locating Blobs in Named Files

If you set **loc_loctype** to LOCFNAME, the database server locates the blob data in a file specified by name. To provide the database server with the name of the file, place the address of the pathname string in **loc_fname**. You must also set the host system file-open-mode flags in **loc_oflags** using LOC_RDONLY, LOC_WONLY, or LOC_APPEND. The database server opens the file per the mode flags, sets **loc_fd**, and then proceeds as if the file were opened by your program. When the database server stores data to the named file, it reads **loc_size** bytes or to the end of the file, if **loc_size** contains -1. Fetching a null (or empty) blob column into a named file that already exists truncates the file.

Reading a Blob into a Named File

The following code excerpt from `getcd_fn` executes a select to read the `cat_descr` TEXT column from the `catalog` table and write it to the file named by the third argument on the command line, `argv[2]`:

```

$long cat_num;
$loc_t cat_descr;
-
-
-
if(argc == 4)
{
    sprintf(db_stmnt, "database %s", argv[1]);
    printf("\n%s", db_stmnt);
    $prepare open_db from $db_stmnt;
    err_chk("prepare database");
    $execute open_db;
    err_chk(db_stmnt);
    ++argv;
}
else
{
    $database stores5;
    err_chk("database");
    printf("\nOpening database: stores2");
}
if(rstol(argv[1], &cat_num)                /* cat_num string to long */
{
    printf("\nUsage: prog cat_num file_name\n-Illegal cat_num");
    exit(1);
}

/*
   Prepare locator structure for select of cat_descr
*/
cat_descr.loc_loctype = LOCFNAME;           /* set loctype for in memory */
cat_descr.loc_fname = argv[2];             /* load the addr of file name */
cat_descr.loc_oflags = LOC_APPEND;        /* set loc_oflags to append */
$select catalog_num, cat_descr            /* verify catalog number */
    into $cat_num, $cat_descr from catalog
    where catalog_num = $cat_num;
if(!err_chk("Select"))                    /* if error, display and quit */
    printf("\n\t** Cat_num %ld not found in catalog table **", cat_num);
$close database;
}

```

The `getcd_fn` program either opens the database named on the `argv[1]` command line, if one is given, or it opens the `stores5` database. Next, it converts the catalog number from the `argv[1]` command line to a long integer, the data type of the `catalog_num` column. Then, it prepares the locator structure for `cat_descr`. It moves `LOCFNAME` to `cat_descr.loc_loctype` to tell `INFORMIX-OnLine` to place the contents of `cat_descr` in the file named in `argv[2]`. It moves `LOC_APPEND` to `cat_descr.loc_oflags`, the file-open-mode flags, to tell `INFORMIX-OnLine` to append it to the existing file. Then, the program executes the select to retrieve the row.

Writing a Blob from a Named File

The following code excerpt from the `updc_d_nf` program updates the `cat_descr` column in the `catalog` table from text in a named file:

```

$long cat_num;
$loc_t cat_descr;

-
-
-
    cat_descr.loc_loctype = LOCMEMORY;           /* set loctype for in memory */
    cat_descr.loc_bufsize = -1;                 /* let server get memory */
    $select catalog_num, cat_descr              /* verify catalog number */
        into $cat_num, $cat_descr from catalog
        where catalog_num = $cat_num;
    if(!err_chk("Select"))                      /* if error, display and quit */
        printf("\n\t** Cat_num %ld not found in catalog table **", cat_num);
    prdesc();                                   /* print current cat_descr */
/*
    Update?
*/
while((ans[0] = LCASE(ans[0])) != 'y' && ans[0] != 'n')
{
    printf("\n\tUpdate? (y/n) . . .");
    scanf("%1s", ans);
}
if(ans[0] == 'y')
{
    cat_descr.loc_loctype = LOCFNAME;           /* set type to named file */
    cat_descr.loc_fname = argv[2];             /* supply file name */
    cat_descr.loc_oflags = LOC_RDONLY;         /* set file-open mode (read) */
    $update catalog
        set cat_descr = $cat_descr             /* update cat_descr column */
        where catalog_num = $cat_num;
    err_chk("Update");                          /* check status */
}

```

The `updc_d_nf` program first performs a select for the catalog number given by the user as the second argument on the `argv[1]` command line. The select returns the `catalog_num` and `cat_descr` columns. The `prdesc()` function then displays the current content of `cat_descr`. The program asks whether the user wants to update the description. If the user answers yes (`ans[0] == 'y'`), `updc_d_nf` prepares the locator structure to update `cat_descr` from the text in the file named by the third argument on the `argv[2]` command line. The program sets `cat_descr.loc_loctype` to `LOCFNAME` to indicate that the source of the update text is a named file. It sets `cat_descr.loc_fname` to `argv[2]` and then sets `cat_descr.loc_oflags` to `LOC_RDONLY` to tell `INFORMIX-OnLine` to open the file in read-only mode. Then it performs the update.

The following example shows the command to run the **updcn_nf** program. The command line specifies the catalog number for the row to be updated and the name of the file that stores the update text. The program displays the current contents of the **cat_descr** column and asks whether the user wants to update the column.

```
% updcn_nf 10002 catdescr
      database stores5 ...
Description (catalog_num: 10002):
Babe Ruth signature glove. Black leather.
      Update? (y/n) . . .y
%
```

The contents of the **catdescr** file are as follows:

```
Babe Ruth signature glove. Black leather. Infield/outfield style. Specify right-
or left-handed.
```

User-Programmed Location

You can set **loc_loctype** to **LOCUSER** so that the C functions that you supply in your **ESQL/C** program completely control the blob data. In **loc_open**, **loc_close**, **loc_read**, and **loc_write**, you supply the addresses of functions to open, close, read and write the data. The database server then calls these functions, as required, to handle the data.

Each of the functions receives the address of the **loc_t** structure as its first or only parameter. You can use the **loc_usr_env** field for these functions. For example, you can set **loc_usr_env** to the address of a common work area. In addition, the **loc_xfercount** and all the fields of the **loc_union** substructure are available for these functions.

User-Programmed Open Function

The following skeleton function is an example of a user-programmed open function. The open function receives two parameters from the database server: the address of the locator structure, **loc_t**, and a flag that contains

LOC_RDONLY if the database server sends input to the database, or LOC_WONLY if the database server sends output, or fetches, from the database. The function returns 0 for success and -1 for failure.

```
openblob(adloc, oflags)
    loc_t*adloc;
    int oflags;
{
    adloc->loc_status = 0;
    adloc->loc_xfercount = 0L;
    if (0==(oflags & adloc->loc_oflags))
        return(-1);
    if (oflags & LOC_RDONLY)
        /** prepare for store to db ***/
    else
        /** prepare for fetch to program ***/
    return(0);
}
```

User-Programmed Close Function

When a transfer to or from the database server is complete, the close function that you supply is called. The following skeleton function is an example of a user-programmed close function. It sets the status in the **loc_status** field of the locator structure and sets the number of bytes transferred into **loc_size**.

```
closeblob (adloc)
    loc_t*adloc;
{
    adloc->loc_status = 0;
    if (adloc->loc_oflags & LOC_WONLY) /* if fetching */
    {
        adloc->loc_indicator = 0; /* clear indicator */
        adloc->loc_size = adloc->loc_xfercount;
    }
    return(0);
}
```

User-Programmed Read Function

To store data in the database, the database server calls on the supplied read function for data. The read function receives three arguments: the address of the locator structure, the address of the buffer to receive the data from your program, and the number of bytes to read. The database server takes data in segments of some maximum size until it reads all the data. You set the size of the data in **loc_size** when you set up the locator structure for the blob column. If the program set **loc_size** to -1, the database server reads in data until the read function returns an end-of-file (EOF) signal. The read function must return the count of bytes it transferred. When the count is not equal to the

number of bytes requested, the database server assumes an EOF signal. The following skeleton function is an example of a user-programmed read function:

```
readblob(adloc, bufp, ntoread)
    loc_t*adloc;
    char*bufp;
    int ntoread;
{
    int ntoxfer;

    ntoxfer = ntoread;
    if (adloc->loc_size != -1)
        ntoxfer = min(ntoread,
            adloc->loc_size - adloc->loc_xfercount);

    /** transfer "ntoread" bytes to *bufp ***/

    adloc->loc_xfercount += ntoxfer;
    return(ntoxfer);
}
```

User-Programmed Write Function

To fetch data from the database, the database server calls on the write function that you supplied to dispose of the data. The function receives three parameters from the database server: the address of the locator structure, the address of the buffer where the data is stored, and the number of bytes to write. The database server can call the function more than once per data item, receiving the address and length of a segment of data each time. Returning a non-zero value indicates an error. The following skeleton function is an example of a user-programmed write function:

```
writeblob(adloc, bufp, ntowrite)
    loc_t*adloc;
    char*bufp;
    int ntowrite;
{
    /** transfer ntowrite bytes from *bufp ***/
    adloc->loc_xfercount += ntowrite;
    return(0);
}
```

LOC_DESCRIPTOR

When reading or writing a blob column that is stored on a write-once-read-many (WORM) optical disk, you can manipulate only the blob descriptor by setting **loc_oflags** to LOC_DESCRIPTOR. LOC_DESCRIPTOR should only be used in conjunction with blobs that are stored on WORM optical media.

Data rows that include blob data do not include the blob data in the row itself. Instead, the data row contains a 56-byte blob descriptor that includes a forward pointer (rowid) to the location where the first segment of blob data is stored. The descriptor can point to a dbspace blob-page, a blob-space blob-page, or a platter in an optical storage subsystem. See *The INFORMIX-OnLine Administrator's Guide* for details.

When a blob is stored on a WORM optical-storage subsystem, you can conserve storage space on the WORM optical disk by having a single physical blob reside in more than one table. The LOC_DESCRIPTOR flag allows you to do this by enabling you to migrate a blob descriptor, rather than the blob itself, from one table to another.

The following example selects the **stock_num**, **manu_code**, **cat_descr**, and **cat_picture** columns from the **catalog** table of the named database. The program uses the **descr** function expression to retrieve the blob descriptor, rather than the blob itself, for the **cat_picture** column. It then sets LOC_DESCRIPTOR in the **loc_oflags** of the **cat_picture** locator structure to signal that the blob descriptor, rather than the blob, is to be inserted into the **cat_picture** column of the **pictures** table. The result is that the **cat_picture** columns in both the **catalog** and **pictures** tables refer to a single set of physical blobs.

```
#include <stdio.h>
#include locator.h;

$char db_stmt[250];

char errmsg[400];

$long cat_num;
$short stock_num;
$char manu_code[4];
$loc_t cat_descr;
$loc_t cat_picture;
$vvarchar cat_advert[256];

main(argc, argv)
int argc;
char *argv[];
{
    if (argc > 2) /* correct no. of args? */
    {
        printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
            argv[0]);
        exit(1);
    }
    if(argc == 2)
    {
        sprintf(db_stmt, "database %s", argv[1]);
        printf("\n%s", db_stmt);
        $prepare open_db from $db_stmt;
        err_chk("prepare database");
        $execute open_db;
    }
    else
    {
```

```

        $database stores5;
        err_chk("database");
        printf("\nOpening database: stores5");
    }
$declare catcurs cursor for          /* setup cursor for select */
select stock_num, manu_code, cat_descr, descr(cat_picture)
from catalog where cat_picture is not null;
/*
    Prepare locator structures cat_descr(TEXT blob) and
    cat_picture (TEXT blob that is the blob descriptor).
*/
cat_descr.loc_loctype = LOCMEMORY;      /* set loctype for in memory */
cat_picture.loc_loctype = LOCMEMORY;    /* set loctype for in memory */
while(1)
{
    /*
        Let db get buffers and set loc_buffer (buffer for blob descriptor)
        and loc_bufsize (size of buffer)
    */
    cat_descr.loc_bufsize = -1;
    cat_picture.loc_bufsize = -1;
    /*
        Select row from catalog table (descr() returns TEXT blob descriptor
        for cat_picture. For cat_descr, the actual blob is returned.)
    */
    $fetch catcurs into $stock_num, $manu_code, $cat_descr,
        $cat_picture;
    if(!err_chk("Fetch"))                /* end of data */
        break;
    /*
        Set LOC_DESCRIPTOR in loc_oflags to indicate blob descriptor
        is being inserted rather than blob data.
    */
    cat_picture.loc_oflags |= LOC_DESCRIPTOR;
    /*
        Insert
    */
    $insert into pictures values ($stock_num, $manu_code,
        $cat_descr, $cat_picture);
    err_chk("Insert");
    printf("Insert failed for stock_num %d, manu_code %s", stock_num,
        manu_code);
}
}

/*
err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode < 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);

        exit(1);
    }
    return((sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1);
}

```

Note: Rarely must you set `loc_oflags` to `LOC_DESCRIPTOR`. You can achieve the same result without setting `loc_oflags` to `LOC_DESCRIPTOR`. The following SQL statement accomplishes the same task as the preceding example.

```
$INSERT INTO pictures (stock_num, manu_code, cat_descr, cat_picture)
SELECT stock_num, manu_code, cat_descr, DESCR(cat_picture)
FROM catalog
WHERE cat_picture IS NOT NULL
```

Guide to *dispcat_pic*

The `dispcat_pic` program annotated on the following pages uses the `ESQL/C` `loc_t` locator structure to retrieve two blob columns. The program retrieves the `cat_descr` TEXT blob column and the `cat_picture` BYTE blob column from the `catalog` table of the `stores5` demonstration database. See “The Demonstration Database” in the Introduction for information on creating the demonstration database.

The `dispcat_pic` program allows you to select a database from the command line in case you created the `stores5` database under a different name. If no database name is given, `dispcat_pic` opens the `stores5` database. The program prompts the user for a `catalog_num` value and performs a select to read the `description` column from the `stock` table and the `catalog_num`, `cat_descr`, and `cat_picture` columns from the `catalog` table. If the database server finds the catalog number and the `cat_picture` column is not null, it writes the `cat_picture` column to a temporary file. If the program is compiled with the conditional `cat_picture` display logic, the program forks a second process and executes a SUN `screenload` system utility to display the raster image from the temporary file. In all cases, if the select succeeds, the program displays the `catalog_num`, `cat_descr`, and `description` columns. Finally, the program deletes the temporary file it created for `cat_picture` and allows the user to enter another `catalog_num` value or terminate the program.

Before Using *dispcat_pic*

The `dispcat_pic` source program is provided with `INFORMIX-ESQL/C` in the `$INFORMIXDIR/esqlc/demo` directory so that you can compile and run it. The program includes conditional display logic to illustrate the logic you can use to display a graphic image from a `BYTE` type column. As provided, however, the conditional logic can only display the content of the `cat_picture` column on a Sun workstation, using the Sun `screenload` utility program. To display the `cat_picture` column on another `ESQL/C` platform, you must sub-

stitute display logic that runs on that platform. If the program is compiled normally, without the conditional display logic, the executable program displays only the **catalog_num** and **cat_descr** columns from the **catalog** table and the **description** column from the **stock** table of the **stores5** database. Since these columns store text, they can be displayed on any **ESQL/C** platform.

Use the following command to compile **dispcat_pic** *without* the conditional display logic:

```
esql -o dispcat_pic dispcat_pic.ec
```

The **-o dispcat_pic** option causes the executable program to be named **dispcat_pic**. Without the **-o** option, the name of the executable program defaults to **a.out**. See “Compiling INFORMIX-ESQL/C Programs” on page 1-20 for more information on the **esqlc** preprocessor command.

Using the Conditional Display Logic

Use the following command to compile **dispcat_pic** with the conditional **cat_picture** display logic for a Sun workstation:

```
esql -o dispcat_pic -DSUNVIEW dispcat_pic.ec
```

The **-DSUNVIEW** option causes the conditional display logic to be compiled. The **-o dispcat_pic** option causes the resulting executable program to be named **dispcat_pic**. Without the **-o dispcat_pic** option, the name of the program defaults to **a.out**. See “Compiling INFORMIX-ESQL/C Programs” on page 1-20 for more information on the **esqlc** preprocessor command.

*Note: For simplicity, the program is not designed to display the **cat_picture** raster image under SunView or any other windowed environment. If you are displaying the **cat_picture** column, for best results, the program should be run directly from the SunOS command line, not through the window manager.*

Loading the *cat_picture* Column

When the **catalog** table is created as part of the **stores5** demonstration database, the **cat_picture** column for all rows is set to null. **INFORMIX-ESQL/C** provides five graphic images and the **blobload.ec** program to load five rows of the **catalog** table with graphic images that can be displayed. The images must be loaded to the **cat_picture** **BYTE** column.

Informix provides the five **cat_picture** images in two formats: Sun raster image format (**.rs** files) and Graphics Interchange Format (**.gif** files). If you are compiling **dispcat_pic.ec** with the conditional display logic, you must use the Sun raster image files to load the five **cat_picture** columns. The names of the raster image files and the images that they contain are as follows:

Image File	Image
cn_10001.rs	baseball glove
cn_10027.rs	bicycle crankset
cn_10031.rs	bicycle helmet
cn_10046.rs	golf balls
cn_10049.rs	running shoe

The numeric portion of the image filename is the **catalog_num** value for the row of the **catalog** table to which the image should be updated. For example, **cn_10027.rs** should be updated to the **cat_picture** column of the row where 10027 is the value of **catalog_num**.

ESQL/C also provides the images in **.gif** files to provide them in a standard format that can be displayed on other ESQL/C platforms or translated into other formats using filter programs supplied by other vendors. The names of the **.gif** image files and the images they contain are as follows:

Image File	Image
cn_10001.gif	baseball glove
cn_10027.gif	bicycle crankset
cn_10031.gif	bicycle helmet
cn_10046.gif	golf balls
cn_10049.gif	running shoe

*Note: The **dispcat_pic** program, as delivered, only displays Sun raster images. To display the Graphics Interchange Format, or any other format, you must modify **dispcat_pic**, substituting display logic for the format you are using.*

Using blobload

Prior to running **dispcat_pic** to display these images, you must perform the following steps to load the images to the **catalog** table:

1. Compile the **blobload.ec** program.

Use the following command to compile the **blobload** program:

```
esql -o blobload blobload.ec
```

2. Run the **blobload** program to load each image to its proper **cat_picture** column.

Enter **blobload** on the UNIX command line without any arguments, as follows:

```
blobload
```

This displays the following message that describes the command-line arguments that **blobload** expects:

```
Sorry, you left out a required parameter.
Usage: blobload {-i | -u}      -- choose insert or update
        -f filename          -- file containing the blob data
        -d database_name     -- database to open
        -t table_name        -- table to modify
        -b blob_column       -- name of target column
        -k key_column key_value -- name of key column and a value
        -v                   -- verbose documentary output
```

All parameters except -v are required.

Parameters may be given in any order.

As many as 8 -k parameter pairs may be specified.

Run **blobload** with the **-u** option to update the Sun raster images to the **catalog** table. Run the **blobload** program once for each image file that you want to update. For example, the following command loads the content of **cn_10027.rs** into the **cat_picture** column of the row for **catalog_num** 10027. The **catalog_num** column is the key column in the **catalog** table.

```
blobload -u -f cn_10027.rs -d stores5 -t catalog -b cat_picture -k catalog_num
10001
```

Use the same command to update each of the four remaining image files, substituting the name and corresponding **catalog_num** value of the image file that you want to load.

You can use the **blobload** program to load the .gif files to the **catalog** table in the same manner that it is used to load the .rs files.

The *dispcat_pic* Program

```
1 #include <stdio.h>
2 $include sqltypes.h;
3 $include locator.h;
4
5 #define LCASE(c) (isupper(c) ? tolower(c) : c)
6 #define BUFFSIZ 256
7
8 extern int errno
9
10 char errmsg[512];
11 $char db_name[20];
12
13 $char description[16];
14 $long cat_num;
15 $short stock_num;
16 $char manu_code[4];
17 $loc_t cat_descr;
18 $loc_t cat_picture;
19 $varchar cat_advert[256];
20
21 char cpfl[18];
22
```

Continued on page 6-28

Lines 1 to 4

The `#include <stdio.h>` statement includes the `stdio.h` UNIX header file from the `/usr/include` directory. The `stdio.h` file enables `dispcat_pic` to use the standard C I/O library. The `sqltypes.h` file is an `INFORMIX-ESQL/C` header file that defines names for integer values that identify SQL and C data types. The `locator.h` file is an `ESQL/C` header file that contains the definition of the locator structure. The locator structure is the host variable for a blob column that is retrieved from or stored to the database. The locator structure has a `loc_t` typedef. The program uses the locator structure to specify blob size and location. (Lines 17 and 18 specify the locator structure as the data type for host variables that receive data for the `cat_descr` and `cat_picture` blob columns.)

Lines 5 to 11

Line 5 defines `LCASE`, a macro that converts an uppercase character to a lowercase character. Line 6 defines `BUFFSIZ` to be the number 256. The program uses `BUFFSIZ` to specify the size of arrays that store input from the user. Line 8 defines `errno`, an external integer where system calls store an error number to indicate the specific error that occurred. Line 10 defines `errmsg[]`, a character array that receives the text of error messages from `rgetmsg()`. See the

err_chk() function (lines 192-204) to see how **errmsg[]** is used. The **db_name[]** character array is a host variable that stores the database name, if the user names the database on the command line.

Lines 13 to 20

These lines define host variables for the **description** column of the **stock** table and for all columns from the **catalog** table. A host variable receives data that is fetched from a table and supplies data that is written to a table. Note that lines 17 and 18 define locator structures for the two blob columns of the catalog table, **cat_descr** and **cat_picture**.

Line 21

The **cpfl** character array stores the name of the temporary file where the raster image of **cat_picture** is written by the database server.

```
23 main(argc, argv)
24 int argc;
25 char *argv[];
26 {
27     char ans[BUFSZ], db_stmt[50];
28
29     if (argc > 2)
30     {
31         printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
32             argv[0]);
33         exit(1);
34     }
35     if(argc == 2)
36     {
37         strcpy(db_name, argv[1]);
38         $database $db_name;
39         sprintf(db_stmt, "database %s", argv[1]);
40         err_chk(db_stmt);
41         printf("\nOpening database: %s ...", db_stmt);
42     }
43     else
44     {
45         $database stores5;
46         err_chk("database");
47         printf("\nOpening database: stores5");
48     }
```

Continued on page 6-30

Lines 23 to 25

The `main()` function is the point at which program execution begins. The first argument, `argc`, is an integer that gives the number of arguments submitted on the command line. The second argument, `argv[]`, is a pointer to an array of character strings that contain the command-line arguments. The `dispcat_pic` program expects only the `argv[1]` argument, which is optional. It is the name of the database to access. If `argv[1]` is not present, the program opens the `stores5` database.

Line 27

Line 27 defines variables that are local in scope to the `main()` function. The `ans[BUFSZ]` array is the buffer that receives input from the user, namely the catalog number for the associated `cat_picture` column. The `db_stmt[50]` array is used to pass the DATABASE statement to the `err_chk()` function.

Lines 29 to 42

These lines handle the command-line arguments. Line 29 checks to see whether more than two arguments are entered on the command line. If so, `dispcat_pic` displays a message to show the arguments that it expects and

then it terminates. Line 35 tests to see whether the number of command-line arguments is equal to 2. If so, **dispcat_pic** assumes that the second argument, **argv[1]**, is the name of the database that the user wants to open. Line 37 copies the name of the database from the **argv[1]** command line, into the **db_name** host variable. Then, on line 38, the program opens the specified database with the DATABASE statement, using **db_name**. On line 39, the program reproduces the DATABASE statement in the **db_stmt[]** array. It does so for the sake of the **err_chk()** call on line 40, which takes as its argument the name of a statement. Line 41 displays the name of the database that is opened.

Lines 43 to 48

The **else** on line 43 handles cases in which a database name is *not* entered on the command line. If a database name is not given on the command line, line 45 executes the DATABASE statement for the default database, **stores5**. Line 46 calls the **err_chk()** function to check on the outcome and line 47 displays the name of the database that is opened.

```
49     while(1)
50     {
51         strcpy(cpfl, "./cpfl.XXXXXX");
52         if(!mktemp(cpfl))
53         {
54             printf("\n\tCan't create temp file for picture");
55             exit();
56         }
57         printf("\n\tEnter cat_num: ");
58         if(!getans(ans, 6))
59             continue;
60         if(rstol(ans, &cat_num))
61         {
62             printf("\n\tIllegal cat_num %s", ans);
63             exit();
64         }
65         cat_descr.loc_loctype = LOCMEMORY;
66         cat_descr.loc_bufsize = -1;
67         cat_descr.loc_oflags = 0;
```

Continued on page 6-32

Lines 49 to 56

The `while(1)` on line 49 begins the main processing loop within `dispcat_pic`. The first operation that the loop performs is to create a uniquely named file to receive `cat_picture`. Line 51 copies the name of the temporary file to the `cpfl[]` array. Line 52 calls the UNIX `mktemp()` function to create a unique filename, passing `cpfl[]` as the argument. If `mktemp()` cannot create a unique filename, it returns 0; lines 54 and 55 display a message to the user and exit.

Lines 57 to 59

Line 57 prompts the user to enter a catalog number for the corresponding `cat_picture` column that the user wants to see. Line 58 calls `getans()` to receive the catalog number that the user inputs. The arguments for `getans()` are the address in which the input should be stored, `ans[]`, and the maximum length of the input that is expected, including the terminating null. If the input is unacceptable, `getans()` returns 0 and line 59 returns control to the `while` at the top of the loop, causing the prompt for the catalog number to be redisplayed.

Lines 60 to 64

Line 60 calls the `ESQL/C` library function `rstol()` to convert the input string to a `long` data type to match the data type of the `catalog_num` column. If `rstol()` returns a nonzero value, the conversion failed and lines 62 and 63, respectively, display a message to the user and exit.

Lines 65 to 67

Line 65 sets **loc_loctype** in the **cat_descr** locator structure to LOCMEMORY to tell the database server to load the data for **cat_descr** into memory. Line 66 sets **loc_bufsize** to -1 so that the database server allocates a memory buffer to receive the data for **cat_descr**. If the select is successful, the database server returns the address of the buffer in **loc_buffer**. Line 67 sets the **loc_oflags** file-open-mode flags to 0 because the program retrieves the blob into memory rather than a file.

```
68     cat_picture.loc_loctype = LOCFNAME;
69     cat_picture.loc_fname = cpfl;
70     cat_picture.loc_oflags = LOC_WONLY;
71     cat_picture.loc_size = -1;
72     $select description, catalog_num, cat_descr, cat_picture
73     into $description, $cat_num, $cat_descr, $cat_picture
74     from stock, catalog
75     where catalog_num = $cat_num and
76     catalog.stock_num = stock.stock_num and
77     catalog.manu_code = stock.manu_code;
78     if(!err_chk("Select"))
79     {
80         printf("\n\t** Cat_num %ld not found in catalog table **", cat_num);
81         printf("\n\t** OR item not found in stock table **");
82         if(!more())
83             break;
84         continue;
85     }
86     if(cat_picture.loc_indicator == -1)
87         printf("\n\t\t** No cat_picture for this catalog_num **\n");
```

Continued on page 6-34

Lines 68 to 71

Lines 68 to 71 prepare the locator structure to retrieve the **cat_picture** column into a named file. Line 68 moves LOCFNAME to **loc_loctype** to tell the database server to load the data for **cat_descr** into a named file. Line 69 moves the address of the **cpfl** filename into **loc_fname**. Line 70 moves the LOC_WONLY value into the **loc_oflags** file-open-mode flags to tell the database server to open the file for writing only.

Lines 72 to 77

Lines 72 to 77 perform the select to retrieve the **catalog_num**, **cat_descr**, and **cat_picture** columns from the **catalog** table and the **description** column from the **stock** table for the catalog number entered by the user. The select checks to see whether the **stock_num** and **manu_code** for the selected row in the **catalog** table also exist in the **stock** table. It does this because the **catalog** table should not contain a row that does not have a corresponding row in the **stock** table.

Lines 78 to 85

Lines 78 to 85 check the outcome of the select and handle a not-found condition. Line 78 calls the **err_chk()** function that checks the **sqlca.sqlcode** value to determine whether **INFORMIX-OnLine** processed the SQL statement successfully. If the **err_chk()** function returns 0 to indicate that the row was not found, lines 80 and 81 display a message to that effect. Line 82 calls **more()** to

ask whether the user wants to continue. If the user answers **n** for no, line 83 executes a **break** to terminate the main processing loop and transfer control to line 96, which closes the database prior to program termination.

Lines 86 to 87

Line 86 checks **cat_picture.loc_indicator** for a value of -1 to determine whether the **cat_picture** column contains a null. If so, line 87 informs the user that the **cat_picture** column for the given catalog number does not contain a picture. The program then continues to line 90 to display the other columns that were returned.

```
88         else
89             display_picture();
90         printf("Item %d: %s\n", cat_num, description);
91         prdesc();
92         unlink(cpfl);
93         if(!more())
94             break;
95     }
96     $close database;
97 }
```

Continued on page 6-36

Lines 88 and 89

Lines 88 and 89 handle cases in which **cat_picture** is *not* null. Line 89 calls the **display_picture()** function to display the **cat_picture** data that the database server wrote to a raster image file.

Lines 90 and 91

Lines 90 and 91 display the other columns returned by the select. Line 90 displays the catalog number that is being processed and the **description** column from the stock table. Line 91 calls **prdesc()** to display the **cat_descr** column. See “Lines 159 to 177” on page 6-40 for a detailed discussion of **prdesc()**.

Lines 92 to 95

Line 92 deletes the file named in **cpfl[]**, the temporary file that contains the raster image for **cat_descr**. Line 93 calls **more()** to ask whether the user wants to enter more catalog numbers. If not, **more()** returns 0 and the program performs a **break** to terminate the main processing loop, close the database, and terminate the program. The closing brace on line 95 terminates the main processing loop, which began with the **while(1)** on line 49. If the user wants to enter another catalog number, control returns to that point.

Line 96 and 97

When a **break** statement terminates the main processing loop begun by the **while(1)** on line 49, control transfers to line 96, which closes the database. The closing brace on line 97 terminates the **main()** function and the program.

```

98
99  /*
100     Display the sunview raster file.  Note that this function works only
101     on SUN platforms.
102  */
103
104  display_picture()
105  {
106  #ifdef SUNVIEW
107      int child, childstat, w;
108      static char path[] = "/bin/screenload";
109      static char *slargs[] =
110          {
111              "-w",
112              "-x260",
113              "-y300",
114              "-X400",
115              "-Y350",
116              cpfl,
117              (char *) 0,
118          };
119
120      if((child = fork()) == 0)
121          {
122              execv(path, slargs);
123              fprintf(stderr, "Couldn't execute %s, errno %d", path, errno);
124              exit();
125          }
126      if((w = wait(&childstat)) != child && w != -1)
127          {
128              printf("Error or orphaned child %d", w);
129              exit(-1);
130          }
131  #endif /* SUNVIEW */
132  }

```

Continued on page 6-38

Lines 98 to 132

Line 104 begins the `display_picture` function to display `cat_picture`. It displays the `cat_picture` column *only* if the program is compiled with the `-DSUNVIEW` option. (See “Before Using `dispcat_pic`” page 6-22.) Line 106 checks to see whether `SUNVIEW` is defined. Lines 107 to 130 execute *only* if `SUNVIEW` is defined. If it is not defined, the function exits and returns to `main()`. Lines 107 to 118 define variables that are involved in storing and displaying `cat_picture`, as follows:

- | | |
|------------------|---|
| child | receives the process id of the child process that <code>display_picture</code> creates with <code>fork()</code> . |
| childstat | stores the status of the child process returned by <code>wait()</code> . |
| w | is the value returned by <code>wait()</code> , the process id of the terminated child process. |

path	stores the location and name of screenload , the program that displays the cat_picture image from cpfl .
slargs	stores the command-line arguments for screenload , the program that displays the cat_picture image, as follows:
-w	specifies the background color.
-x260, y300	specifies the location of the picture in pixels.
-X400, -Y350	specifies the size of the picture in pixels.
cpfl	is the name of the file containing the picture.
(char *) 0	is null terminator for the slargs array

The database server writes the image from the **cat_picture** column to the file named in **cpfl**. To display it, **dispcat_pic** calls the **fork()** system function to create a second process. The **dispcat_pic** program distinguishes the parent from the child process by checking the value that **fork()** returns. The **fork()** returns 0 to the child process and the child process id to the parent, so that only the child process executes lines 122 to 124. In the child process, line 122 executes the program named in **path**, the Sun **screenload** utility program. The **screenload** utility program overlays **dispcat_pic** in the child process and executes with the command-line arguments that are passed in **slargs[]**. Lines 123 and 124, which display an error message and exit, only execute if the system is unable to launch **screenload**.

On line 126, the parent process calls the **wait()** system function and waits for the child process to terminate. The **wait()** system function returns the process id of the process that terminates. Line 126 checks to see that this value, **w**, is the same as **child**, the value returned by **fork()**. It also checks to see that the value is not -1, which indicates an error or an interrupt occurred in the child process. If this occurs, line 128 displays a message to the user and line 129 exits the program.

```
133 getans(ans, len)
134 char *ans;
135 int len;
136 {
137     char buf[BUFFSZ];
138     int c, n = 0;
139
140     if(len >= BUFFSZ)
141     {
142         printf("\ngetans(): len > buffer");
143         return 0;
144     }
145     while(((c = getchar()) != '\n') && n < BUFFSZ)
146         buf[n++] = c;
147     buf[n] = '\0';
148     if(n > 1 && n >= len)
149     {
150         printf("Input exceeds maximum length");
151         return 0;
152     }
153     if(len <= 1)
154         *ans = buf[0];
155     else
156         strncpy(ans, buf, len);
157     return 1;
158 }
```

Continued on page 6-40

Lines 133 to 158

Lines 133 to 158 constitute the `getans()` function. The `getans()` function uses the `getchar()` standard library function to accept input from the user. Lines 134 and 135 define the incoming arguments for `getans()`, the address of the buffer where it copies the `ans` input, and the maximum number of characters that the `len` calling function expects. Line 137 defines a `buf[]` array that is the input buffer. The `int` on line 138, `c`, receives the character returned by `getchar()`. The second integer defined on line 138, `n`, is used to subscript the `buf[]` input buffer. Lines 140 to 144 check to see that the maximum length of the expected input, `len`, is less than or equal to the size of the BUFFSZ input buffer. If it is not, line 142 displays an error message and line 143 returns 0 to the calling function.

Line 145 calls `getchar()` to receive input from the user until a `\n` NEWLINE character is encountered or until the maximum input is received; that is, `n` is *not* less than BUFFSZ. Line 146 moves the `c` input character into the current position in `buf`. Line 147 places a terminating null at the end of the `buf[n]` input.

Lines 148 checks to see whether the number of characters received, `n`, is less than the number of characters expected, `len`. If not, line 150 displays a message to the user and line 151 returns 0 to the calling function to indicate that

an error occurred. Line 153 checks to see whether one or more characters were entered. If the expected number of characters, **len**, is less than or equal to 1, line 154 moves only a single character to the address given by the **ans** calling function. If only one character is expected, **getans()** does not append a terminating null to the input. If the expected input, **len**, is greater than 1, line 156 copies the user's input, **buf**, to the address supplied by the **ans** calling function. Line 157 returns 1 to the calling function to indicate successful completion.

```

159 prdesc()
160 {
161     long size;
162     char shdesc[81], *p;
163
164     size = cat_descr.loc_size;
165     printf("\nDescription for %ld:\n", cat_num);
166     p = cat_descr.loc_buffer;
167     while(size >= 80)
168     {
169         ldchar(p, 80, shdesc);
170         printf("\n%80s", shdesc);
171         size -= 80;
172         p += 80;
173     }
174     strncpy(shdesc, p, size);
175     shdesc[size] = '\0';
176     printf("%-s\n", shdesc);
177 }
178
179 more()
180 {
181     char ans;
182
183     while((ans = LCASE(ans)) != 'y' && ans != 'n')
184     {
185         printf("\nMore? (y/n) . . .");
186         if(!getans(&ans, 1))
187             continue;
188     }
189     return (ans == 'n') ? 0 : 1;
190 }
191

```

Continued on page 6-42

Lines 159 to 177

Lines 159 to 177 make up the `prdesc()` function that displays the `cat_descr` column of the `catalog` table. Line 161 defines `size`, a long integer that `prdesc()` initializes with the value in `cat_descr.loc_size`. Line 162 defines `shdesc[81]`, an array in which `prdesc()` temporarily moves 80 byte chunks of the `cat_descr` text for output. Line 162 also defines `*p`, a pointer that marks the current position in the buffer as it is being displayed.

`INFORMIX-OnLine` returns the size of the buffer that it allocates for a blob in `loc_size`. Line 164 moves `cat_descr.loc_size` to `size`. Line 165 displays the string "Description: " as a header for the `cat_descr` text. Line 166 sets the `p` pointer to the buffer address that `OnLine` returned in `cat_descr.loc_size`. Line 167 begins the loop that displays the `cat_descr` text to the user. The `while()` repeats the loop until `size` is less than 80. Line 169 begins the body of the loop. The `ldchar()` function copies 80 bytes from the current position in the buffer, addressed by `p`, to `shdesc[]`, removing any trailing blanks. Line 170 prints the content of `shdesc[]`. Line 171 subtracts 80 from `size` to account for

the portion of the buffer that was just printed. Line 172, the last in the loop, adds 80 to **p** to move it past the portion of the buffer that was just displayed. The process of displaying **cat_descr.loc_size** 80 bytes at a time continues until there are fewer than 80 characters left to be displayed (**size < 80**). Line 174 copies the remainder of the buffer into **shdesc[]** for the length of **size**. Line 175 appends a null to **shdesc[size]** to mark the end of the array and line 176 displays **shdesc[]**.

Lines 179 to 190

The **more()** function displays "More? (y/n) . . ." to ask whether the user wants to enter another catalog number. The **more()** function does not have any input arguments. Line 181 defines a one character field, **ans**, to receive the user's response. The condition expressed on line 183 causes the question to be redisplayed until the user answers **y(es)** or **n(o)**. The **LCASE** macro converts the user's answer to lowercase letters for the comparison. Line 185 displays the question and line 186 calls **getans()** to accept the user's input. Once the user answers **y(es)** or **n(o)**, control passes to line 189, which returns 1 for **y(es)** and 0 for **n(o)** to the calling function.

```
192 err_chk(name)
193 char *name;
194 {
195     if(sqlca.sqlcode < 0)
196     {
197         if((rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg)) < 0)
198             printf("\n\tError %d during %s:\n", sqlca.sqlcode, name);
199         else
200             printf("\n\tError %d during %s: %s\n", sqlca.sqlcode, name, errmsg);
201         exit(1);
202     }
203     return((sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1);
204 }
```

Lines 192 to 204

The `err_chk()` function examines the `sqlcode` field in the `sqlca` structure to determine the outcome of an SQL statement. The function takes the `name` input argument defined on line 193, which is a character pointer that gives the name of the executed SQL statement. If an error occurred, `name` is displayed to the user to identify the statement that was in error. Line 195 tests the value of `sqlca.sqlcode` to see whether it is less than 0. If so, `err_chk()` calls `rgetmsg()` to retrieve the message text for the error and store it in `errmsg[]`. If `rgetmsg()` is not successful, line 198 displays an error message that gives the values of `sqlca.sqlcode` and `name`. If `rgetmsg()` is successful, line 200 displays an error message that includes `sqlca.sqlcode`, `name` and the text in `errmsg[]`. In either case, if `sqlca.sqlcode` is less than 0, `err_chk()` exits the program rather than return to the calling function. If `sqlca.sqlcode` is greater than 0, line 203 checks to see whether its value is equal to `SQLNOTFOUND`. If so, it returns 0 to the calling function. Otherwise, it returns 1 to the calling function to indicate that the SQL statement was successful.

Error Handling

Chapter Overview	3
The Role of the <code>sqlca</code> Structure	3
General Error Handling	5
Error Status < 0	5
Error Status = 0	5
Error Status > 0 and < 100	5
Error Status = SQLNOTFOUND or 100	6
Error Status = 100 After a FETCH Statement	6
Error Status = 100 After Other Statements	6
Using the SQLCODE Variable	7
Checking for an Error Using In-Line Code	8
Automatically Checking for Errors with the WHENEVER Statement	9
Checking for Warnings	10
Errors After a PREPARE Statement	12
Errors After an EXECUTE Statement	12
RGETMSG	13
A Program That Uses Full Error Checking	15



Chapter Overview

Proper database management requires that all logical sequences of statements that modify the database continue successfully to completion. If, for example, you update a customer account to show a reduction of \$100 in the payable balance and the next step of updating the cash balance fails, your books become out of balance. It is important to check that every SQL statement executes as you anticipate.

This chapter describes how to use the **sqlca** structure to check for run-time errors in your **INFORMIX-ESQL/C** program. The **sqlca** structure is defined in **sqlca.h** and is shown in Figure 7-1. The **sqlca** structure is explained fully in Chapter 5 of *The Informix Guide to SQL: Reference*.

Also included in this chapter are the syntax and description of the **rgetmsg** routine, which you can use to find the error text of a given error number.

The Role of the *sqlca* Structure

After each SQL statement is executed, the database server returns information to the **sqlca** structure. The error status is one of the returned pieces of information. Information relevant to performance or the nature of the data handled also is returned. For some statements, warnings are returned instead of error information. You can take advantage of any of this information in your **ESQL/C** program.

Figure 7-1 contains the contents of the `sqlca.h` header file. The `sqlca.h` header file is automatically included in an `ESQL/C` program.

```

#ifndef SQLCA_INCL
#define SQLCA_INCL

struct sqlca_s
{
    long sqlcode;
    char sqlerrm[72]; /* error message parameters */
    char sqlerrp[8];
    long sqlerrd[6];
        /* 0 - estimated number of rows returned */
        /* 1 - serial value after insert or ISAM error code */
        /* 2 - number of rows processed */
        /* 3 - estimated cost */
        /* 4 - offset of the error into the SQL statement */
        /* 5 - rowid after insert */
    struct sqlcaw_s
    {
        char sqlwarn0; /* = W if any of sqlwarn[1-7] = W */
        char sqlwarn1; /* = W if any truncation occurred or
            database has transactions */
        char sqlwarn2; /* = W if a null value returned or
            ANSI database */
        char sqlwarn3; /* = W if no. in select list != no. in into
            list or OnLine backend */
        char sqlwarn4; /* = W if no where clause on prepared update,
            delete or incompatible float format */
        char sqlwarn5; /* = W if non-ANSI statement */
        char sqlwarn6; /* reserved */
        char sqlwarn7; /* reserved */
    } sqlwarn;
};

extern struct sqlca_s sqlca;

extern long SQLCODE;

#define SQLNOTFOUND 100

#endif /* SQLCA_INCL */

```

Figure 7-1 *The sqlca record structure*

General Error Handling

INFORMIX-ESQL/C returns a result code into the **sqlca** structure after executing every SQL statement except DESCRIBE. The returned code is put into the **sqlcode** field of the **sqlca** structure. You can test the contents of this field after each statement executes to verify that the statement executed correctly. There are four possible states:

Value	Definition
<code>sqlca.sqlcode < 0</code>	Failure
<code>sqlca.sqlcode = 0</code>	Success
<code>sqlca.sqlcode > 0 and < 100</code>	Depends on statement; see following discussion
<code>sqlca.sqlcode = SQLNOTFOUND</code>	No rows found; SQLNOTFOUND is equal to 100

Error Status < 0

If the statement failed to execute correctly, the database server sets the **sqlca.sqlcode** and **SQLCODE** to a negative value. It also can set other fields in the **sqlca** structure, as well as an **ISAM** return code. Checking and dealing with errors is explained in detail later in this chapter.

Error Status = 0

If execution succeeded, the database server sets **sqlca.sqlcode** to 0. Other information can be returned to the **sqlca** structure by the database server, if the statement succeeded. For information about the other fields in the **sqlca** structure, see the section “Checking for Warnings” on page 7-10 and the description of the **sqlca** structure in Chapter 5 of *The Informix Guide to SQL: Reference*.

Error Status > 0 and < 100

After a DESCRIBE statement, the database server sets **SQLCODE** to an integer value that represents the type of SQL statement that is described. The DESCRIBE statement operates on a statement id that is previously assigned by a PREPARE statement to a dynamic SQL statement. See “Constants and sql-stype.h” on page 9-9 for a list of possible **SQLCODE** values after a DESCRIBE statement.

Error Status = SQLNOTFOUND or 100

Five statements can cause the `SQLCODE` field to be set to the value 100. The five statements are shown in the following list. The last four statements cause positive values only if the database is ANSI-compliant.

- `FETCH`
- `DELETE...WHERE...`
- `INSERT...WHERE...`
- `SELECT INTO TEMP...WHERE...`
- `UPDATE...WHERE...`

Error Status = 100 After a `FETCH` Statement

The `FETCH` statement is a special case with respect to error handling. After a fetch, `sqlca.sqlcode` can contain the values 0, 100, or a negative value. The 0 and negative values indicate success and failure, as they do after the execution of other statements. The 100 value indicates that no additional rows are retrieved. For readability, the 100 value is defined as `SQLNOTFOUND` in `sqlca.h`. By checking for `sqlca.sqlcode = SQLNOTFOUND`, you can write code to process the results of queries only when rows are returned.

Error Status = 100 After Other Statements

In an ANSI-compliant database, if any of the following statements fail to access any rows, the database server sets `sqlca.sqlcode` equal to 100.

- `INSERT INTO tablename SELECT...WHERE`
- `SELECT INTO TEMP...WHERE...`
- `DELETE...WHERE`
- `UPDATE...WHERE`

In the following example, the INSERT statement inserts into the **hot_items** table any **stock** item that is ordered in a quantity greater than 10,000. If no items are ordered in that great a quantity, the SELECT part of the statement fails to insert any rows. The database server returns SQLNOTFOUND (100) in an ANSI-compliant database and 0 in a database that is not ANSI-compliant.

```
$insert into hot_items
      select distinct stock.stock_num,
                    stock.manu_code,description
      from items, stock
      where stock.stock_num = items.stock_num
            and stock.manu_code = items.manu_code
            and quantity > 10000;
```

The following example of an UPDATE statement fails to update any rows if there is no manufacturer with the **manu_code** SWK. The database server returns SQLNOTFOUND (100) in an ANSI-compliant database and 0 in a database that is not ANSI-compliant.

```
$update stock
      set unit_price = unit_price * 1.05
      where manu_code = "SWK";
```

Using the SQLCODE Variable

In the **sqlca.h** header file, the SQLCODE global variable is defined as a long integer. Whenever the database server returns a value to **sqlca.sqlcode**, the value is copied into SQLCODE. For readability and brevity, you can use SQLCODE in your **ESQL/C** program in place of **sqlca.sqlcode**.

You can use the SQLCODE error-checking variable in pure C modules linked to an **ESQL/C** program to return the same values returned in **sqlca.sqlcode** in **ESQL/C** modules. To use SQLCODE in a pure C module, declare SQLCODE as an external variable, as follows:

```
extern long SQLCODE;
```

The SQLCODE variable is allocated in the **libsql.a** library.

Checking for an Error Using In-Line Code

To check for an error, test the value of `sqlca.sqlcode` (or `SQLCODE`) after an SQL statement executes. For example, if you want to check that a `CREATE DATABASE` statement executed as expected, you can use the code shown in Figure 7-2.

```
$create database personnel with log;
if (SQLCODE < 0)
{
    printf("Error %d in creating database\n", sqlca.sqlcode);
    exit(1);
}
```

Figure 7-2 Condition to test for an error during an SQL statement

Alternatively, you can write a function that processes any error. Your program can call the error function each time that `SQLCODE` is returned as a negative value. The `do_error` function shown in Figure 7-3 retrieves the message associated with an error. It also checks for more information, if it is available, by checking whether an ISAM error also is returned. It prints any messages and then exits the program.

```
void do_error( st_name, errnum)
char *st_name;
int errnum;
{
    char errmsg[400];

    printf("Error %d occurred on %s.\n", errnum, st_name);
    rgetmsg(errnum, errmsg, sizeof(errmsg));
    printf("%s\n", errmsg);
    if (sqlerrd[1] != 0 )
    {
        printf("The ISAM code is %d\n", sqlerrd[1]);
        rgetmsg(sqlerrd[1], errmsg, sizeof(errmsg));
        printf("ISAM message: %s\n", errmsg);
    }
    exit(1);
}

$ create database personnel with log;
if ( SQLCODE < 0)
    do_error( "Create database" ,SQLCODE);
```

Figure 7-3 Example of an error-handling function

Check the status of `sqlca.sqlcode` after each SQL statement. Use the WHENEVER statement to reduce the amount of code that you must write to check for errors. Using the WHENEVER statement is explored in the following section.

Automatically Checking for Errors with the WHENEVER Statement

You can use the WHENEVER statement to trap for all errors and warnings that occur during the execution of SQL statements. Using the WHENEVER statement to check for errors replaces the conditional test of the SQLCODE value after each SQL statement.

Use the WHENEVER statement to check for errors, SQLNOTFOUND, or warnings. You can direct the program to take any of the following actions:

- Continue execution
- Stop execution
- Call a function
- Go to a labeled section of code

For details of the syntax and use of the WHENEVER statement, see Chapter 7 of *The Informix Guide to SQL: Reference*.

If you want to call the `do_error` function (shown in Figure 7-3) every time an error occurs in a program block, you can put the following statement in the early part of the program block, before any SQL statements:

```
$WHENEVER SQLERROR GOTO error_start;
```

As a result, your program contains SQL statements and the following code exists in the block of code that uses SQL statements. In this code example, `msg` is a global character variable that contains the type of statement executed. The program assigns the contents of `msg` before executing an SQL statement.

```
error_label:  
    do_err (msg, SQLCODE);
```

If you use the WHENEVER statement with the GOTO keyword and a label, you must provide that label and appropriate code in each function that contains SQL statements. Using the GOTO keyword in the WHENEVER statement is consistent with the ANSI standard.

If you do not want to use a GOTO construct, you can use the CALL keyword in the WHENEVER statement to call a function. (The CALL option is an Informix extension to the ANSI standard.) If you want to call the `do_error` function (shown in Figure 7-3) every time an error occurs in a program, you take the following two steps:

- Modify the `do_error` function so that it does not need any arguments. In this case, make the `st_name` a global variable.
- Put the following WHENEVER statement in the early part of your program, before any SQL statements

```
$WHENEVER SQLERROR CALL do_error;
```

Checking for Warnings

In addition to checking for errors after executing each SQL statement, you can check for warnings. Warnings that are issued by the database server when a statement is executed are stored in the `sqlcaw_s` structure within the `sqlca` structure. The form and possible contents of the warning structure are discussed in detail in Chapter 5 of *The Informix Guide to SQL: Reference*.

The database server generates the following warnings, grouped by the causing statement. The flag that is set for each situation is listed after the description of the warning.

DATABASE	Whether it uses transactions (sqlwarn1). Whether it operates as ANSI-compliant (sqlwarn2). Whether the database server is INFORMIX-OnLine (sqlwarn3).
START DATABASE	Whether it uses transactions (sqlwarn1). Whether the database is ANSI-compliant (sqlwarn2).
SELECT	Whenever a value from a database column is truncated to fit into a character host variable (sqlwarn1). Whenever an aggregate function encounters a null value (sqlwarn2). If the number of items in the select list of a SELECT clause is not the same as the number of host variables in the INTO clause (sqlwarn3). Whenever a float-to-decimal conversion is used (sqlwarn4).

INSERT	Whenever a float-to-decimal conversion is used (sqlwarn4).
UPDATE	Whenever a float-to-decimal conversion is used (sqlwarn4).
DELETE	Whenever a float-to-decimal conversion is used (sqlwarn4).
EXECUTE	Whenever a float-to-decimal conversion is used (sqlwarn4).
OPEN	Whenever a float-to-decimal conversion is used (sqlwarn4).
PREPARE	Whenever the UPDATE or DELETE statement is prepared without a WHERE clause (sqlwarn4).
All statements	Whenever an Informix extension to SQL is executed and the DBANSIWARN environment variable is set (sqlwarn5).
Subqueries	Whenever an aggregate function encounters a null value (sqlwarn2).

You can test for warnings using in-line code by checking whether the first warning field (**sqlwarn0**) is set to "W". You also use the WHENEVER statement with the SQLWARNING keyword to test whether any warnings are issued by the database server. Once you know a warning is issued by the database server, you can determine the exact nature of the warning by checking the values of the six used fields in **sqlcaw_s**.

For example, if you want to find out what kind of database was opened with a DATABASE statement, you can use the block of code shown in Figure 7-4.

```
msg = "DATABASE stmt"
$DATABASE stores5;
if (SQLCODE < 0 ) do_error(msg, SQLCODE);
else if (sqlca.sqlwarn0 == "W")
{
    if (sqlca.sqlwarn1 == "W" ) trans_db = 1;
    if (sqlca.sqlwarn2 == "W" ) ansi_db = 1;
    if (sqlca.sqlwarn3 == "W" ) online_db = 1;
}
```

Figure 7-4 In-line code to check if and what warnings are returned after a DATABASE statement

Note: After a DATABASE statement, **sqlwarn2** indicates that the database is ANSI-compliant. After a SELECT statement, **sqlwarn2** indicates that an aggregate function encounters a null value. The same warning flags are used differently for each SQL statement.

Errors After a PREPARE Statement

If a PREPARE statement fails with SQLCODE < 0, it is usually because of a syntax error in the prepared text. When a PREPARE statement fails, the offset into the text at which the error occurs is returned in the **sqlca.sqlerrd[4]** variable. Your program can use the value in **sqlca.sqlerrd[4]** to indicate where the syntax of the dynamically prepared text is incorrect. If you are using PREPARE with several statements, error status is returned on the first error in the text, even if there are several errors.

Errors After an EXECUTE Statement

If a prepared statement cannot be executed successfully, the database server returns SQLCODE < 0 after the EXECUTE statement. The SQLCODE variable holds the error that the database server returns from the failed statement. If the SQLCODE is equal to 0 after the completion of an EXECUTE statement, the prepared statement in the block succeeded; if the prepared block includes multiple statements, all of the statements succeeded.

RGETMSG

Purpose

The **rgetmsg** function converts an Informix error message number into the corresponding message text string. The error must be between -32766 and +32767.

Syntax

```
int rgetmsg(msgnum,msgstr,lenmsgstr)
    short msgnum;
    char *msgstr;
    short lenmsgstr;
```

lenmsgstr is the size of the message string.
msgnum is the error message number.
msgstr is the message string (output buffer).

Usage

The message number is typically one returned in **sqlca.sqlcode**. The **rgetmsg** function uses the system file for error message text (**/usr/informix/msg**).

Unlike **rgetmsg**, you can use **rgetlmsg** for any error message, both those within and outside the range handled by **rgetmsg**.

Return Codes

0	The conversion was successful.
-1232	Unknown message number.

Example

```
/*
 * rgetmsg.ec *

The following program demonstrates the usage of rgetmsg() function.
It displays an error message for accessing a non-existent database.
*/

#include sqlca;

char errmsg[400];

main()
{
    $database stories; /* Access a non-existent database */
    if (sqlca.sqlcode != 0)
    {
        rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
        printf("\nError %d (in database stories): %s\n", sqlca.sqlcode, errmsg);
    }
}
```

Example Output

```
Error -329 (in database stories): Database not found or no system permission.
```

A Program That Uses Full Error Checking

The program that follows is a modified version of **demo1.ec**, which was fully explained in Chapter 1 of this manual. The version listed and described here contains error checking on each of the SQL statements contained in the program; only the error-handling statements are described.

Error handling in this modified **demo1.ec** program uses the **WHENEVER** statement to call the **sql_error** C function. The **sql_error** function is defined in the program. If an error occurs in an SQL statement, the **sql_error** function is called. It displays the error number and the accompanying **ISAM** error, if there is one.

```

1  #include <stdio.h>
2  $include sqlca;
3
4  /* Uncomment the following line if the database has
5     transactions: */
6
7  /* $define TRANS; */
8
9  $define FNAME_LEN      15;
10 $define LNAME_LEN     15;
11
12 char state_name[20];
13
14 main()
15
16 {
17 $char fname[ FNAME_LEN + 1 ];
18 $char lname[ LNAME_LEN + 1 ];
19
20
21 printf( "DEMO1 Sample ESQL program running.\n\n");
22
23
24 $WHENEVER SQLERROR CALL sql_error;
25
26 strcpy (state_name, "DATABASE stmt");
27 $database stores;
28
29 strcpy (state_name, "DECLARE stmt");
30 $declare democursor cursor for
31     select fname, lname
32     into $fname, $lname
33     from customer
34     where lname > "C";
35
36 $ifdef TRANS;
37 strcpy (state_name, "BEGIN WORK stmt");
38 $begin work;
39 $endif;
40
41 strcpy (state_name, "OPEN stmt");
42 $open democursor;
43
44 strcpy (state_name, "FETCH stmt");
45 for (;;)
46     {
47     $fetch democursor;
48     if (sqlca.sqlcode != 0) break;
49     printf("%s %s\n",fname, lname);
50     }
51
52 strcpy (state_name, "CLOSE stmt");
53 $close democursor;
54
55 $ifdef TRANS;
56 strcpy (state_name, "COMMIT WORK stmt");
57 $commit work;
58 $endif;
59
60 printf("\nProgram Over.\n");
61 exit(1);
62 }                               /*End of main routine */

```

Continued on page 7-18

Line 12

The **char state_name[20]** statement declares a global variable that holds the name of the statement.

Line 24

The WHENEVER statement with the SQLERROR keyword indicates that after each SQL statement, the contents of **sqlca.sqlcode** are checked to determine whether an error occurred. If an error occurred, the CALL keyword indicates that the function **sql_error** is called.

Line 26

This line puts the string "DATABASE stmt" into the **state_name** variable so that the **sql_error** function can print out the type of statement that caused the error.

Lines 29, 37, 41, 44, 52, 56

These lines put a descriptive string into the **state_name** variable before each SQL statement, in case the SQL statement fails.

Line 61

The program exits with a return code of 1 to indicate that no errors occurred.

```
63
64 /* Function to handle errors */
65 sql_error ()
66 {
67     char errmsg[400];
68
69     printf("Error %d occurred on %s.\n", SQLCODE, state_name);
70     rgetmsg(SQLCODE, errmsg, sizeof(errmsg));
71     printf("%s\n", errmsg);
72     if (sqlca.sqlerrd[1] != 0 )
73     {
74         printf("The ISAM code is %d\n", sqlca.sqlerrd[1]);
75         rgetmsg(sqlca.sqlerrd[1], errmsg, sizeof(errmsg));
76         printf("ISAM message: %s\n", errmsg);
77     }
78
79     exit(0);
80 }
```

Line 65

Line 65 is the first line of the `sql_error` function. The `sql_error` function does not take any arguments. (Any function called from a `WHENEVER` statement cannot use arguments.) The `sql_error` function expects two global variables to contain information: `SQLCODE` and `state_name`. `SQLCODE` is declared in the `sqlca.h` file; `state_name` is declared above `main()`; the value of `state_name` is assigned before each SQL statement in the program.

Line 67

Line 67 declares a local character buffer to hold the error text associated with an SQL error or an **ISAM** error.

Lines 69 to 71

Lines 69 to 71 print the error number and error text. The error number is obtained from the `SQLCODE` global variable. The error text is obtained by calling the `INFORMIX-ESQL/C` routine `rgetmsg`.

Lines 72 to 77

After the error number and text is printed, the second element of the `sqlerrd` record is evaluated to determine whether it contains a nonzero number. If it contains a nonzero value, the value is the **ISAM** error number. In that case, the **ISAM** error number and the **ISAM** error text are printed. The `rgetmsg` routine is used to obtain the text of the **ISAM** error message.

Lines 79

The program exits with a return code of 0 to indicate that an error of some kind occurred.

Working with the Database Server

Chapter Overview 3

Database Server Control Functions 3

SQLBREAK 4

SQLDETACH 5

SQLEXIT 6

SQLSTART 7



Chapter Overview

This chapter contains information on the following functions that you can use to control the database server processes:

- **sqlbreak**
- **sqldetach**
- **sqlexit**
- **sqlstart**

Database Server Control Functions

The functions that you can use to control the database server processes are listed and fully described in the following sections.

Function Name	Description
sqlbreak	Sends the database server a request to stop processing.
sqldetach	Detaches a child process from a parent process.
sqlexit	Terminates a database server process.
sqlstart	Starts a database server process.

SQLBREAK

Purpose

The **sqlbreak** function is a run-time SQL function that sends the database server a request to interrupt processing.

Syntax

```
int sqlbreak()
```

Usage

When the database server receives the interrupt signal, it returns status and control to the application process, as if the SQL statement terminated with an error condition.

Return Codes

0	The call to sqlbreak was successful.
!=0	There is no database server process running when sqlbreak is called.

SQLDETACH

Overview

The **sqldetach** function is a run-time SQL database server control function that detaches a **child** process from a **parent** process so that the **child** process has its own database connection.

Syntax

```
void sqldetach()
```

Usage

The **sqldetach** function should be used when the **child** process shares the same code space as the **parent** process; that is, when there is no **exec()** after a **fork()** call.

This function does not work with the **vfork()** call.

SQLEXIT

Purpose

The **sqlexit** function is a run-time SQL function that terminates a database server process, thereby freeing resources. It can be used to reduce database overhead in programs that refer to a database only briefly and at long intervals or that access a database only during initialization.

Syntax

```
void sqlexit()
```

Usage

The **sqlexit** function should be called only when no databases are open. For example, before calling **sqlexit**, issue a CLOSE DATABASE statement. If **sqlexit** is called when a database that uses transactions is open, it rolls back any current transactions and closes the database.

SQLSTART

Purpose

The **sqlstart** function is a run-time SQL function that starts a database server process.

Syntax

```
int sqlstart()
```

Usage

The **sqlstart** function should be called only when no databases are open. If it is called when a database is open, **sqlstart** does nothing.

Executing the `$database` statement has the same effect as calling **sqlstart**, but also opens a database.

Return Codes

0	The call to sqlstart was successful.
<0	The call to sqlstart failed.

Example

```

/*
 * sqls.ec *

The following program reads the systables table for the selected database
and displays the name of each table found, followed by a list of the table
columns and their lengths. If the program is interrupted with CTRL-C, onintr()
intercepts the signal, interrupts the database server with sqlbreak(), and allows
the user to select another database.
*/

#include <stdio.h>
#include <ctype.h>
#include <decimal.h>
#include <setjmp.h>
#include <signal.h>
#include <errno.h>
#include sqltypes.h;

#define LCASE(c) (isalpha(c) ? (isupper(c) ? tolower(c) : c) : c)

jmp_buf sjbuf;

$char dbname[19];
char msgbuf[80];

main()
{
    int onintr();

    signal(SIGINT, onintr);
    setjmp(sjbuf);
    sqlstart();
    while(1)
    {
        printf("\n\n\tEnter the database name: ");
        if(getans(dbname, sizeof(dbname)) < 0)
        {
            printf("\n\tIllegal name\t");
            continue;
        }
        $database $dbname;
        if(err_chk("OPEN", dbname) < 0)
        {
            rgetmsg(sqlca.sqlcode, msgbuf, (short)sizeof(msgbuf));
            printf("\n\tstat: %d, %s\n", sqlca.sqlcode, msgbuf);
            continue;
        }
        break;
    }
    dsptbls();
    $close database;
    exit(1);
}

/*
The onintr() function catches SIGINT and terminates the database server.
It then allows the user to select another database.
*/

```

```

onintr()
{
    char ans;

    ans = ' ';
    printf("\n ***INTERRUPT *** \n");
    signal(SIGINT, onintr);
    sqlbreak();
    $close database;
    while(ans != 'y' && ans != 'n')
    {
        printf("\n\n\t*** Select another database? (y/n)");
        getans(&ans,1);
    }
    if(ans == 'y')
        longjmp(sjbuf,0);
    exit(1);
}

/*
The dsptbpls() function selects the tabname and tabid columns from the
systables table and displays them. It then calls dspcols() to display
the columns.
*/

dsptbpls()
{
    $char tabnm[19];
    $long tabid;

    $declare systabs cursor for select tabname, tabid from systables;
    $open systabs;
    if(err_chk("OPEN", "systabs") < 0)
        exit(1);
    while(1)
    {
        $fetch systabs into $tabnm, $tabid;
        if(!err_chk("fetch", "systabs"))
            break;
        printf("\n\n\tTable Name: %s", tabnm, tabid);
        dspcols(tabid);
    }
}

/*
The dspcols() function accesses the syscolumns table to display the name,
data type and length of each column in the table specified by tabid.
*/

dspcols(tabid)
$int tabid;
{
    $char colname[18];
    $short coltype, collength;
    char *rtypname();

    $declare syscols cursor for select colname, coltype, collength
        from syscolumns where tabid = $tabid;
    $open syscols;
    if(err_chk("OPEN", "syscols") < 0)
        exit(1);
    printf("\n\n\tColumn          Type          \t\tLength\n");
    while(1)
    {

```

```
    $fetch syscols into $colname, $coltype, $collength;
    if(!err_chk("fetch", "syscols"))
        return;
    printf("\n\t%s %-10s\t\t%d", colname, rtypname(coltype), collength);
    }
}

getans(ans, len)
char *ans;
int len;
{
    char buf[512], c;
    int n = 0;

    while((c = getchar()) != '\n')
        buf[n++] = LCASE(c);
    buf[n] = '\0';
    if(n > 1 && n >= len)
        return(-1);
    if(len <= 1)
        *ans = buf[0];
    else
        strncpy(ans, buf, len);
    return 1;
}

err_chk(str, ptr)
char *str, *ptr;
{
    if(sqlca.sqlcode < 0)
    {
        printf("\n\tError on %s: %s, %d", str, ptr, sqlca.sqlcode);
        return sqlca.sqlcode;
    }
    return (sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1;
}
```

Example Output

```

.
.
.
      dtype          char          1

Table Name: syssynonyms
      Column          Type          Length
      owner          char          8
      synname        char         18
      created        date          4
      tabid          integer        4

Table Name: syssyntable
      Column          Type          Length
      tabid          integer        4
      servername     char         18
      dbname         char         18
      owner          char          8
      tabname        char         18
      btabid         integer        4

Table Name: sysconstraints
^C      Column          Type          Length
      customer_num   integer        4
***INTERRUPT ***

      *** Select another database? (y/n)

```

Dynamic Management in INFORMIX- ESQL/C

Chapter Overview	3
Dynamic SQL Statements and Management Techniques	3
Types of Dynamic Management Situations	4
The System Descriptor Area	5
The <code>sqllda</code> Structure	6
Constants in <code>sqltypes.h</code>	7
Defined Constants for Use with Dynamic Statements	7
Constants and <code>sqlstype.h</code>	9
Non-SELECT Statements That Do Not Receive Values at Run Time	12
Using EXECUTE IMMEDIATE	12
SELECT Statements in Which Select-List Values Are Determined at Run Time	13
Using Descriptors	13
Example of a SELECT Statement Using Descriptors	15
Using an <code>sqllda</code> Structure	17
Example of a SELECT Statement Using the <code>sqllda</code> Structure	18

SELECT Statements That Receive WHERE-Clause Values at Run Time	21
Using Host Variables	21
Example of Using Host Variables in a Program	23
Using a System Descriptor Area	24
Example of Using a System Descriptor Area	26
Using an sqlda Structure	28
Non-SELECT Statements That Receive Values at Run Time	29
Using Host Variables	30
Using a System Descriptor Area	30
Example of Using SET DESCRIPTOR with a Locator Structure	31
Using an sqlda Structure	32

Chapter Overview

Dynamic management in **INFORMIX-ESQL/C** involves using SQL statements in which the contents are not known at the time the program is compiled. All or part of these dynamic SQL statements can be generated at the time the program is executed.

The SQL statements used for dynamic management are illustrated and described in Chapter 7 of *The Informix Guide to SQL: Reference*. The names of the statements that are used in dynamic management are as follows:

ALLOCATE DESCRIPTOR	FREE
DEALLOCATE DESCRIPTOR	GET DESCRIPTOR
DECLARE	OPEN
DESCRIBE	PREPARE
EXECUTE	PUT
EXECUTE IMMEDIATE	SET DESCRIPTOR
FETCH	

Dynamic SQL is also discussed in Chapter 6 of *The Informix Guide to SQL: Tutorial*.

This chapter discusses the general concepts of dynamic management and how to use it. It also includes two annotated example programs that use dynamic SQL.

Dynamic SQL Statements and Management Techniques

The basic process of using dynamic SQL statements is handled in four steps:

1. Your `ESQL/C` program assembles the text of an SQL statement in a character string variable.
2. Your program uses a `PREPARE` statement to ask the database server to examine the statement text and prepare it for execution.
3. Your program executes the prepared statement.
4. Your program uses the `FREE` statement to explicitly free resources associated with the prepared statement id.

You can use a `DESCRIBE` statement with any previously prepared statement to determine what type of statement was prepared. After a `DESCRIBE` statement, if the value in `SQLCODE` is 0, a `SELECT` statement without an `INTO TEMP` clause was prepared. If any other type of statement was prepared, the value of `SQLCODE` is some positive number. You can test the `SQLCODE` value against the values defined in `sqlstype.h` to determine what kind of statement was prepared. See the section “Constants and `sqlstype.h`” on page 9-9 for more information about using the defined constants.

Types of Dynamic Management Situations

In the following circumstances, the treatment of dynamically defined SQL statements is more complicated than that of static SQL statements, which were described in Chapter 1 of this manual. There are four situations for which you need to use dynamic SQL:

- When your program contains an SQL statement that is not a `SELECT` statement, uses no input parameters, and the statement is not known until run time. This type of statement is sometimes known as a “non-parameterized non-`SELECT` statement.” (See “Non-`SELECT` Statements That Do Not Receive Values at Run Time” on page 9-12.)
- When your program contains a `SELECT` statement and, at compile time, you do not know the number or the data types of the columns or expressions in the select list but you do know that the `SELECT` statement does not have a `WHERE` clause. This type of statement is sometimes known as a “non-parameterized `SELECT` statement.” (See “`SELECT` Statements in Which Select-List Values Are Determined at Run Time” on page 9-13.)
- When your program contains a `SELECT` statement that requires input at run time to provide information for the `WHERE` clause. You might or might not know the number or data type of the parameters in the `WHERE`

clause. This type of statement is sometimes known as a “parameterized SELECT statement.” (See “SELECT Statements That Receive WHERE-Clause Values at Run Time” on page 9-21.)

- When your program contains a statement that is not SELECT but you do not know the number or data type of the input parameters, such as an INSERT. This type of statement is sometimes known as a “parameterized non-SELECT statement.” (See “Non-SELECT Statements That Receive Values at Run Time” on page 9-30.)

If you are not using these kinds of statements, you can skip the rest of this chapter.

The last three types of statements require that you manage memory space for variables at run time. You can use either of two methods of dynamic memory management:

- Use system descriptor areas to hold the dynamic information. This is a language-independent method that uses the ALLOCATE DESCRIPTOR, GET DESCRIPTOR, or SET DESCRIPTOR statements in SQL. Use of the system descriptor area conforms to X/Open standards.
- Use the **sqlda** structure to hold the dynamic information. This method requires explicit use of the **sqlda** structure. Because it uses a C-language structure within SQL statements, this method is not language independent. Use of the **sqlda** structure does not conform to X/Open standards.

Both methods are outlined and described in this chapter. The system descriptor area and the **sqlda** structure are described in detail in Chapter 6 of *The Informix Guide to SQL: Reference*.

The System Descriptor Area

INFORMIX-ESQL/C uses a system descriptor area when your program contains the ALLOCATE DESCRIPTOR, GET DESCRIPTOR, and SET DESCRIPTOR statements. These statements allow you to handle the dynamic allocation of memory and to pass data between the application program and the database server.

A system descriptor area has a field for the count of values returned by a SELECT statement or inserted in an INSERT statement. It also has a set of fields for each value being input or returned. Figure 9-1 shows what a system descriptor area looks like for two values.

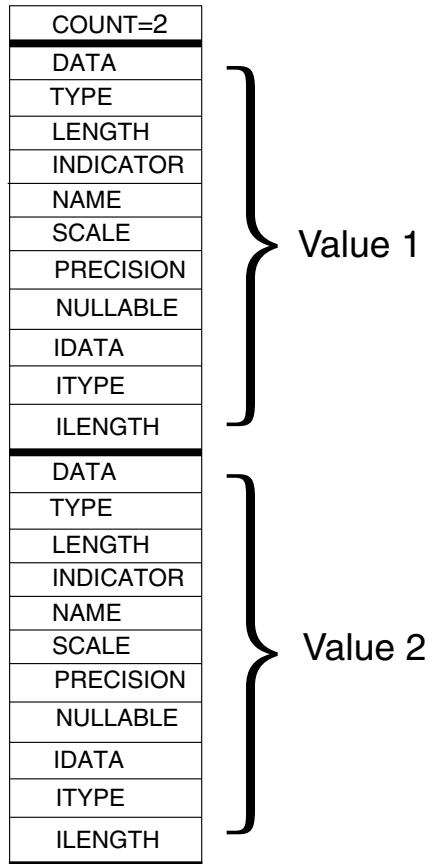


Figure 9-1 A system descriptor area for two values

The *sqlda* Structure

You use the **sqlda** structure when your program performs the dynamic memory allocation and you choose to allocate the memory for each of the dynamic variables in your own code. You do not use the **sqlda** structure if you are using the ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR statement set.

When all of its components are fully defined, the **sqllda** structure points to the beginning of a sequence of **sqlvar_struct** structures that contain the necessary information for each variable in the set. Figure 9-2 shows an **sqllda** structure.

```
struct sqlvar_struct {
    short sqltype;
    short sqlllen;
    char *sqldata;
    short *sqlind;
    char *sqlname;
    char *sqlformat;
    short sqlitype;
    short sqlilen;
    char *sqlidata;
};

struct sqllda {
    short sqld;
    struct sqlvar_struct *sqlvar;
};
```

Figure 9-2 Definition of the **sqllda** structure in **sqllda.h**

Constants in *sqltypes.h*

The **sqltypes.h** file contains two lists of defined integer constants for the data types used by Informix database servers.

Defined Constants for Use with Dynamic Statements

Within an INFORMIX-ESQL/C program using dynamically defined SQL statements, you can use the constants shown in Figure 9-3. The values listed are those returned by a DESCRIBE statement into the TYPE field of a system descriptor area or the **sqltype** field in an **sqllda** structure. Use them when analyzing the information returned by a DESCRIBE statement to either a system descriptor area or an **sqllda** structure. You also can use the literal values when setting the TYPE field with a SET DESCRIPTOR statement or setting the **sqltype** field in an **sqllda** structure.

Constant	SQL Data Type	Integer Value
SQLCHAR	CHAR	0
SQLSMINT	SMALLINT	1
SQLINT	INTEGER	2
SQLFLOAT	FLOAT	3
SQLSMFLOAT	SMALLFLOAT	4
SQLDECIMAL	DECIMAL	5
SQLSERIAL	SERIAL	6
SQLDATE	DATE	7
SQLMONEY	MONEY	8
SQLDTIME	DATETIME	10
SQLBYTES	BYTE	11
SQLTEXT	TEXT	12
SQLVCHAR	VARCHAR	13
SQLINTERVAL	INTERVAL	14

Figure 9-3 Values returned by a DESCRIBE statement and their defined constants

If you are compiling with the **-xopen** flag, you cannot use the integer values of the language-independent constants listed in Figure 9-3 in a SET DESCRIPTOR statement. You must use the values shown in Figure 9-3 for X/Open compatibility.

Constant	SQL Data Type	Integer Value
SQLCHAR	CHAR	1
SQLSMINT	SMALLINT	5
SQLINT	INTEGER	4
SQLFLOAT	FLOAT	6
SQLDECIMAL	DECIMAL	3

Figure 9-4 Values returned by a DESCRIBE statement in an X/Open environment

When you fill an **sqllda** structure to indicate where and what kind of host variables are used as input parameters to a dynamically defined SQL statement, you can use the values shown in Figure 9-5 (for which constants are defined in **sqltypes.h**) to identify the C data type.

Constant	C Data Type	Integer Value
CCHARTYPE	char	100
CSHORTTYPE	short int	101
CINTTYPE	int	102
CLONGTYPE	long	103
CFLOATTYPE	float	104
CDOUBLETYPE	double	105
CDECIMALTYPE	dec_t	107
CFIXCHARTYPE	fixchar	108
CSTRINGTYPE	string	109
CDATETYPE	long	110
CMONEYTYPE	dec_t	111
CDTIMETYPE	dtime_t	112
CLOCATORTYPE	loc_t	113
CVCHARTYPE	varchar	114
CINVTTYPE	intrvl_t	115
CFILETYPE	char	116

Figure 9-5 Values that can be used to set the sqltype field in an sqlda structure

Constants and *sqlstype.h*

After you use the DESCRIBE statement, the database server sets SQLCODE to a positive integer value that indicates the type of statement that was described. That is, SQLCODE indicates whether the statement was an INSERT, SELECT, CREATE TABLE, or any other statement. You can determine which kind of statement was described by checking the value of SQLCODE against a predefined set of values.

The **sqlstype.h** file contains the list of predefined integer constants for types of SQL statements. A portion of the file is shown in Figure 9-6.

```

.
.
.
#define SQ_UPDATE      4
#define SQ_DELETE      5
#define SQ_INSERT      6
#define SQ_UPDCURR     7
#define SQ_DELCURR     8
#define SQ_LDINSERT    9
#define SQ_LOCK        10
.
.
.

```

Figure 9-6 A portion of the contents of the sqlstype.h file

The complete list of predefined integer constants for types of SQL statements, and the full name of the statement, is shown in Figure 9-7 and Figure 9-8.

Statement	Defined Constant	Value
SELECT		0
DATABASE	SQ_DATABASE	1
SELECT INTO	SQ_SELINTO	3
UPDATE...WHERE	SQ_UPDATE	4
DELETE...WHERE	SQ_DELETE	5
INSERT	SQ_INSERT	6
UPDATE WHERE CURRENT OF	SQ_UPDCURR	7
DELETE WHERE CURRENT OF	SQ_DELCURR	8
LOAD FROM INSERT INTO	SQ_LDINSERT	9
LOCK	SQ_LOCK	10
UNLOCK	SQ_UNLOCK	11
CREATE DATABASE	SQ_CREADB	12
DROP DATABASE	SQ_DROPDB	13
CREATE TABLE	SQ_CRETAB	14
DROP TABLE	SQ_DRPTAB	15
CREATE INDEX	SQ_CREIDX	16
DROP INDEX	SQ_DRPIDX	17
GRANT	SQ_GRANT	18
REVOKE	SQ_REVOKE	19
RENAME TABLE	SQ_RENTAB	20
RENAME COLUMN	SQ_RENCOL	21
CREATE AUDIT	SQ_CREAUD	22

Figure 9-7 A portion of the contents of the sqlstype.h file and the corresponding statement type (1 of 2)

Statement	Defined Constant	Value
DROP AUDIT	SQ_DRPAUD	25
RECOVER TABLE	SQ_RECTAB	26
CHECK TABLE	SQ_CHKTAB	27
REPAIR TABLE	SQ_REPTAB	28
ALTER	SQ_ALTER	29
START DATABASE	SQ_STATS	30
CLOSE DATABASE	SQ_CLSDB	31
DELETE without WHERE	SQ_DELALL	32
UPDATE without WHERE	SQ_UPDALL	33
BEGIN WORK	SQ_BEGWORK	34
COMMIT	SQ_COMMIT	35
ROLLBACK	SQ_ROLLBACK	36
START DATABASE	SQ_STARTDB	38
ROLL FORWARD	SQ_RFORWARD	39
CREATE VIEW	SQ_CREVIEW	40
DROP VIEW	SQ_DROPVIEW	41
CREATE SYNONYM	SQ_CREASYN	43
DROP SYNONYM	SQ_DROPSYN	44
CREATE TEMP TABLE	SQ_CTEMP	45
SET LOCK MODE TO WAIT	SQ_WAITFOR	46
ALTER INDEX	SQ_ALTIDX	47
SET ISOLATION	SQ_ISOLATE	48
SET LOG	SQ_SETLOG	49
SET EXPLAIN	SQ_EXPLAIN	50
CREATE SCHEMA	SQ_SCHEMA	51
SET OPTIMIZATION	SQ_OPTIM	52
CREATE PROCEDURE	SQ_CREPROC	53
DROP PROCEDURE	SQ_DRPPROC	54
SET CONSTRAINTS	SQ_CONSTRMODE	55
EXECUTE PROCEDURE	SQ_EXECPROC	56
SET DEBUG FILE TO	SQ_DBGFILE	57
CREATE OPTICAL CLUSTER	SQ_CREOPCL	58
ALTER OPTICAL CLUSTER	SQ_ALTOPCL	59
DROP OPTICAL CLUSTER	SQ_DRPOPCL	60
OPTICAL RESERVE	SQ_OPRESERVE	61
OPTICAL RELEASE	SQ_OPRELEASE	62
SET TIMEOUT	SQ_OPTIMEOUT	63
UPDATE STATS...for procedure	SQ_PROCSTATS	64

Figure 9-8 A portion of the contents of the sqlstype.h file and the corresponding statement type (2 of 2)

Non-SELECT Statements That Do Not Receive Values at Run Time

If you want to assemble a statement at run time, in many cases it is a simple process. As long as it is not a SELECT statement, and as long as you know the basic structure of the statement and all of the components when you write your program, you can simply prepare and execute the statement.

For example, you can write a general-purpose deletion program that works on any table. Your program would take the following steps:

1. Prompt the user for the name of the table and the text of the WHERE clause and put the information into host variables such as **\$tablename** and **\$search_condition**.
2. Create a text string by concatenating four components: DELETE FROM, **\$tablename**, WHERE, and **\$search_condition**. For this example, call the string **stmt_buf**.
3. Prepare the entire statement. The following PREPARE statement operates on the string in **stmt_buf** and creates a statement id called **d_id**:

```
$PREPARE d_id FROM $stmt_buf;
```

4. Execute the statement. For this example, it uses the following EXECUTE statement:

```
$EXECUTE d_id;
```

Using EXECUTE IMMEDIATE

Instead of preparing the statement and then executing it, you can prepare and execute the statement in the same step using the EXECUTE IMMEDIATE statement.

For example, for the DELETE statement used in the example in the previous section, you can replace the PREPARE-EXECUTE statement sequence with the following statement:

```
$EXECUTE IMMEDIATE $stmt_buf;
```

SELECT Statements in Which Select-List Values Are Determined at Run Time

In the SELECT statement described and illustrated in the example program in Chapter 1 of this manual, the values returned from the query are placed into host variables that are listed in an INTO clause. When your program creates a SELECT statement at run time, you cannot use an INTO clause in the SELECT statement because you do not know at compile time what host variables are needed. Instead, you must use a system descriptor area or an **sqllda** structure to hold the selected values.

Using Descriptors

Follow these steps to program the code using the system descriptor area for a SELECT statement in which select-list values are determined at run time:

1. Prepare the SELECT statement (using the PREPARE statement) and give it a statement identifier. For example, the statement id can be **qid**.

```
$PREPARE qid FROM "SELECT * FROM customer";
```

2. Declare a cursor for the prepared statement identifier (for example, name the cursor **q_cursor**). All dynamically defined SELECT statements must have a declared cursor. For example, the following statement declares the cursor **q_cursor** for **qid**:

```
$DECLARE q_cursor CURSOR FOR qid;
```

3. Allocate a descriptor using the ALLOCATE DESCRIPTOR statement. Provide a name for the descriptor area. Also indicate the maximum number of items that can be in the select list of the query. If you do not provide a maximum, space is allocated for 100 returned items. The following statement allocates a descriptor named **demodesc**, which has room for up to a four-item select list:

```
$ALLOCATE DESCRIPTOR 'demodesc' WITH MAX 4;
```

4. Open the cursor.

```
$OPEN q_cursor
```

Note: If the SELECT statement has a WHERE clause (that is, it is receives WHERE-clause values at run time), your program must handle the WHERE clause as well. Information on handling SELECT statements that receive WHERE-clause values at run time is contained in a later section of this chapter.

5. Determine the contents of the select list of the query. To do this, use the DESCRIBE statement with the descriptor that you allocated for the returned values. For example, the following statement describes the prepared query **qid** into the **demodesc** descriptor:

```
$DESCRIBE qid USING SQL DESCRIPTOR 'demodesc';
```

6. Use the GET DESCRIPTOR statement to determine the count of values in the select list by looking at the COUNT field of the descriptor. For example, the following statement puts the count of the values in the select list into the **desc_count** host variable:

```
$GET DESCRIPTOR 'demodesc' $desc_count = COUNT;
```

7. Determine the type, length, name, and other information about each of the values described into the descriptor; your program needs this information for formatting or processing. For example, to determine the type of the third value in a select list, use the following statement:

```
$GET DESCRIPTOR 'demodesc' VALUE 3 $type_int = TYPE;
```

8. Fetch each row of values returned with the SELECT statement in a loop until no more rows are found (SQLNOTFOUND). After each FETCH statement, use the GET DESCRIPTOR statement on each value in the select list to load the contents of the DATA field into an appropriate host variable

for your program to use. For example, the following statement copies the data for the first value into the **result** host variable:

```

while (sqlca.sqlcode != 100)
  { $fetch q_cursor using sql descriptor 'demodesc';
  for (i = 1; i <= $desc_count; i++)
    { $GET DESCRIPTOR "demodesc" VALUE $i $result = DATA;
      .
      .
      .
    }
  .
  .
  .
}

```

9. After all of the rows are fetched, close the cursor.

Example of a SELECT Statement Using Descriptors

This example is a modified version of the **demo4.ec** program. It uses the GET DESCRIPTOR and SET DESCRIPTOR statements to run a SELECT statement in which select-list values are determined at run time.

```

/*
 * This demo program is a version of the demo demo4.ec that
 * uses the X/OPEN GET/SET DESCRIPTOR
 */
#include <stdio.h>
#include sqlca;
#include sqllda;

/* Uncomment the following line if the database has
   transactions: */

/* #define TRANS; */

#define NAME_LEN 15;

main()
{
  $int i;
  $int desc_count;
  $char demoquery[80];
  $char queryvalue[2];
  $char result[ NAME_LEN + 1 ];

  /* These next four lines have hard-wired
   * the query .
   * This information could have been entered
   * from the terminal and placed into the string
   * demoquery.
   */

```

```
printf(demoquery, "%s ",
       "select fname, lname from customer");

printf("Modified DEMO4 Sample ESQL program running.\n\n");

$database stores5;
$prepare qid from $demoquery;
$declare democursor cursor for qid;
$allocate descriptor 'demodesc' with max 4;

$ifdef TRANS;
$begin work;
$endif;

$open democursor;
$describe qid using sql descriptor 'demodesc';
$get descriptor 'demodesc' $desc_count = count;

printf("There are %d returned columns:\n", desc_count);
/* Print out what DESCRIBE returns */
for (i = 1; i <= desc_count; i++)
    prsqlda(i);

printf("\n\n");
for (;;)
{
    $fetch democursor using sql descriptor 'demodesc';
    if (sqlca.sqlcode != 0) break;
    for (i = 1; i <= desc_count; i++)
    {
        $ get descriptor 'demodesc' value $i $result = data;
        printf("%s ", result);
    }
    printf("\n");
}

$close democursor;

$ifdef TRANS;
$commit work;
$endif;

printf("\nProgram Over.\n");
}

prsqlda(index)
    $ parameter int index;
{
    $ int type;
    $ int len;
    $ char name[40];

    $ get descriptor 'demodesc' value $index
        $type = type,
        $len = len,
        $name = name;
    printf("    Column %d: type = %d, len = %d, name = %s\n",
           index, type, len, name);
}
```

Using an `sqllda` Structure

Follow these steps to program the code using an `sqllda` structure for a SELECT statement in which select-list values are determined at run time:

1. Declare a pointer to an `sqllda` structure. For example, name it `udesc`.
2. Prepare the SELECT statement (with the PREPARE statement) and give it a statement identifier. For example, name the statement identifier `u_query`.
3. Use the DESCRIBE statement to analyze the values returned by the SELECT statement (the number and type of columns and values in the select list). The analysis is stored in the `sqllda` structure. For example, use the following statement:

```
$DESCRIBE u_query INTO udesc;
```

This DESCRIBE statement causes `udesc->sqlvar` to point to a sequence of partially filled `sqlvar_struct` structures. (`udesc->sqlld` gives the number of `sqlvar_struct` structures.) The components of each `sqlvar_struct` structure that are filled by the DESCRIBE statement for each item of the select list are `sqltype`, `sqlllen` (for CHAR type data or a qualifier for DATETIME or INTERVAL), and `sqlname`.

4. After looking at each of the `sqltype` and `sqlllen` fields filled by the DESCRIBE statement, your program must allocate memory for the `sqldata` field appropriately. (This involves using a function like `malloc()` and assigning proper word boundaries, depending on the data type and, in the case of CHAR variables, the length of the variable.)
5. Declare a cursor for the prepared statement identifier (for example, name the cursor `u_cursor`). All dynamically defined SELECT statements must have a declared cursor. For example, the following statement declares the cursor `u_cursor` for `u_query`:

```
$DECLARE u_cursor cursor FOR u_query;
```

6. Open the cursor with the OPEN statement. For example, the following statement opens the `u_cursor` cursor:

```
$OPEN u_cursor;
```

Note: If the SELECT statement has a WHERE clause (that is, it receives WHERE-clause values at run time), your program must handle the WHERE clause as well. Information on handling SELECT statements that receive WHERE-clause values at run time is contained in a later section of this chapter.

7. Retrieve the first set of values from the SELECT statement by issuing a FETCH statement. For example, the following statement gets the first set of values from the SELECT statement prepared in step 2.

```
$FETCH u_cursor USING DESCRIPTOR udesc;
```

This statement assigns values to the variables pointed to by the various `sqllda` pointers.

8. Repeat the FETCH statement until there are no more rows.
9. Close the cursor with the CLOSE statement.

Example of a SELECT Statement Using the `sqllda` Structure

To illustrate this process, consider the program fragment shown in Figure 9-10, in which all error checking is suppressed. This example assumes that a SELECT statement is assembled at run time and stored in the `q_string` string. For the sake of this example, let `q_string` contain the statement shown in Figure 9-9.

```
select order_num, order_date from orders
```

Figure 9-9 Query in `q_string`

```

#include sqlca;
#include sqlda;

        .
        .
        .
/* declare pointer to structure that
   apporitions the data from each row */
struct sqlda *q_desc;

/* declare host variable to hold
   statement string */
$          char  q_string[128];

/* make stores5 the current database */
$          database stores5;
        .
        .
        .
/* At this point the SELECT statement
   is assigned to q_string */

/* prepare the SELECT statement */
$          prepare q_id from $q_string;

/* identify the select-list */
$          describe q_id into q_desc;

        /* this section of the code must
        allocate variables to receive
        the data from the rows and
        set the pointers in q_desc.
        See the following discussion */

/* associate q_cursor with SELECT
   statement */
$          declare q_cursor cursor for q_id;

/* open q_cursor; create active set */
$          open q_cursor;

/* loop through rows in active set */
        for(;;)
        {

/* fetch next row from the active set */
$          fetch q_cursor
            using descriptor q_desc;

/* process data if fetch returned a row */
        if (sqlca.sqlcode == 0)
        {
            /* process data */
            .
            .
            .
        }
        else
        {
            /* out of data */
            break;
        }
        }

```

Figure 9-10 Sequence of statements to use an sqlda structure

After including the header files and declaring the relevant variables, the program selects the `stores5` database as the current database. Once the query is stored in the `q_string` string, it is prepared and associated with the `q_id` identifier. Since the statement is a query, the program then declares a `q_cursor` cursor so that the rows that are returned can be examined one at a time. The `OPEN` statement that follows opens the cursor and defines an active set of rows for the `SELECT` statement.

The complex part of this program fragment follows the `DESCRIBE` statement and is not shown in detail. The `sqlda` component `q_desc->sqlid` now has the value 2, since two columns were selected from the `orders` table. The component `q_desc->sqlvar[0]` is the `sqlvar_struct` structure that contains information on the `order_num` column of the `orders` table. The component `q_desc->sqlvar[1]` is the `sqlvar_struct` structure that contains information on the `order_date` column of the `orders` table. The `q_desc->sqlvar[0].sqltype` component, for example, gives the SQL data type of the first column (`order_num`) requested.

Dynamically Allocating Memory

You must allocate memory storage for each of the variables returned by the query. You must set the `sqldata` pointer for each value returned to the location that receives the value. This process can require aligning data types with proper word boundaries. You can use the `rtypalign` function described in Chapter 5 of this manual for this purpose.

When you dynamically allocate `dtime_t` structures to receive `DATETIME` values, you can set their qualifiers from the associated `sqllen` fields. Alternatively, you can compose a different qualifier (using the values defined in `datetime.h`) and the database value is extended to match it.

When you dynamically allocate `intrvl_t` structures to receive `INTERVAL` values, you should always initialize its qualifier from `sqllen`. Alternatively, you can set its qualifier to zero, and the qualifier from the database column is set during the fetch.

In the preceding example, the `DESCRIBE` statement allocated memory for the necessary number of `sqlvar_struct` structures. It filled in `sqltype` and `sqlname`, and set to null the `sqldata` and `sqlind` pointers. You must set `sqldata` and `sqlind` (if desired) to point to the appropriate memory. If you do not use the `DESCRIBE` statement to create the `sqlvar_struct` structures, you must remember to set the indicator variable pointers to null.

The `unload.ec` example program supplied with `ESQL/C` illustrates the use of the `sqlda` structure with a `SELECT` statement in which select-list values are determined at run time.

SELECT Statements That Receive WHERE-Clause Values at Run Time

Since DESCRIBE statements examine only the select list—that is, the list of column names or expressions in the SELECT clause—they do not tell you about parameters in the WHERE clauses.

You must know the number of parameters in the SELECT statement and their data types. Unless you are writing a general-purpose, interactive interpreter, you usually have this information. If you do not have it, you must write code that determines not only how many question marks (?) appear in the dynamic query, but also the data types to which they belong.

When you know the number of parameters and their data types at compile time, you can declare appropriate host variables to receive the parameter values and run the query using those values.

When you determine the number of parameters and their data types at compile time, you have two options:

- Use a system descriptor area to pass data to the query.
- Allocate memory for an **sqllda** structure to pass data to the query.

The rest of this section outlines how to handle SELECT statements that receive WHERE-clause values at run time.

Using Host Variables

Follow these steps to use host variables with SELECT statements that receive WHERE-clause values at run time:

1. Declare a host variable for each parameter in the WHERE clause of the SELECT statement.
2. Prepare the SELECT statement. It must contain a ? for each missing value in the WHERE clause.
3. Associate a cursor with the prepared SELECT statement by using the DECLARE statement.
4. Assign a value to the host variable for each parameter. (These values probably would be obtained interactively in an application.)
5. Use the OPEN statement with the USING clause to associate the host variables (and their contents) with the question marks in the prepared SELECT statement.

6. Use the FETCH statement to get the first set of values returned by the prepared SELECT statement. Repeat the FETCH statement until no more rows are returned.
7. Close the cursor using the CLOSE statement.

If the host variables corresponding to the parameters in the SELECT statement are **hvar1**, **hvar2**, and **hvar3**, you must execute the OPEN statement, as shown in the program fragment in Figure 9-11.

```
$include sqlca
...
/* declare parameter variables */
$  longhvar1;
$  longhvar2;
$  longhvar3;

/* make stores5 the current database */
$  database stores5;

/* prepare the select statement */
$  prepare q_id from
      "select order_num, customer_num
      from orders
      where order_date = ?
      or paid_date = ?
      or ship_date = ?";

/* associate q_cursor with SELECT statement */
$  declare q_cursor cursor for q_id;

/* this section of the program would
   assign values to hvar1, hvar2, and hvar3 /
*
...

/* open q_cursor to create the active set */
$  open q_cursor using $hvar1, $hvar2, $hvar3;

...
```

Figure 9-11 Program fragment that uses host variables with a cursor

Example of Using Host Variables in a Program

The **demo2.ec** example program distributed with **INFORMIX-ESQL/C** uses a host variable to hold the value of the parameter for a **SELECT** statement. The **demo2.ec** program is shown in Figure 9-12.

```
#include <stdio.h>
#include sqlca;

/* Uncomment the following line if the database has
   transactions: */

/* #define TRANS; */

#define FNAME_LEN      15;
#define LNAME_LEN      15;

main()
{
  $char demoquery[80];
  $char queryvalue[2];
  $char fname[ FNAME_LEN + 1 ];
  $char lname[ LNAME_LEN + 1 ];

  /* These next four lines have hard-wired both
   * the query and the value for the parameter.
   * This information could have been entered
   * from the terminal and placed into the strings
   * demoquery and queryvalue, respectively.
   */
  sprintf(demoquery, "%s %s",
          "select fname, lname from customer",
          "where lname > ? ");
  sprintf(queryvalue, "C");

  printf("DEMO2 Sample ESQL program running.\n\n");

  $database stores;
  $prepare qid from $demoquery;
  $declare democursor cursor for qid;

  $ifdef TRANS;
  $begin work;
  $endif;

  $open democursor using $queryvalue;
```

```

if (sqlca.sqlcode)
    printf("%s %d\n",
        "sqlca.code, after the cursor open, is",
        sqlca.sqlcode);

for (;;)
    {
    $fetch democursor into $fname, $lname;
    if (sqlca.sqlcode != 0) break;
    printf("%s %s\n", fname, lname);
    }
if (sqlca.sqlcode != SQLNOTFOUND)
    printf("%s %d\n",
        "sqlca.code, after fetch, is",
        sqlca.sqlcode);

$close democursor;

#ifdef TRANS;
$commit work;
#endif;

printf("\nProgram Over.\n");
}

```

Figure 9-12 The text of the demo2.ec example program

Using a System Descriptor Area

Your code must take the following steps to use a system descriptor area with a SELECT statement that receives WHERE-clause values at run time. The first four steps are common to SELECT statements that receive WHERE-clause values at run time and SELECT statements in which select-list values are determined at run time.

1. Declare host variables to interactively hold the data obtained from the user.
2. Prepare the SELECT statement (using the PREPARE statement) and give it a statement identifier. The SELECT statement must contain a ? for each missing value in the WHERE clause. For example, the statement id can be **qid**.

```

$PREPARE qid FROM
    "select * from customer where lname > ?";

```

3. Declare a cursor for the prepared statement identifier (for example, name the cursor **q_cursor**). All dynamically defined SELECT statements must

have a declared cursor. For example, the following statement declares the cursor **q_cursor** for **qid**:

```
$DECLARE q_cursor CURSOR FOR qid;
```

4. Allocate a descriptor using the `ALLOCATE DESCRIPTOR` statement. Provide a name for the descriptor area. Also indicate the maximum number of items that can be in the select list of the query. If you do not provide a maximum, space is allocated for 100 returned items. The following statement allocates a descriptor named **demodesc**, which has room for up to a four-item select list:

```
$ALLOCATE DESCRIPTOR 'demodesc' WITH MAX 4;
```

5. Your C code must analyze the `WHERE` clause of the `SELECT` statement to determine how many and what type of parameters are in the `WHERE` clause.
6. Once you determine the number of question marks in the query, you must use an `ALLOCATE DESCRIPTOR` statement to allocate a descriptor large enough to handle the filter variables. You can use the descriptor allocated in step 4, if it is large enough.
7. Use the `SET DESCRIPTOR` statement to set the `COUNT` field in the descriptor to the number of parameters (question marks) in the `WHERE` clause. For example, if three question marks are in the `WHERE` clause, your program must set `COUNT` to 3.

```
$SET DESCRIPTOR 'demodesc' COUNT = 3;
```

8. Issue a `SET DESCRIPTOR` statement for each of the question marks (filter values) in the `SELECT` statement. The `SET DESCRIPTOR` statement must set the `TYPE` and `DATA` fields of the descriptor area. If you are setting the descriptor for a `CHAR` or `VARCHAR` item, you also must provide a value for the `LENGTH` field. The other fields are optional. The following state-

ment sets the first value in the descriptor area for a character value with a value assigned from the **hostchar** host variable:

```
$SET DESCRIPTOR 'demodesc' VALUE 1
    TYPE = 0,
    LENGTH = 15,
    DATA = $hostchar;
```

*Note: If you are using X/Open code (and compiling with the **-xopen** flag), you must use the literal integer values listed in Figure 9-3 on page 9-8 in the TYPE field of the SET DESCRIPTOR statement. For the preceding example, TYPE would be 1 rather than 0.*

9. Once you set all the necessary information for each VALUE, open a cursor using the descriptor. For the previous SET DESCRIPTOR statement, the OPEN statement is as follows:

```
$OPEN q_cursor USING SQL DESCRIPTOR 'demodesc';
```

10. Fetch each row of values returned with the SELECT statement in a loop until no more rows are found (SQLNOTFOUND). For example, the following FETCH statement puts the results of the query into the **col1**, **col2**, and **col3** host variables. (You can put the results into host variables if you know the contents of the select list at compile time. If you do not know the list of columns in the select list at compile time, you must apply the techniques described in the section "SELECT Statements in Which Select-List Values Are Determined at Run Time" on page 9-13 to determine the contents of the select list.)

```
while (sqlca.sqlcode != 100)
{
    $FETCH q_cursor INTO $col1, $col2, $col3;
    /* Do something with results */
}
```

11. After all of the rows are fetched, use the CLOSE statement to close the cursor.

Example of Using a System Descriptor Area

The program in Figure 9-13 is a modified version of the **demo4.ec** example program. It illustrates the use of a system descriptor area to handle both input parameters and the values in the select list.

```

/*
 * This program is a modified version of the demo4.ec that
 * uses the X/OPEN GET/SET DESCRIPTOR. This program shows how to use
 * a system descriptor for the parameters of a SELECT statement.
 */
#include <stdio.h>
#include sqlca;
#include sqllda;

/* Uncomment the following line if the database has
   transactions: */

/* #define TRANS; */

#define NAME_LEN 15;

main()
{
    $int i;
    $int desc_count;
    $char demoquery[80];
    $char queryvalue[2];
    $char result[ NAME_LEN + 1 ];

    /* These next three lines have hard-wired both
     * the query and the value for the parameter.
     * This information could have been entered
     * from the terminal and placed into the strings
     * demoquery and queryvalue, respectively.
     */
    sprintf(demoquery, "%s %s",
            "select fname, lname from customer",
            "where lname > ? ");

    /* This section of the program must evaluate $demoquery
     * to count how many question marks are in the where clause
     * and what kind of data type is expected for each question mark.
     * For this example, there is one paramter of type char(15).
     * It would then obtain the value for $queryvalue. The value of
     * queryvalue is hard-wired in the next line.
     */

    sprintf(queryvalue, "C");

    printf("Modified DEMO4 Sample ESQL program running.\n\n");

    $database stores;
    $prepare qid from $demoquery;
    $declare democursor cursor for qid;
    $allocate descriptor 'demodesc' with max 4;

    $ifdef TRANS;
    $begin work;
    $endif;

```

```

/*number of parameters to be held in descriptor is 1 */
    $ SET DESCRIPTOR "demodesc" COUNT = 1;
/* Put the value of the parameter into the descriptor */
$ SET DESCRIPTOR "demodesc" VALUE 1
    TYPE = 0, LENGTH = 15, DATA = $queryvalue;
/* Associate the cursor with the parameter value */
$open democursor using sql descriptor $demodesc;

/*Reuse the descriptor to determine the contents of the Select-list*/
$describe qid using sql descriptor 'demodesc';
$set descriptor 'demodesc' $desc_count = count;

printf("There are %d returned columns:\n", desc_count);
/* Print out what DESCRIBE returns */
for (i = 1; i <= desc_count; i++)
    prsqlda(i);

printf("\n\n");
for (;;)
{
    $fetch democursor using sql descriptor 'demodesc';
    if (sqlca.sqlcode != 0) break;
    for (i = 1; i <= desc_count; i++)
    {
        $ get descriptor 'demodesc' value $i $result = data;
        printf("%s ", result);
    }
    printf("\n");
}

$close democursor;

#ifdef TRANS;
$commit work;
#endif;

printf("\nProgram Over.\n");
}

prsqlda(index)
    $ parameter int index;
{
    $ int type;
    $ int len;
    $ char name[40];

    $ get descriptor 'demodesc' value $index
        $type = type,
        $len = length,
        $name = name;
    printf("    Column %d: type = %d, len = %d, name = %s\n",
        index, type, len, name);
}

```

Figure 9-13 Program that uses a system descriptor area for input and output values

Using an *sqlda* Structure

Follow these steps to use an **sqlda** structure with a SELECT statement that receives WHERE-clause values at run time:

1. Declare host variables to interactively hold the data obtained from the user. Declare a pointer to an **sqlda** structure to hold the parameter information.
2. Prepare the SELECT statement. It must contain a ? for each missing value in the WHERE clause.
3. Associate a cursor with the prepared SELECT statement by using the DECLARE statement.
4. Using C code, determine the number and type of each parameter (or question mark) in the prepared statement.
5. Assign the total number of parameters to the **sqld** field in the structure **sqlvar_struct**. Allocate the memory for each variable in the **sqlvar_struct** in the **sqlda** structure.

6. Assign values to the appropriate fields in the `sqlvar_struct`, as follows:

sqlvar	is a pointer to an array of <code>sqlvar_struct</code> structures.
sqltype	is an integer that represents the data type of the column involved. You can use the predefined constants in <code>sqltypes.h</code> rather than literal values.
sqllen	is the size, in bytes, of a character array (for CHAR and VARCHAR); the encoded qualifiers for DATETIME, and INTERVAL.
sqldata	is the name of the host variable that contains the data.
sqlind	is the name of the indicator variable.
7. Once you set all the necessary information for each parameter, open a cursor using the `sqlda` pointer to associate the host variables (and their contents) with the question marks in the prepared SELECT statement. For example, if you declared `in_vals` as the pointer to an `sqlda` structure, the OPEN statement is as follows:

```
$OPEN q_cursor USING DESCRIPTOR in_vals ;
```

8. Use the FETCH statement to get the first set of values returned by the prepared SELECT statement. Repeat the FETCH statement until no more rows are returned.
9. Close the cursor using the CLOSE statement.

Non-SELECT Statements That Receive Values at Run Time

There are three kinds of non-SELECT statements that receive values at run time:

- INSERT
- DELETE...WHERE
- UPDATE...WHERE

For an INSERT statement, the values being inserted are called the parameters. The type and number of the values being inserted are not known at compile time, so you cannot simply use host variables to hold the data being inserted.

The DELETE and UPDATE statements can both contain a WHERE clause. Using a dynamic DELETE or UPDATE statement is similar to using a SELECT statement that receives WHERE-clause values at run time, but your program associates the parameters to the prepared statement in an EXECUTE statement rather than in an OPEN statement. If a DELETE or UPDATE statement is used dynamically, your program must take the following steps:

1. The statement must first be prepared using the PREPARE statement.
2. Your code should describe the statement and check the value of `sqlwarn4` in the `sqlca` structure. If `sqlwarn4` contains a **W**, the DELETE or UPDATE statement does not contain a WHERE clause and, if executed, all of the rows in the table are deleted or updated.
3. The prepared statement id is executed using the EXECUTE statement with the appropriate USING clause. You can use host variables, a system descriptor area, or an `sqllda` structure to hold the parameters.

Using Host Variables

If the number of parameters and their data types are known at compile time, a prepared SQL statement can be executed with the names of the host variables that hold the parameter data. A DELETE or UPDATE statement that requires three parameters, the value of which is stored in `$hvar1`, `$hvar2`, and `$hvar3` and which was prepared with the `stateid` identifier, can be run with the following statement:

```
$EXECUTE stateid USING $hvar1, $hvar2, $hvar3;
```

Using a System Descriptor Area

Take the following steps to use a system descriptor area to hold the information about the parameters in the WHERE clause:

1. Allocate a descriptor large enough to hold the parameters with the ALLOCATE DESCRIPTOR statement.

```
$ALLOCATE DESCRIPTOR 'up_desc' WITH MAX 10;
```

2. For each parameter, use the SET DESCRIPTOR statement to set the TYPE and associate the host variable that holds the data with the descriptor field. The following example sets the second value in the `up_desc` descriptor to an integer value (TYPE = 2) that receives its data from the

host variable called **h_int**. This is used for the second ? in the WHERE clause.

```
$SET DESCRIPTOR 'up_desc' VALUE 2
      TYPE = 2, DATA = $h_int;
```

3. Use the EXECUTE statement with the USING SQL DESCRIPTOR clause. For example, the following statement associates information about the parameters held in **up_desc** with the prepared statement called **stateid**:

```
$EXECUTE stateid USING SQL DESCRIPTOR 'up_desc';
```

Example of Using SET DESCRIPTOR with a Locator Structure

The program in Figure 9-14 is an example of how to use a dynamic INSERT statement. The INSERT statement “insert into a value (?,?)” is hard coded here, but it can be created at run time. The two values inserted are blobs. Notice the use of an **ESQL/C** macro **CLOCTYPE**.

```
#include <locator.h>

$ define CLOCTYPE 113;

main()
{
    $ int i;
    $ int cnt;
    $ loc_t loc1;
    $ loc_t loc2;

    $ create database test;
    chkerr("CREATE DATABASE");
    $ create table a (t1 text not null, t2 text);
    chkerr("CREATE TABLE");
    $ allocate descriptor "desc";
    chkerr("ALLOCATE");
    /* The INSERT statement could have been created at run-time. */
    $ prepare sid from "insert into a values (?, ?)";
    chkerr("PREPARE");
    $ describe sid using sql descriptor "desc";
    chkerr("DESCRIBE");
    $ get descriptor "desc" $cnt = count;
    chkerr("GET DESCRIPTOR");
    for (i = 1; i <= cnt; i++)
        prsqlda(i);

    loc1.loc_loctype = loc2.loc_loctype = LOCFNAME;
    loc1.loc_fname = loc2.loc_fname = "d1.ec";
    loc1.loc_size = loc2.loc_size = -1;
    loc1.loc_oflags = LOC_RDONLY;
    $ set descriptor "desc" value 1 type = CLOCTYPE,
        data = $loc1;
```

```

chkerr("SET DESC 1");
$ set descriptor "desc" value 2 type = CLOCTYPE,
    data = $loc2;
chkerr("SET DESC 2");
$ execute sid using sql descriptor "desc";
chkerr("EXECUTE");
loc1.loc_loctype = loc2.loc_loctype = LOCFNAME;
loc1.loc_fname = "out1";
loc2.loc_fname = "out2";
loc1.loc_oflags = LOC_WONLY;

$ select * into $loc1, $loc2 from a;
chkerr("SELECT");
$ close database;
chkerr("CLOSE DATABASE");
$ drop database test;
chkerr("DROP DATABASE test");
}

prsqlda(index)
$ parameter int index;
{
    $ int type;
    $ int len;
    $ int nullable;
    $ char name[40];

    $ get descriptor 'desc' value $index
        $type = type,
        $len = length,
        $nullable = nullable,
        $name = name;
    printf("    Column %d: type = %d, nullable = %d, len = %d, name = %s\n",
        index, type, nullable, len, name);
}

chkerr(s)
char *s;
{
if (SQLCODE) printf("%s error %ld\n", s, SQLCODE);
}

```

Figure 9-14 Example of using a system descriptor area with an *INSERT* statement

Using an *sql*da Structure

If the number of parameters is not known until run time and the parameter data is entered into the *sql*da structure pointed to by *in_vals*, the dynamic statement *stateid* is run with the statement:

```
EXECUTE stateid USING DESCRIPTOR in_vals;
```

List of INFORMIX-ESQL/C Routines

List of Routines 3



List of Routines

All of the library routines provided with INFORMIX-ESQL/C are listed in alphabetical order, as follows:

Function name	Description	Page
bycmp	Compares two groups of contiguous bytes	3-5
bycopy	Copies bytes from one area to another	3-7
byfill	Fills the specified area with a character	3-9
byleng	Counts the number of bytes in a string	3-11
decadd	Add two decimal numbers	4-25
deccmp	Compare two decimal numbers	4-31
deccopy	Copy a decimal number	4-33
deccvasc	Convert a C char type to a decimal type	4-6
deccvdbl	Convert a C double type to a decimal type	4-20
deccvint	Convert a C int type to a decimal type	4-12
deccvlong	Convert a C long type to a decimal type	4-16
decdiv	Divide two decimal numbers	4-25
dececv	Convert a decimal value to an ASCII string	4-35
decfcvt	Convert a decimal value to an ASCII string	4-35
decmul	Multiply two decimal numbers	4-25
decround	Round a decimal number	4-41
decsb	Subtract two decimal numbers	4-25
dectoasc	Convert a decimal type to a C char type	4-9
dectodbl	Convert a decimal type to a C double type	4-22
dectoint	Convert a decimal type to a C int type	4-14
dectolong	Convert a decimal type to a C long type	4-18
dctrunc	Truncate a decimal number	4-43
dtcurrent	Get current date and time	5-30
dtvasc	Convert an ANSI-compliant character string to datetime	5-32
dtcvfmtasc	Convert a character string to datetime	5-35
dttextend	Change the qualifier of datetime	5-37
dttoasc	Convert a datetime to an ANSI-compliant character string	5-42
dttofmtasc	Convert a datetime to a character string	5-42
incvasc	Convert an ANSI-compliant character string to interval	5-44
incvfmtasc	Convert a character string to interval	5-46
intoasc	Convert an interval to an ANSI-compliant character string	5-48

intofmtasc	Convert an interval to a string	5-50
ldchar	Copies a fixed-length string to a null-terminated string	3-13
rdatestr	Convert an internal format to string	5-5
rdayofweek	Return the day of the week	5-7
rdefmtdate	Convert string to an internal format	5-9
rdownshift	Converts all letters to lowercase	3-15
rfmtdate	Convert an internal format to a string	5-12
rfmtdec	Convert a decimal type to a formatted string	4-45
rfmtdouble	Convert a double to a string	2-37
rfmtlong	Convert a long integer to a formatted string	2-40
rgetmsg	Convert an error message integer into a string	7-13
risnull	Check whether the C variable is null	2-13
rjulmdy	Return month, day, and year from an internal format	5-15
rleapyear	Determine whether it is a leap year	5-17
rmdyjul	Return an internal format from month, day, and year	5-19
rsetnull	Set a C variable to null	2-16
rstod	Converts a string to double	3-16
rstoi	Converts a string to short	3-18
rstol	Converts a string to long	3-20
rstrdate	Convert a string to an internal format	5-21
rtoday	Return a system date in internal format	5-23
rtypalign	Align data on a proper type boundary	2-19
rtypmsize	Give byte size of SQL data types	2-22
rtypname	Convert a data type to a string	2-25
rtypwidth	Give minimum conversion byte size	2-28
rupshift	Convert all letters to uppercase	3-22
sqlbreak	Send the database server a request to stop processing	8-4
sqldetach	Detach child process from parent process	8-5
sqlexit	Terminate a database server process	8-6
sqlstart	Start a database server process	8-7
stcat	Concatenates one string to another	3-23
stchar	Copies a null-terminated string to a fixed-length string	3-25
stcmp	Compares two strings	3-27
stcopy	Copies one string to another string	3-29
stleng	Counts the number of bytes in a string	3-30

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX; DB2; DB2 Universal Database; Distributed Relational Database Architecture; NUMA-Q; OS/2, OS/390, and OS/400; IBM Informix®; C-ISAM®; Foundation.2000™; IBM Informix® 4GL; IBM Informix® DataBlade® Module; Client SDK™; Cloudscape™; Cloudsync™; IBM Informix® Connect; IBM Informix® Driver for JDBC; Dynamic Connect™; IBM Informix® Dynamic Scalable Architecture™ (DSA); IBM Informix® Dynamic Server™; IBM Informix® Enterprise Gateway Manager (Enterprise Gateway Manager); IBM Informix® Extended Parallel Server™; i.Financial Services™; J/Foundation™; MaxConnect™; Object Translator™; Red Brick Decision Server™; IBM Informix® SE; IBM Informix® SQL; InformiXML™; RedBack®; SystemBuilder™; U2™; UniData®; UniVerse®; wintegrate® are trademarks or registered trademarks of International Business Machines Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Windows, Windows NT, and Excel are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names used in this publication may be trademarks or service marks of others.

A

ANSI

- checking for Informix extensions 1-23
- compliance Intro-13
- error handling considerations 7-6
- function parameter syntax 1-17
- use of EXEC SQL keywords 1-7

ansi flag 1-21, 1-23

Array

- and non-NULL SQL value 1-18
- declaration of variables 1-14
- in a host-variable typedef 1-15
- limit on size 1-14
- use within ESQL/C statements 1-14

B

Binary Large Object (blob) 1-13, 6-3

Blob

- See* Binary Large Object

BLOB data type 1-13

Block

- defining 1-12
- nesting 1-12

bycmpr() library function 3-5

bycopy() library function 3-7

byfill() library function 3-9

byleng() library function 3-11

BYTE data type 2-4, 6-3

C

C function calls

- DATE type routines listed 5-4
- DATETIME type routines listed 5-28
- DECIMAL type routines listed 4-5
- INTERVAL type routines listed 5-28
- numeric formatting routines listed 2-31

- C program structure
 - dynamic SQL statements 9-4
 - embedding SQL statements 1-4
 - error handling 7-5
 - header files 1-5
 - host variables 1-9
 - indicator variables 1-17
 - sqlca structure 7-5
 - sqlda structure 9-4
 - C programs
 - compiling 1-20
 - embedding SQL statements in 1-4
 - Case sensitivity 1-4
 - CHAR data type 1-13
 - CLOCTYPE macro 9-32
 - Colon (:)
 - between main variable and indicator variable 1-18
 - preceding host variables 1-9
 - specifying indicator variable 1-18 - Comments, in ESQL/C program 1-5
 - Compiler
 - and .ec extension 1-20
 - CC 1-27
 - creating an object file 1-20
 - diagnosing errors 1-11
 - esqlcargs 1-21
 - esql, for ESQL/C programs 1-20
 - linking library functions 3-3
 - preprocessing 1-20
 - syntax 1-21
 - Conditional compilation statements 1-7
 - Conventions
 - command line Intro-5
 - typographical Intro-5
 - Conversion
 - from DATE to DATETIME 2-10, 5-27
 - from DATETIME to DATE 2-9, 5-27
 - of data types 2-6
 - when fetching DATETIME 5-26
 - when fetching INTERVAL 5-26
 - when storing DATETIME 5-27
 - when storing INTERVAL 5-27
 - COUNT field
 - after a DESCRIBE 9-14
 - setting for input values 9-25
 - cs compile option Intro-15, 1-21, 1-26
 - Cursor
 - multiple Intro-16
 - names, case sensitivity 1-26
 - names, case sensitivity 1-4
 - scope of name 1-26
- ## D
- dash, double (--) 1-5
 - Data conversion
 - among data types 2-6
 - from DATE to DATETIME 2-10, 5-27
 - from DATETIME to DATE 2-9, 5-27
 - when fetching DATETIME 5-26
 - when fetching INTERVAL 5-26
 - when storing DATETIME 5-27
 - when storing INTERVAL 5-27
 - DATA field
 - after a FETCH 9-14
 - setting for input values 9-25
 - Data types
 - and declaration of array 1-14
 - DATE 5-3
 - DATETIME 5-24
 - DECIMAL 4-3
 - decimal structure 4-4
 - decimal.h 4-4
 - dec_t 4-4
 - defined in sqltypes.h 9-7
 - dtime_t 5-24
 - fixchar 2-6
 - INTERVAL 5-24
 - intrvl_t 5-24
 - relation of C and SQL 1-12, 2-3
 - string 2-6
 - type conversion 2-6
 - Database
 - engine control routine 8-4, 8-6, 8-7
 - stores Intro-11
 - Database variable, discrepancy with host variable 2-6
 - DATE data type 1-13, 2-4
 - converting to DATETIME data type 2-10, 5-27
 - definition of 5-3
 - rdatestr() 5-5
 - rdayofweek() 5-7
 - rdefmtdate() 5-9
 - rfmtdate() 5-12
 - rjulmdy() 5-15
 - rleapyear() 5-17
 - rmdyjul() 5-19
 - rstrdate() 5-21

rtoday() 5-23
 DATETIME
 columns defined 5-24
 converting to DATE 2-9, 5-27
 data conversion when fetching 5-26
 data conversion when storing 5-27
 declaration of host variable 5-25
 dynamically allocating structures for
 9-20
 fetching values 5-26
 storing values 5-27
 DATETIME data type 1-13, 2-4
 columns defined 5-24
 converting to DATE data type 2-9,
 5-27
 data conversion when fetching 5-26
 data conversion when storing 5-27
 declaration 5-25
 definition of 5-24
 dtcurrent() 5-30
 dtcvasc() 5-32
 dtextend() 5-37
 dttoasc() 5-40
 fetching values 5-26
 host variables of 5-25
 incvasc() 5-44
 intoasc() 5-48
 storing values 5-27

 DBANSIWARN environment variable
 1-23, 7-11
 decadd() decimal manipulation routine
 4-25
 deccmp() decimal manipulation routine
 4-31
 deccopy() decimal manipulation routine
 4-33
 deccvasc() decimal manipulation routine
 4-6
 deccvint() decimal manipulation routine
 4-12
 deccvlong() decimal manipulation
 routine 4-16
 decdiv() decimal manipulation routine
 4-25
 dececvt() decimal manipulation routine
 4-35
 decfcvt() decimal manipulation routine
 4-35
 Decimal arithmetic
 addition 4-25
 division 4-25
 multiplication 4-25
 subtraction 4-25
 DECIMAL data type 1-13, 2-4
 decadd() 4-25
 deccmp() 4-31
 deccopy() 4-33
 deccvasc() 4-6
 deccvint() 4-12
 deccvlong() 4-16
 decdiv() 4-25
 dececvt() 4-35
 decfcvt() 4-35
 decimal structure shown 4-4
 decmul() 4-25
 decround() 4-41
 decsub() 4-25
 dectoasc() 4-9
 dectodbl() 4-22
 dectoint() 4-14
 dectolong() 4-18
 dectrunc() 4-43
 definition of 4-3
 scale and precision 2-9
 Declaration
 of DATETIME 5-25
 of INTERVAL 5-25

decmul() decimal manipulation routine 4-25
 decround() decimal manipulation routine 4-41
 decsub() decimal manipulation routine 4-25
 dectoasc() decimal manipulation routine 4-9
 dectodbl() decimal manipulation routine 4-22
 dectoint() decimal manipulation routine 4-14
 dectolong() decimal manipulation routine 4-18
 detrunc() decimal manipulation routine 4-43
 dec_t typedef shown 4-4
 define instruction 1-7, 1-8
 DELETE statement,dynamic 9-30
 Demonstration database
 copying Intro-12
 example programs 1-28
 installation script Intro-11
 overview Intro-11
 DESCRIBE statement
 analyzing returned values 9-7
 determining the statement type 9-4
 sqlca after executing 7-5
 values of SQLCODE after 9-9
 with sqlvar_struct 9-17
 Dimensions of a host-variable array 1-14
 dispcat_pic
 before using 6-22
 loading cat_picture 6-23
 summary 6-22
 the program 6-26 to 6-42
 using blobload 6-24
 using the conditional display logic 6-23
 Documentation notes Intro-7
 Documentation, other useful Intro-4
 Dollar sign (\$)
 between main variable and indicator variable 1-18
 displaying a literal 2-32
 embedding SQL statements in C routines 1-4
 or EXEC SQL keywords 1-4, 1-7
 preceding host variables 1-9
 with include files 1-6
 \${ SeeDollar-brace sign
 Dollar-brace sign (\$)
 ANSI standard 1-12
 using with block 1-12
 DOUBLE PRECISION data type 1-13
 dtcurrent() datetime routine 5-30
 dtcvasc() datetime routine 5-32
 dtextend() datetime routine 5-37
 dtime structure shown 5-24
 dtime_t typedef shown 5-24
 dttoasc() datetime routine 5-40
 Dynamic management
 constants 1-6
 memory management methods 9-5
 non-parameterized non-selects 9-4
 non-parameterized selects 9-4
 parameterized non-selects 9-5
 parameterized selects 9-5
 use of rtypalign() 9-20
 Dynamic management statements, sqlca structure 9-4
 Dynamic SQL *See* Dynamic Management

E

e compile option 1-21
 .ec extension, meaning of 1-20
 elif instruction Intro-16, 1-7, 1-9
 else instruction 1-7, 1-9
 Embedding SQL statements 1-4
 endif instruction 1-7, 1-9
 Error handling
 after a PREPARE Statement 7-12
 after an EXECUTE Statement 7-12
 checking for no rows 7-6
 checking for success 7-5
 in line tests 7-8
 role of sqlca.h 1-5
 with sqlca structure 7-5
 Error messages Intro-4
 esql
 and library functions 3-3
 compiler for ESQL/C 1-20
 syntax 1-26
 EUname compile option 1-22
 EXEC SQL keywords
 and embedded SQL statements 1-4

- and host variables 1-8
- and include file 1-7
- EXECUTE IMMEDIATE statement, using 9-12
- Expressions, formatting 2-31

F

- Fetching
 - DATETIME values 5-26
 - INTERVAL values 5-26
- File extensions
 - .ec 1-20
- File-open-mode flags 6-14
- Fixchar host variables 2-10
- Flag, warning
 - See* Warnings
- FLOAT data type 1-13, 2-4
- Formatting numeric expressions
 - examples 2-32
 - overview 2-31
 - valid characters 2-31
- Formatting numeric strings 2-31
- Formatting routines, numeric 2-31
- Function library
 - bycmptr() 3-5
 - bycopy() 3-7
 - byfill() 3-9
 - byleng() 3-11
 - ldchar() 3-13
 - rdownshift() 3-15
 - rfmtdec() 4-45
 - rfmtdouble() 2-37
 - rfmtlong() 2-40
 - rgetmsg() 7-13
 - risnull() 2-13
 - rsetnull() 2-16
 - rstod() 3-16
 - rstoi() 3-18
 - rstol() 3-20
 - rtyalign() 2-19
 - rtypmsize() 2-22
 - rtypname() 2-25
 - rtypwidth() 2-28
 - rupshift() 3-22
 - sqlbreak() 8-4
 - sqlexit() 8-6
 - sqlstart() 8-7
 - stcat() 3-23
 - stchar() 3-25

- stcmptr() 3-27
- stcopy() 3-29
- stleng() 3-30
- using esql 3-3

- Function, parameter 1-16

G

- g compile option 1-22

H

- Header file
 - sqlca.h 1-5
 - sqlda.h 1-5
 - sqltype.h 1-6
 - sqltypes.h 1-6
 - syntax for including 1-6
- Host variable
 - and standard C typedef expressions 1-15
 - C function calls for DATETIME 5-28
 - C function calls for INTERVAL 5-28
 - DATETIME data type 5-25
 - declared as C variable 1-10
 - declared with C initializer
 - expressions 1-11
 - definition of 1-9
 - discrepancy with database variable 2-6
 - fetching DATETIME value 5-26
 - fetching INTERVAL value 5-26
 - in non-parameterized SELECT 9-13
 - in parameterized SELECT 9-21
 - of type DATETIME 5-25
 - of type INTERVAL 5-25
 - preceded by EXEC SQL 1-10
 - preceded by \$ 1-9, 1-10
 - preceded by : 1-9
 - scope rules 1-11
 - setting to NULL value 1-15
 - storing DATETIME value 5-27
 - storing INTERVAL value 5-27
 - structure of DATETIME value 5-24
 - structure of INTERVAL value 5-24
 - testing for NULL value 1-15

Host variables 3-32
hyphen, double(--)
1-5

I

icheck flag 1-22
 compiling programs with 1-22
 errors returned 1-18
ifdef instruction 1-7, 1-9
ifndef instruction 1-7, 1-9
Include files 5-24
 automatic inclusion 1-6
 datetime.h 5-28
 decimal.h 4-4
 preprocessor statement for 1-7
 sqltypes.h 9-7, 9-8
 syntax for 1-6
 to check success of ESQL/C
 statements 1-6
incvasc() datetime routine 5-44
INDICATOR keyword
 and indicator variable 1-18
Indicator variable
 and associated host variable 1-17
 and INDICATOR keyword 1-18
 checking for missing indicator 1-23
 definition of 1-17
 how to specify in SQL statement 1-18
 main variable 1-17
 specification of 1-18
 truncation of 1-18
 with NULL and NOT NULL values
 1-18
INSERT statement,dynamic 9-30
INSERTstatement, dynamic 9-5
INTEGER data type 1-13, 2-4
INTERVAL
 data conversion when fetching 5-26
 data conversion when storing 5-27
 declaration of host variable 5-25
 dynamically allocating structures for
 9-20
 fetching values 5-26
 storing values 5-27
INTERVAL data type 1-13, 2-4
 data conversion when fetching 5-26
 data conversion when storing 5-27
 declaration 5-25
 definition of 5-24
 fetching values 5-26

 host variables of 5-25
 storing values 5-27
intoasc() datetime routine 5-48
intrvl structure shown 5-24
intrvl_t typedef shown 5-24

L

ldchar() library function 3-13
LENGTH field,with character or varchar
 data 9-25
Libraries
 C iii
 using with esql 1-27
Library functions
 bycmp() 3-5
 bycopy() 3-7
 byfill() 3-9
 byleng() 3-11
 ldchar() 3-13
 rdownshift() 3-15
 rfmtdec() 4-45
 rfmtdouble() 2-37
 rfmtlong() 2-40
 rgetmsg() 7-13
 risnull() 2-13
 rsetnull() 2-16
 rstdod() 3-16
 rstoi() 3-18
 rstol() 3-20
 rtypalign() 2-19
 rtypmsize() 2-22
 rtypname() 2-25
 rtypwidth() 2-28
 rupshift() 3-22
 sqlbreak() 8-4
 sqlexit() 8-6
 sqlstart() 8-7
 stcat() 3-23
 stchar() 3-25
 stcmp() 3-27
 stcopy() 3-29
 stleng() 3-30
local compile option Intro-15, 1-22
Locating a blob
 in a named file 6-14
 in an open file 6-10
 in memory 6-6
 reading into a named file 6-15
 reading into an open file 6-11

- reading into memory 6-7
- with close function 6-18
- with open function 6-17
- with program functions 6-17
- with read function 6-18
- with write function 6-19
- writing from a named file 6-16
- writing from an open file 6-12
- writing from memory 6-9

Locator structure

- defined in 6-4
- description of 6-3

locator.h 6-4

- See also* Locator structure

LOC_DESCRIPTOR flag 6-19

- description of 6-19
- using with blobs on WORM optical disk 6-19

log compile option Intro-15, 1-22

M

Machine notes Intro-8

macros

- with datetime and interval data types 5-28
- with varchars 3-3

MONEY data type 1-13, 2-4

N

New features Intro-14

nln compile option 1-22

Non-parameterized non-select statements 9-4

Non-parameterized SELECT statements 9-13

Non-parameterized selects 9-4

Null value

- in host variables 1-15
- inserting into table using indicator value 1-19
- returned in indicator 1-18
- risnull with 1-15
- rsetnull with 1-15

NUMERIC data type 1-13

Numeric expressions

- example formats 2-32
- formatting 2-31

- rfmtdec() routine 4-45
- rfmtdouble() routine 2-37
- rfmtlong() routine 2-40
- valid characters 2-31

Numeric formatting routines 2-31

O

o compile option 1-22

On-line files Intro-7

P

Parameter keyword 1-16

Parameterized non-SELECT statements 9-12, 9-30

Parameterized non-select statements 9-5

Parameterized SELECT statements 9-21

Parameterized selects 9-5

Parameter, function 1-16

Pointer, declaring as host variable 1-16

Preprocessor

- defining and undefining 1-24
- EXEC SQL include statement 1-7
- role in ESQL/C 1-3
- search sequence for included files 1-8
- stage 1 1-7
- stage 2 1-7
- \$else statement 1-9
- \$endif statement 1-9
- \$ifdef statement 1-9
- \$ifndef statement 1-9
- \$include statement 1-7, 1-9
- \$undef statement 1-8

Program

- compiling 1-20
- including header files 1-5
- preprocessing 1-20
- preprocessor statement to include files 1-7

R

- rdatestr() date manipulation routine 5-5
- rdayofweek() date manipulation routine 5-7
- rdefmtdate() date manipulation routine 5-9
- rdownshift() library function 3-15
- REAL data type 1-13
- Release notes Intro-7
- rfmtdate() date manipulation routine 5-12
- rfmtdec() numeric formatting routine 4-45
- rfmtdouble() numeric formatting routine 2-37
- rfmtlong() numeric formatting routine 2-40
- rgetmsg() library function 7-13
- risnull() library function 2-13
- rjulmdy() date manipulation routine 5-15
- rleapyear() date manipulation routine 5-17
- rmdyjul() date manipulation routine 5-19
- Routine
 - decadd() 4-25
 - deccmp() 4-31
 - deccopy() 4-33
 - deccvasc() 4-6
 - deccvdbl() 4-20
 - deccvint() 4-12
 - deccvlong() 4-16
 - decdiv() 4-25
 - dececv() 4-35
 - decfcvt() 4-35
 - decmul() 4-25
 - decround() 4-41
 - decsub() 4-25
 - dectoasc() 4-9
 - dectodbl() 4-22
 - dectoint() 4-14
 - dectolong() 4-18
 - detrunc() 4-43
 - dtcurrent() 5-30
 - dtcvasc() 5-32
 - dttextend() 5-37
 - dttoasc() 5-40
 - incvasc() 5-44
 - intoasc() 5-48
 - rdatestr() 5-5
 - rdayofweek() 5-7
 - rdefmtdate() 5-9
 - rfmtdate() 5-12
 - rfmtdec() 4-45
 - rfmtdouble() 2-37
 - rfmtlong() 2-40
 - rjulmdy() 5-15
 - rleapyear() 5-17
 - rmdyjul() 5-19
 - rstrdate() 5-21
 - rtoday() 5-23
- rsetnull() library function 2-16
- rstod() library function 3-16
- rstoi() library function 3-18
- rstol() library function 3-20
- rstrdate() date manipulation routine 5-21
- rtoday() date manipulation routine 5-23
- rtypalign()
 - use with dynamic memory allocation 9-20
- rtypalign() library function 2-19
- rtypmsize() library function 2-22
- rtypname() library function 2-25
- rtypwidth() library function 2-28
- rupshift() library function 3-22

S

- Scope of
 - cursor names 1-26
 - statement ids 1-26
 - variables 1-11
- SELECT statement
 - non-parameterized 9-13
 - parameterized 9-21
- SERIAL data type 1-13, 2-4
- SMALLFLOAT data type 1-13
- SMALLINT data type 1-13, 2-4
- SQL
 - and EXEC SQL keywords 1-4
 - dynamic statements and sqllda 9-4
 - embedding statements in C programs 1-4
- sqlbreak() library function 8-4
- sqlca structure
 - after DESCRIBE executes 7-5
 - and error handling 7-5

- and truncation 1-18
- definition of 7-5
- how to use 7-3
- See also* Error handling
- sqlcaw_s structure 7-10
- sqlcode field 7-5
- SQLCODE variable
 - after a DESCRIBE statement 9-4
 - definition and use 7-7
- sqlda structure
 - definition of 9-6
 - dynamic SQL statements 9-4
 - values for sqltype field 9-8
- sqlexit() library function 8-6
- sqlstart() library function 8-7
- sqlstype.h file
 - contents and use 9-10
- sqltypes.h file
 - contents 2-5
 - contents and use 9-7
- Statement id
 - case sensitivity 1-4
 - scope of 1-26
- stcat() library function 3-23
- stchar() library function 3-25
- stcmpr() library function 3-27
- stcopy() library function 3-29
- stleng() Library function 3-30
- stores database
 - copying Intro-12
 - creating on INFORMIX-OnLine Intro-12
 - creating on INFORMIX-SE Intro-12
 - overview Intro-11
- Storing DATETIME values 5-27
- Storing INTERVAL values 5-27
- String host variables 2-10
- Strings, formatting numeric 2-31
- struct
 - decimal shown 4-4
 - dtime shown 5-24
 - intrvl shown 5-24
 - sqlvar shown 9-7
- Structures
 - declared as host objects 1-14
 - dynamic SQL statements and sqlda 9-4
 - error handling and sqlca 7-5

- nesting 1-14
- sqlda defined 9-6
- using sqlda 9-29
- System descriptor area
 - definition 9-5
 - description 9-5

T

- TEXT data type 1-13, 2-4, 6-3
- Trailing blanks 3-33
- TYPE field
 - after a DESCRIBE 9-14
 - setting for input 9-31
 - setting for input values 9-25
 - values for 9-7
- typedef
 - as host variable 1-15
 - dec_t shown 4-4
 - dtime_t shown 5-24
 - intrvl_t shown 5-24
- Typographical conventions Intro-5

U

- undef instruction 1-7, 1-8
- Union
 - in a typedef-host variable 1-15
- UPDATE statement,dynamic 9-30

V

- V compile option 1-22
- Values
 - fetching DATETIME 5-26
 - fetching INTERVAL 5-26
 - storing DATETIME 5-27
 - storing INTERVAL 5-27
- VARCHAR conversion 2-11
- VARCHAR data type 1-13, 2-4, 3-3
 - as host variable in ESQL/C 3-32
 - macros 3-33
 - role of varchar.h 1-6
- Variable
 - conversion of data types 2-6
 - database versus host 2-6
 - host 1-9
 - indicator 1-17
 - rules for DECIMAL type 2-8

W

Warnings

- checking for 7-10
- DBANSIWARN 1-23
- redirecting 1-24
- with WHENEVER statement 7-11

WHENEVER statement

- reducing error checking code 7-9
- using 7-9

X

- xopen compile option Intro-15, 1-22, 1-25
- values for the TYPE field when using 9-8

X/Open

- support of dynamic SQL Intro-14
- warnings on extensions 1-25
- xopen compile option 1-22