IBM Informix
Version 3.50

*IBM Informix
ESQL/C Programmer's Manual*

IBM

IBM Informix
Version 3.50

IBM Informix
ESQL/C Programmer's Manual

IBM

Contents

Introduction

This introduction provides an overview of the information in this publication and describes the conventions it uses.

About this publication

This publication explains how to use IBM® Informix® ESQL/C, the Informix implementation of embedded Structured Query Language (SQL) for C (ESQL/C), to create client applications with database-management capabilities. This publication is a complete guide to the features of Informix ESQL/C that enable you to interact with the database server, access databases, manipulate the data in your program, and check for errors. However, certain operating systems do not support every documented ESQL/C feature. Check the IBM Informix Client Software Development Kit (Client SDK) Machine Notes for your operating system to determine exactly which features do not operate in your environment.

This publication progresses from general topics to more advanced programming techniques and examples.

Types of users

This publication is written primarily for C programmers who want to embed SQL statements in their programs to access IBM Informix databases.

This publication assumes that you have the following background:
- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational or object-relational databases, or exposure to relational database concepts
- C language programming

The following users might also be interested in some of the topics in this book:
- Database server administrators
- Performance engineers

If you have limited experience with relational databases, SQL, or your operating system, see the *IBM Informix Getting Started Guide* for your database server for a list of supplementary titles.

Software dependencies

In places where this publication presents database server-specific information, this information applies to one of the following database servers:
- IBM Informix Version 11.50
- IBM Informix Extended Parallel Server (XPS)
- IBM Informix SE

If you are using a database server that is not listed here, see your release notes for information about client behavior on your database server.

Note: All versions of IBM Informix ESQL/C support IPv6. For more information about using IPv6 see the *IBM Informix Administrator's Guide*.

Assumptions About Your Locale

IBM Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a GLS (Global Language Support) locale.

The examples in this publication are written with the assumption that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale.

For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration Databases

The DB-Access utility, which is provided with your IBM Informix database server products, includes one or more demonstration databases that contain information about a fictitious wholesale sporting-goods distributor. You can create and populate these demonstration databases with command files that are included with the database server.

Many examples in IBM Informix publications are based on these databases. For a complete explanation of how to create and populate the demonstration databases, refer to your *IBM Informix DB–Access User's Guide*. For a description of the demonstration databases and their contents, see your *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases reside in the **$INFORMIXDIR/bin** directory on UNIX and in the **%INFORMIXDIR%\bin** directory on Windows.

For additional information about IBM Informix ESQL/C and the demonstration database, see Appendix A, "The ESQL/C example programs," on page A-1.

What's new in ESQL/C for Client SDK, Version 3.50

This publication includes information about new features and changes in existing functionality. The following changes and enhancements are relevant to this publication.

Table 1. What's New in IBM Informix ESQL/C Programmer's Manual for Version 3.50.xC8

Overview	Reference
New editions and product names	For more information about the Informix product family, go to http://www.ibm.com/software/data/informix/.
IBM Informix Dynamic Server editions were withdrawn and new Informix editions are available. Some products were also renamed. The publications in the Informix library pertain to the following products:	
• IBM Informix database server, formerly known as IBM Informix Dynamic Server (IDS)	
• IBM Informix OpenAdmin Tool for Informix, formerly known as OpenAdmin Tool for Informix Dynamic Server (IDS)	
• IBM Informix SQL Warehousing Tool, formerly known as Informix Warehouse Feature	

Table 2. What's New in IBM Informix ESQL/C Programmer's Manual for Version 3.50.xC4

Overview	Reference
Error Checking During Data Transfer	"Error Checking During Data Transfer" on page 12-26
With the **IFX_LOB_XFERSIZE** environment variable, you can specify the number of kilobytes in a CLOB or BLOB to transfer from a client application to the database server before checking whether an error has occurred.	

Table 3. What's New in IBM Informix ESQL/C Programmer's Manual for Version 3.50.xC1

Overview	Reference
This release introduces two data types, BIGINT and BIGSERIAL. These data types have the same ranges as the existing INT8 and SERIAL8 data types, and have storage and computational advantages.	"SQL Data Type Constants" on page 3-4
	"Choosing data types for host variables" on page 3-1

Example Code Conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
```

```
     WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB-Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional Documentation

Documentation about this release of IBM Informix products is available in various formats.

All of the product documentation (including release notes, machine notes, and documentation notes) is available from the information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

Compliance with Industry Standards

IBM Informix products are compliant with various standards.

IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.
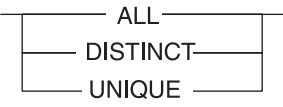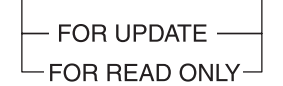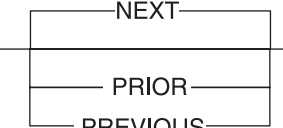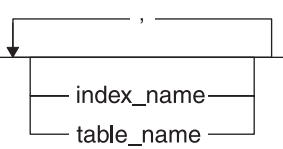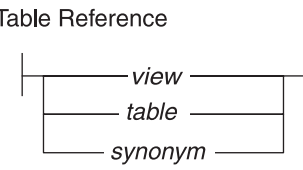
The IBM Informix Geodetic DataBlade Module supports a subset of the data types from the Spatial Data Transfer Standard (SDTS)—Federal Information Processing Standard 173, as referenced by the document Content Standard for Geospatial Metadata, Federal Geographic Data Committee, June 8, 1994 (FGDC Metadata Standard).

IBM Informix Dynamic Server (IDS) Enterprise Edition, Version 11.50 is certified under the Common Criteria. For more information, see *Common Criteria Certification: Requirements for IBM Informix Dynamic Server*, which is available at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss?CTY=US &FNC=SRX&PBL=SC23-7690-00.

Syntax Diagrams

Syntax diagrams use special components to describe the syntax for statements and commands.

Table 4. Syntax Diagram Components

Component represented in PDF	Component represented in HTML	Meaning
►►———————————	`>>----------------------`	Statement begins.
————————————►	`---------------------->`	Statement continues on next line.
►—————————————	`>----------------------`	Statement continues from previous line.
——————————————►◄	`----------------------><`	Statement ends.
———— SELECT ————	`--------SELECT----------`	Required item.
┌──── LOCAL ────┐	`--+----------------+---` `  '------LOCAL------'`	Optional item.

┌──── ALL ────┐
├── DISTINCT ──┤
└── UNIQUE ──┘
	`---+-----ALL-------+---`
	`   +--DISTINCT-----+`
	`   '---UNIQUE------'`
		Required item with choice. One and only one item must be present.

┌── FOR UPDATE ──┐
└─ FOR READ ONLY ─┘
	`---+-----------------+---`
	`   +--FOR UPDATE-----+`
	`   '--FOR READ ONLY--'`
		Optional items with choice are shown below the main line, one of which you might specify.

┌──── NEXT ────┐
├─── PRIOR ───┤
└── PREVIOUS ──┘
	`.---NEXT---------.`
	`----+---------------+---`
	`    +---PRIOR--------+`
	`    '---PREVIOUS-----'`
		The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.

┌─────── , ───────┐
├── index_name ──┤
└── table_name ──┘
	`.-------,-----------.`
	`V                   |`
	`---+----------------+---`
	`   +---index_name---+`
	`   '---table_name---'`
		Optional items. Several items are allowed; a comma must precede each repetition.

►►─┤ Table Reference ├─►◄	`>>-| Table Reference |-><`	Reference to a syntax segment.

Table Reference
├─── *view* ───┤
├─── *table* ───┤
└── *synonym* ──┘
	Table Reference
	`|--+-----view--------+--|`
	`   +------table------+`
	`   '----synonym------'`
		Syntax segment.

How to Read a Command-Line Syntax Diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a No-Conversion Job

```
►►──onpladm create job──job──┬────────────────┬── -n── -d──device── -D──database──────►
                             └─ -p──project──┘
```

```
►── -t──table───────────────────────────────────────────────────────────────────────►
```

```
     ┌────────────────────────────────────────────────────────┐
►────┴──┬────────────┬──┬────────────┬──┤ Setting the Run Mode ├──(1)──┴──────────────►◄
        └─ -S──server─┘  └─ -T──target─┘
```
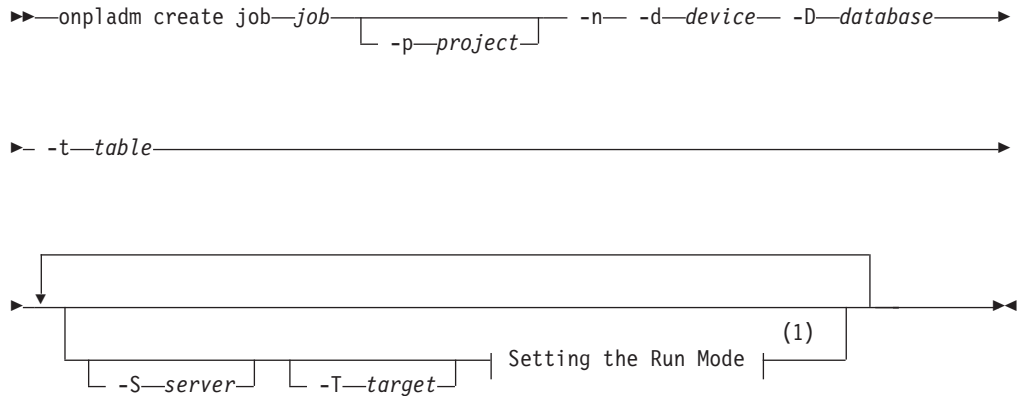
Notes:

1   See page Z-1

This diagram has a segment named "Setting the Run Mode," which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you would find this segment in on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the Run Mode:

```
                 ┌──l──┐
                 │  └c─┘│
├── -f──┬────┬───┴──u───┴──┬───┬──┬───┬──────┤
        ├─d──┤              └n──┘  └N──┘
        ├─p──┤
        └─a──┘
```
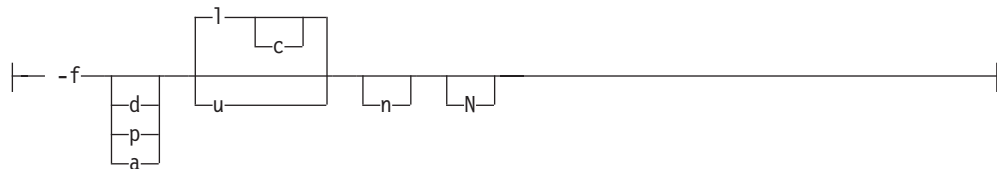
To see how to construct a command correctly, start at the top left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
    * **-n**
    * **-d** and the name of the device
    * **-D** and the name of the database
    * **-t** and the name of the table

4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and Punctuation

Keywords are words reserved for statements and all commands except system-level commands.

When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.

```
►►──SELECT──column_name──FROM──table_name─────────────────────────────►◄
```

When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:
- Send e-mail to docinf@us.ibm.com.
- Go to the information center at http://publib.boulder.ibm.com/infocenter/ idshelp/v115/index.jsp and open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.
- Add comments to topics directly in the Informix information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and

more! Find out more at http://publib.boulder.ibm.com/infocenter/idshelp/ v115/topic/com.ibm.start.doc/contributing.htm.

Feedback from all methods is monitored by those who maintain the user documentation. The feedback methods are reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support website at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Part 1. What Is IBM Informix ESQL/C

Chapter 1. Programming with IBM Informix ESQL/C

These topics contain the following information:

- What is IBM Informix ESQL/C?
- Declaring and using host variables in SQL statements
- Embedding SQL statements in C programs
- Using Informix ESQL/C preprocessor directives
- Using Informix ESQL/C header files
- Setting and retrieving Informix ESQL/C environment variables in Windows environments

The last section of these topics, "A Sample IBM Informix ESQL/C Program" on page 1-39, presents the **demo1** sample program, which is annotated. The **demo1** program illustrates the basic concepts of Informix ESQL/C programming that these topics introduce.

What Is IBM Informix ESQL/C?

IBM Informix ESQL/C is an SQL *application programming interface* (API) that enables you to embed Structured Query Language (SQL) statements directly into a C program. The Informix ESQL/C preprocessor, **esql**, converts each SQL statement and all IBM Informix-specific code to C-language source code and starts the C compiler to compile it.

ESQL/C Components

IBM Informix ESQL/C consists of the following software components:

- The Informix ESQL/C libraries of C functions, which provide access to the database server.
- The Informix ESQL/C header files, which provide definitions for the data structures, constants, and macros useful to the Informix ESQL/C program.
- The **esql** command, which processes the Informix ESQL/C source code to create a C source file that it passes to the C compiler.
- The **finderr** utility on the UNIX system and the **Informix Error Messages** Windows-based utility that enable you to obtain information about IBM Informix-specific error messages.
- The IBM Informix GLS *locale* and *code-set-conversion files*, which provide locale-specific information.

    For more information about these files, see the *IBM Informix GLS User's Guide*.

ESQL/C Files for Windows

For Windows environments, the IBM Informix ESQL/C product contains the following additional executable files:

- The **Setnet32** utility is a Windows-based utility that enables you to set configuration information.

    For more information, see the *IBM Informix Client Products Installation Guide*.

- The **ILOGIN** utility is a demonstration program that opens a dialog box with fields for the connection parameters, for testing a connection to the database server (uses the **stores7** database).

  For more information, see the *IBM Informix Client Products Installation Guide*.
- The **ESQLMF.EXE** multibyte filter changes escape characters in multibyte strings into hexadecimal literals.

These Informix ESQL/C executable files are located in the **%INFORMIXDIR%\bin**, **%INFORMIXDIR%\lib,** and **%INFORMIXDIR%\demo** directories. The **%INFORMIXDIR%** variable represents the value of the **INFORMIXDIR** environment variable.

ESQL/C Library Functions

The IBM Informix ESQL/C library contains a set of C functions that you can use in your application. The library functions are described in Appendix B, "ESQL/C function library," on page B-1. These functions fall into the following categories:

- Data type alignment library functions provide support for computer-independent size and alignment information for different data types and assist in working with null database values.
- Character and string library functions provide character-based manipulations such as comparison and copying.
- DECIMAL library functions support access to DECIMAL values through the **decimal** structure.
- Formatting functions enable you to specify display formats for different data types.
- DATE library functions support access to DATE values.
- DATETIME and INTERVAL library functions support access to values of these data types through the **datetime** and **interval** structures.
- Error message functions provide support for obtaining and formatting Informix-specific error-message text.
- Database server control functions enable your application to implement such features as canceling queries and terminating connections.
- INT8 library functions enable you to access INT8 values through the **int8** structure.
- Smart-large-object library functions provide a file-like interface to the BLOB and CLOB data types.

Creating an ESQL/C Program

You create an IBM Informix ESQL/C program with the following steps:

1. Embed Informix ESQL/C statements in a C-language source program to perform the following tasks:
   - Define host variables to store data for transfer between the Informix ESQL/C program and the database server.
   - Access the database server through SQL statements.
   - Provide directives for the Informix ESQL/C preprocessor and the C compiler.
2. Preprocess the Informix ESQL/C source file with the **esql** command to create a C-language source file and start the C compiler.
3. As necessary, correct errors reported by the preprocessor and the compiler and repeat step 2.
4. Using the **esql** command, link the compiled object code into one or more executable files.

An Informix ESQL/C source file can contain the following types of statements:

- Preprocessor directives
  - Informix ESQL/C preprocessor directives to create simple macro definitions, include Informix ESQL/C files, and perform conditional Informix ESQL/C compilation.
  - C preprocessor directives to create macro definitions, include system and C source files, and perform conditional C compilation.
- Language statements
  - Informix ESQL/C host variable definitions to store data for transfer between the Informix ESQL/C program and the database server.
  - Embedded SQL statements to communicate with the database server.
  - C language statements to provide program logic.

For information about C preprocessor directives and C language statements, see a C programming text. This chapter provides an overview of embedded SQL statements (1-4), Informix ESQL/C host variables (1-9), Informix ESQL/C preprocessor directives (1-28), and Informix ESQL/C header files (1-25).

Your Informix ESQL/C source code file name can have either of the following forms:

- **esqlc_source.**_ec_
- **esqlc_source.**_ecp_

The particular suffix that your Informix ESQL/C source file has determines the default order in which that source file gets compiled by the **esql** command. The .ec suffix is the default suffix. For more information about the .ecp suffix and the non-default order of compilation, see "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

Embedding SQL Statements

An IBM Informix ESQL/C program can use SQL statements to communicate with the database server. The program can use both _static_ and _dynamic SQL_ statements. A static SQL statement is one in which all the components are known when you compile the program. A dynamic SQL statement is one in which you do not know all the components at compile time; the program receives all or part of the statement at runtime. For a description of dynamic SQL, see Chapter 14, "Using Dynamic SQL," on page 14-1.

You can embed SQL statements in a C function with one of two formats:

- The EXEC SQL keywords:

  ```
  EXEC SQL SQL_statement;
  ```

  Using EXEC SQL keywords is the ANSI-compliant method to embed an SQL statement.
- The dollar sign ($) notation:

  ```
  $SQL_statement;
  ```

In either of these formats, replace _SQL_statement_ with the complete text of a valid statement. Informix ESQL/C statements can include host variables in most places where you can use a constant. For any exceptions, see the syntax of individual statements in the _IBM Informix Guide to SQL: Syntax_.

This section describes the following topics that are related to using embedded SQL statements in an Informix ESQL/C program:

- Case sensitivity of identifiers
- Using quotation marks and escape characters
- Commenting statements
- Host variables
- Header files

Handling Case Sensitivity in Embedded SQL Statements

The following table describes how the IBM Informix ESQL/C preprocessor treats uppercase and lowercase letters.

Table 1-1. Case Sensitivity in ESQL/C Files

ESQL/C Identifier	Case Sensitive	Example
Host variable	Yes	Informix ESQL/C treats the variables **fname** and **Fname** as distinct variables: ```EXEC SQL BEGIN DECLARE SECTION;\n char fname[16], lname[16];\n char Fname[16];\nEXEC SQL END DECLARE SECTION;``` This sample does not generate a warning from the preprocessor. For more information about host variables, see "Declaring and Using Host Variables" on page 1-9.

Variable types	Yes	Both Informix ESQL/C and C treat the names of data types as case sensitive. The CHAR type in the following example is considered distinct from the **char** data type and it generates an error:
		
		```EXEC SQL BEGIN DECLARE SECTION;\n char fname[16], lname[16];\n CHAR Fname[16];\nEXEC SQL END DECLARE SECTION;```
		
		The CHAR type does not generate an error, however, if you provide a typedef statement for it. In the following example, the **CHAR** type does not generate an error:
		
		```typedef char CHAR;```
		
		```EXEC SQL BEGIN DECLARE SECTION;\n char fname[16], lname[16];\n CHAR Fname[16];\nEXEC SQL END DECLARE SECTION;```

SQL keyword	No	Both CONNECT statements are valid ways of establishing a connection to the **stores7** demonstration database: ```EXEC SQL CONNECT TO 'stores7';\nor\nEXEC SQL connect to 'stores7';``` In examples given in this publication, SQL keywords are displayed as lowercase characters.

Table 1-1. Case Sensitivity in ESQL/C Files (continued)

ESQL/C Identifier	Case Sensitive	Example
Statement identifiers Cursor names	No	The following example shows the creation of statement IDs and cursor names:

```
EXEC SQL prepare st from
    'select * from tab1';
/* duplicate */
EXEC SQL prepare ST from
    'insert into tab2
    values (1,2)';
EXEC SQL declare curname cursor
    for st;
/* duplicate */
EXEC SQL declare CURNAME cursor
    for ST;
```

This code produces errors because the statement IDs **st** and **ST** are duplicates, as are the cursor names **curname** and CURNAME. For more information about statement IDs and cursor names, see the *IBM Informix Guide to SQL: Syntax*.

Using Quotation Marks and Escape Characters

An *escape character* indicates to the Informix ESQL/C preprocessor that it should print the following character as a literal character instead of interpreting it. You can use the escape character with an interpreted character to make the compiler *escape*, or ignore, the interpreted meaning.

In ANSI SQL, the backslash character (\) is the escape character. To search for data that begins with the string **\abc**, the WHERE clause must use an escape character as follows:

```
... where col1 = '\\abc';
```

However, ANSI standards specify that using the backslash character (\) to *escape* single (' ') or double (" ") quotation marks is invalid. For example, the following attempt to find a quotation mark does not conform to ANSI standards:

```
... where col1 = '\'';
```

In nonembedded tools such as DB-Access, you can escape a quotation mark with either of the following methods:

- You can use the same quotation mark as an escape character, as follows:

```
... where col1 = '''';
```

- You can use an alternative quotation mark. For example, to look for a double quotation mark, you can enclose this double quotation mark with quotation marks, as follows:

```
... where col1 = ' "';
```

The following figure shows a SELECT statement with a WHERE clause that contains a double quotation mark enclosed with quotation marks.

```
EXEC SQL select col1 from tab1 where col1 = ' "';
```

Figure 1-1. A SELECT Statement with an Invalid WHERE Clause

For the WHERE clause in Table 1-2, the Informix ESQL/C preprocessor does not process a double quotation mark; it passes it on to the C compiler. When the C

compiler receives the string ' " ' (double quotation mark enclosed with quotation marks), it interprets the first quotation mark as the start of a string and the double quotation mark as the end of a string. The compiler cannot match the quotation mark that remains and therefore generates an error.

To cause the C compiler to interpret the double quotation mark as a character, precede the double quotation mark with the C escape character, the backslash (\). The following example illustrates the correct syntax for the query in Table 1-2:

```
EXEC SQL select col1 from tab1 where col1 = '\"';
```

Because both C and ANSI SQL use the backslash character as the escape character, be careful when you search for the literal backslash in embedded queries. The following query shows the correct syntax to search for the string "\" (where the double quotation marks are part of the string):

```
EXEC SQL select col1 from tab1 where col1 = '\"\\\\\"';
```

This string requires five backslashes to obtain the correct interpretation. Three of the backslashes are escape characters, one for each double quotation mark and one for the backslash. Table 1-2 shows the string after it passes through each of the processing steps.

Table 1-2. Escaped Query String As It Is Processed

Processor	After Processing
ESQL/C preprocessor	'\"\\\\\"'
C compiler	'"\\"'
ANSI-compliant database server	'"\"'

Informix ESQL/C supports strings in either quotation marks ('*string*') or double quotation marks ("*string*"). However, the C language supports strings only in double quotation marks. Therefore, the Informix ESQL/C preprocessor converts every statement in an Informix ESQL/C source file into a double-quoted string.

Including Newline in Quoted Strings

IBM Informix ESQL/C does not allow a newline character (0x0A) in a quoted string. The database server does allow a newline character in a quoted string, however, if you specify that you want to allow it. Consequently, you can include the newline character in a quoted string that is part of a dynamically prepared SQL statement because the database server, rather than Informix ESQL/C, processes the prepared statement. For more information about dynamic SQL programs, see Chapter 14, "Using Dynamic SQL," on page 14-1.

You can specify that you want the database server to allow the newline character in a quoted string either on a *per session* basis or on an *all session* basis. A session is the duration of the client connection to the database server.

To allow or disallow a newline character in a quoted string for a particular session, you must execute the user-defined routine **ifx_allow_newline(boolean)**. The following example illustrates how to start the **ifx_allow_newline()** user-defined routine to allow newlines in quoted strings:

```
EXEC SQL execute procedure ifx_allow_newline('t');
```

To disallow newline in quoted strings, change the argument to f as in the following example:

```
EXEC SQL execute procedure ifx_allow_newline('f');
```

To allow or disallow a newline character in a quoted string for all sessions, set the ALLOW_NEWLINE parameter in the ONCONFIG file. A value of 1 allows the newline character. A value of 0 (zero) disallows the newline character. For more information about the ALLOW_NEWLINE parameter, see your *IBM Informix Administrator's Guide*.

Adding Comments

To add comments to an IBM Informix ESQL/C program, you can use either of the following formats:

- You can use a double dash (--) comment indicator on any Informix ESQL/C statement. The statement must begin with either EXEC SQL or $ and terminate with a semicolon. The comment continues to the end of the line.

  For example, the comment on the first of the following lines notes that the Informix ESQL/C statement opens the **stores7** demonstration database:

  ```
  EXEC SQL database stores7;   -- stores7 database is open now!
  printf("\nDatabase opened\n");  /* This is not an ESQL/C */
                                  /* line so it needs a    */
                                  /* regular C notation    */
                                   /* for a comment */
  ```

- You can use a standard C comment on an Informix ESQL/C line, as the following example shows:

  ```
  EXEC SQL begin work;  /* You can also use a C comment here */
  ```

Specifying Host Variables

Host variables are IBM Informix ESQL/C or C variables that you use in embedded SQL statements to transfer data between database columns and the Informix ESQL/C program. When you use a host variable in an SQL statement, you must precede its name with a symbol to distinguish it as a host variable. You can use either of the following symbols:

- A colon (:)

  For example, to specify the host variable that is called **hostvar** as a connection name, use the following syntax:

  ```
  EXEC SQL connect to :hostvar;
  ```

- A dollar sign ($)

  For example, to specify the host variable that is called **hostvar** as a connection name, use the following syntax:

  ```
  EXEC SQL connect to $hostvar;
  ```

Important: Using the colon (:) as a host-variable prefix conforms to ANSI SQL standards.

When you list more than one host variable within an SQL statement, separate the host variables with commas (,). For example, the **esql** command interprets the following line as two host variables, **host1** and **host2**:

```
EXEC SQL select fname, lname into :host1, :host2 from customer;
```

If you omit the comma, **esql** interprets the second variable as an indicator variable for the first host variable. The **esql** command interprets the following line as one host variable, **host1**, and one indicator variable, **host2**, for the **host1** host variable:

```
EXEC SQL select fname, lname into :host1 :host2 from customer;
```

For more information about the syntax of indicator variables, see "Using Indicator Variables" on page 1-21.

Outside an SQL statement, treat a host variable as you would a regular C variable. For more information about how to declare and use host variables, see "Declaring and Using Host Variables" on page 1-9.

Declaring and Using Host Variables

In an IBM Informix ESQL/C application, the SQL statements can refer to the contents of *host variables*. A host variable is an Informix ESQL/C program variable that you use to transfer information between the Informix ESQL/C program and the database.

You can use host variables in Informix ESQL/C expressions in the same way that you use literal values except that they cannot be used:
- In prepared statements
- In stored procedures
- In check constraints
- In views
- In triggers
- As part of a string concatenation operation

To use a host variable in an SQL statement:
1. Declare the host variable in the C program.
2. Assign a value to the host variable.
3. Specify the host variable in an embedded SQL statement.

Declaring a Host Variable

You must define the data storage that a host variable needs before you can use that variable in an IBM Informix ESQL/C program. To assign an identifier to the variable and associate it with a data type, you *declare* the variable.

You declare host variables within the Informix ESQL/C program as C variables, with the same basic syntax as C variables. To identify the variable as a host variable, you must declare it in either of the following ways:
- Put the declarations in an ESQL declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
   -- put host variable declarations here
EXEC SQL END DECLARE SECTION;
```

  Make sure you terminate the statements EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION with semicolons.
- Preface each declaration with a dollar sign ($).

Important: Using the EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION keywords conforms to ANSI standards.

Within the declaration itself, you must specify the following information:
- The name of the host variable
- The data type of the host variable
- The initial value of the host variable (optional)
- The scope of the host variable (which the placement of the declaration within the program determines)

For examples of the EXEC SQL and dollar sign ($) formats for host-variable declarations, see "Sample Host-Variable Declarations" on page 1-11.

Host-Variable Names

The name of the host variable must conform to the naming conventions of the C language. In addition, you must follow any limitations that your C compiler imposes. In general, a C variable must begin with a letter or an underscore (_) and can include letters and digits and underscores.

Warning: Many variable names used in the implementation of the IBM Informix ESQL/C product begin with an underscore. To avoid conflicting with internal Informix ESQL/C variable names, avoid using an underscore for the first character of a variable name.

C variable names are case-sensitive, so the variables **hostvar** and **HostVar** are distinct. (For more information, see "Handling Case Sensitivity in Embedded SQL Statements" on page 1-5.)

You can use non-ASCII (non-English) characters in Informix ESQL/C host-variable names if your client locale supports these non-ASCII characters. For more information about how the client locale affects host-variable names, see the *IBM Informix GLS User's Guide*.

Tip: Good programming practice requires that you create a naming convention for host-variable names.

Host-Variable Data Types

Because a host variable is a C variable, you must assign a C data type to the variable when you declare it. Likewise, when you use a host variable in an SQL statement, you also associate it with an SQL data type.

For more information about the relationship between SQL data types and C data types, see Chapter 3, "Informix ESQL/C data types," on page 3-1. In addition, the *IBM Informix Guide to SQL: Reference* contains information about the SQL data types.

You can also declare host variables as many of the more complex C data types, such as pointers, structures, **typedef** expressions, and function parameters. For more information, see "Using Host Variables in Data Structures" on page 1-16.

Initial Host-Variable Values

IBM Informix ESQL/C allows you to declare host variables with normal C initializer expressions. Some valid examples of C initializers follow:

```
EXEC SQL BEGIN DECLARE SECTION;
   int varname = 12;
   long cust_nos[8] = {0,0,0,0,0,0,0,9999};
   char descr[100] = "Steel eyelets; Nylon cording.";
EXEC SQL END DECLARE SECTION;
```

The Informix ESQL/C preprocessor does not check initializer expressions for valid C syntax; it copies them to the C source file. The C compiler diagnoses any errors.

Scope of Host Variables

The *scope of reference*, or the *scope*, of a host variable is that portion of the program in which the host variable can be accessed. The placement of the IBM Informix ESQL/C declaration statement determines the scope of the variable as follows:

- If the declaration statement is inside a program block, the variable is *local* to that program block.

  Only statements within that program block can access the variable.
- If the declaration statement is outside a program block, the variable is *modular*.

  All program blocks that occur after the declaration can access the variable.

Host variables that you declare within a block of code are local to that block. You define a block of code with a pair of curly braces ({ }).

For example, the host variable **blk_int** in the following figure is valid only in the block of code between the curly braces, whereas **p_int** is valid both inside and outside the block.

```
EXEC SQL BEGIN DECLARE SECTION;
     int p_int;
EXEC SQL END DECLARE SECTION;
 .
 .
EXEC SQL select customer_num into :p_int from customer
   where lname = "Miller";
 .
 .
{
   EXEC SQL BEGIN DECLARE SECTION;
      int blk_int;
   EXEC SQL END DECLARE SECTION;

   blk_int = p_int;

 .
 .
   EXEC SQL select customer_num into :blk_int from customer
      where lname = "Miller";
 .
 .
}
```

Figure 1-2. Declaring Host Variables Inside and Outside a Code Block

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

The following C rules govern the scope of Informix ESQL/C host variables as well:
- A host variable is an automatic variable unless you explicitly define it as an **external** or **static** variable or unless it is defined outside of any function.
- A host variable that a function declares is local to that function and masks a definition with the same name outside the function.
- You cannot define a host variable more than once in the same block of code.

Sample Host-Variable Declarations
The following figure shows an example of how to use the EXEC SQL syntax to declare host variables.

```
EXEC SQL BEGIN DECLARE SECTION;
    char   *hostvar;    /* pointer to a character */
    int    hostint;     /* integer */
    double hostdbl;     /* double */
    char   hostarr[80]; /* character array */
    struct {
        int svar1;
        int svar2;
          .
          .
          .

        } hoststruct;    /* structure */
EXEC SQL END DECLARE SECTION;
```

Figure 1-3. Declaring Host Variables with the EXEC SQL Syntax

The following figure shows an example of how to use the dollar sign ($) notation to declare host variables.

```
    $char   *hostvar;
    $int    hostint;
    $double hostdbl;
    $char   hostarr[80];

    $struct {
        int svar1;
        int svar2;
          .
          .
          .

    } hoststruct;
```

Figure 1-4. Declaring Host Variables with the Dollar Sign ($) Notation

For information about how to use a host variable in an SQL statement, see "Specifying Host Variables" on page 1-8.

Assigning a Value to a Host Variable

You can use host variables to contain the following types of information:

SQL identifiers
> SQL identifiers include names of parts of the database such as tables, columns, indexes, and views.

Data   Data is information that the database server fetches from or stores in the database. This information can include null values. A null value indicates that the value of the column or variable is unknown.

Host variables can be displayed within an SQL statement as syntax allows. (For information about the syntax of SQL statements, see the *IBM Informix Guide to SQL: Syntax*.) However, you must precede the host-variable name with a symbol to distinguish it from regular C variables. For more information, see "Specifying Host Variables" on page 1-8.

SQL Identifiers

An SQL identifier is the name of a database object. The following objects are examples of SQL identifiers:

- Parts of the database schema such as tables, columns, views, indexes, synonyms, and stored procedure names

- Dynamic IBM Informix ESQL/C structures such as cursors and statement IDs

As syntax allows, you can use a host variable within an embedded SQL statement to hold the name of an SQL identifier.

For information about the sizes and naming conventions for SQL identifiers, see the Identifier segment in the *IBM Informix Guide to SQL: Syntax*.

Using Long Identifiers: IBM Informix allows identifiers of up to 128 characters in length, and user names up to 32 characters in length. Other versions of Informix database servers support an identifier length of 18 characters and a user name length of 8 characters.

The database server uses the following two criteria to determine whether the client program can receive long identifiers:
- The internal version number of the client program
- The setting of the **IFX_LONGID** environment variable

If the **IFX_LONGID** environment variable is set to 0 (zero) the database server treats the client as if it cannot handle long identifiers. If **IFX_LONGID** is set to 1 (one) and the client version is recent enough, then the database server treats the client as if it is able to receive long identifiers. If **IFX_LONGID** is not set it is treated as if it is set to 1 (one).

Important: If you set **IFX_LONGID** in the environment of the client it will take effect only for that client. If you set the **IFX_LONGID** environment variable in the environment of the database server, it takes effect for all client programs.

For more information about the **IFX_LONGID** environment variable, see the *IBM Informix Guide to SQL: Reference*.

Client programs that meet the following conditions can use long identifiers and long user names without recompiling:
- It was compiled with a version of ESQL/C that was released later than version 9.20
- It uses shared libraries (that is, program was compiled without the `-static` option)

For more information about how to use shared libraries, see "Specifying Versions of Informix General Libraries" on page 2-28.

If the database server truncates a long identifier or long user name, it sets the SQLSTATE variable to `'01004'` and sets the **sqlwarn1** flag to `'W'` in the SQL Communications Area (**sqlca**). For more information, see Chapter 11, "Exception handling," on page 11-1.

Delimited Identifiers: If an identifier name does not conform to naming conventions, you must use a *delimited identifier*. A delimited identifier is an SQL identifier that is enclosed in double quotation marks (" ").

Important: When you use double quotation marks (" ") to delimit identifiers, you conform to ANSI standards; quotation marks (' ') delimit strings.

Use delimited identifiers when your program must specify some identifier name that would otherwise be syntactically invalid. Examples of possible invalid identifiers include:

• An identifier that is the same as an SQL reserved word.

   (For a list of SQL reserved words, see the description of identifiers in the *IBM Informix Guide to SQL: Syntax*.)

• An identifier that contains nonalphabetic characters.

To use delimited identifiers, you must compile and run your IBM Informix ESQL/C program with the **DELIMIDENT** environment variable set. You can set **DELIMIDENT** at either of the following phases:

• At compile time, the Informix ESQL/C preprocessor allows quoted strings in areas of the SQL syntax where identifiers are valid.

• At runtime, the database server accepts quoted strings in dynamic SQL statements where identifiers are valid.

   Database utilities such as **dbexport** and DB-Access also accept delimited identifiers.

Important: When you use the **DELIMIDENT** environment variable, you can no longer use double quotation marks (" ") to delimit strings. If you want to indicate a quoted string, enclose the text with quotation marks (' ').

Delimited identifiers are case-sensitive. All database object names that you place within quotation marks maintain their case. Keep in mind that Informix ESQL/C restricts identifier names to a maximum of 128 characters.

Figure 1-5 shows a delimited identifier that specifies nonalphabetic characters in both a cursor name and a statement ID.

```
EXEC SQL BEGIN DECLARE SECTION;
    char curname1[10];
    char stmtname[10];
EXEC SQL END DECLARE SECTION;

stcopy("%#!", curname1);
stcopy("(_=", stmtname);
EXEC SQL prepare :stmtname from
    'select customer_num from customer';
EXEC SQL declare :curname1 cursor for $stmtname;
EXEC SQL open :curname;
```

Figure 1-5. Using Delimited Identifiers for a Cursor Name

In Figure 1-5, you can also list the cursor name or statement ID directly in the SQL statement. For example, the following PREPARE statement is also valid (with **DELIMIDENT** set):

```
EXEC SQL prepare "%#!" from
    'select customer_num from customer';
```

If you set **DELIMIDENT**, the SELECT string in the preceding PREPARE statement must be enclosed in quotation marks for the preprocessor to treat it as a string. If you enclose the statement in double quotation marks, the preprocessor treats it as an identifier.

To declare a cursor name that contains a double quotation mark, you must use escape characters in the delimited identifier string. For example, to use the string `"abc"` as a cursor name, you must escape the initial quotation mark in the cursor name:

```
EXEC SQL BEGIN DECLARE SECTION;
   char curname2[10];
   char stmtname[10];
EXEC SQL END DECLARE SECTION;

stcopy("\"abc\"", curname2);
EXEC SQL declare :curname2 cursor for :stmtname;
```

In the preceding example, the cursor name requires several escape characters:

- The backslash (\) is the C escape character. You need it to escape the double quotation mark.

  Without the escape character, the C compiler would interpret the double quotation mark as the end of the string.

- The cursor name must contain two double quotation marks.

  The first double quotation mark escapes the double quotation mark and the second double quotation mark is the literal double quotation mark. The ANSI standard states that you cannot use a backslash to escape quotation marks. Instead, you must escape the quotation mark in the cursor name with another quotation mark.

The following table shows the string that contains the cursor name as it is processed.

Table 1-3. Escaped Cursor Name String As It Is Processed

Processor	After Processing
ESQL/C preprocessor	\"\"abc
C Compiler	""abc
ANSI-compliant database server	"abc

The following restrictions apply to delimited identifiers:

- You cannot use a delimited identifier for a database name.

  This restriction prevents the creation of a database name such as " " and avoids conflict with the syntax for IBM Informix SE database names.

- You cannot use a delimited identifier for a storage identifier, for instance, the name of a dbspace.

  The **DELIMIDENT** environment variable applies only to database identifiers.

Null values in host variables

A null value represents unknown or not applicable values. This value is distinct from all legal values in any given data type. The representation of null values depends on both the computer and the data type. Often, the representation does not correspond to a legal value for the C data type. Do not attempt to perform arithmetic or other operations on a host variable that contains a null value.

A program must, therefore, have some way to recognize a null value. To handle null values, IBM Informix ESQL/C provides the following features:

- The **risnull()** and **rsetnull()** library functions enable you to test whether a host variable contains a null value and to set a host variable to a null value.

For a description of these library functions, see Appendix B, "ESQL/C function library," on page B-1.

- Indicator variables are special Informix ESQL/C variables that you can associate with host variables that hold values for database columns that allow null values.

  The value of the indicator variable can show whether the associated host variable contains a null value. For more information, see "Using Indicator Variables" on page 1-21.

Null values in ANSI-compliant databases:

In an ANSI-compliant database, a host variable that is used in an INSERT statement or in the WHERE clause of any SQL statement must be null terminated.

Using Host Variables in Data Structures

IBM Informix ESQL/C supports the use of host variables in the following data structures:

- Arrays
- C structures (**struct**)
- C **typedef** statements
- Pointers
- Function parameters

Arrays of host variables

IBM Informix ESQL/C supports the declaration of arrays of host 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 Informix ESQL/C statements. For example, if you provide the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   long customer_nos[10];
EXEC SQL END DECLARE SECTION;
```

you can use the following syntax:

```
for (i=0; i<10; i++)
{
   EXEC SQL fetch customer_cursor into :customer_nos[i];

   .
   .
   .

}
```

You can also use the array name alone within some SQL statements if the array is of type CHAR. For information about specific statements, see the *IBM Informix Guide to SQL: Syntax*.

C structures as host variables

IBM Informix ESQL/C supports the declaration of a C structure (**struct**) as a host variable. You can use the components of the structure within Informix ESQL/C statements.

The following definition of the **cust_rec** variable serves as a host variable for the first three columns of the **customer** table in the **stores7** database:

```
EXEC SQL BEGIN DECLARE SECTION;
   struct customer_t
      {
      int   c_no;
      char  fname[32];
      char  lname[32];
      } cust_rec;
EXEC SQL END DECLARE SECTION;
```

The following INSERT statement specifies the components of the **cust_rec** host variable in its VALUES clause:

```
EXEC SQL insert into customer (customer_num, fname, lname)
   values (:cust_rec.c_no, :cust_rec.fname,
   :cust_rec.lname);
```

If an SQL statement requires a single host variable, you must use the structure component name to specify the host variable. Informix requires structure component names in the SET clause of an UPDATE statement.

In SQL statements that allow a list of host variables, you can specify the name of the C structure and Informix ESQL/C expands the name of the structure variable to each of its component elements. You can use this syntax with SQL statements such as the FETCH statement with an INTO clause or the INSERT statement with a VALUES clause.

The following INSERT statement specifies the entire **cust_rec** structure in its VALUES clause:

```
EXEC SQL insert into customer (customer_num, fname, lname)
   values (:cust_rec);
```

This insert performs the same task as the insert that specifies the individual component names of the **cust_rec** structure.

C typedef statements as host variables

IBM Informix ESQL/C supports the C **typedef** statements and allows the use of **typedef** names in declaring the types of 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 an array of **serial** variables and a variable **counter** as a **smallint**.

```
EXEC SQL BEGIN DECLARE SECTION;
   typedef short smallint;
   typedef long serial;

   serial row_nums [MAXROWS];
   smallint counter;
EXEC SQL END DECLARE SECTION;
```

You cannot use a **typedef** statement that names a multidimensional array, or a union, or a function pointer, as the type of a host variable.

Pointers as host variables

You can use a pointer as a host variable provided that your program uses the pointer to input data to an SQL statement. For example, the following figure shows how you can associate a cursor with a statement and insert values into a table.

```
EXEC SQL BEGIN DECLARE SECTION;
   char *s;
   char *i;
EXEC SQL END DECLARE SECTION;

/* Code to allocate space for two pointers not shown */

s = "select * from cust_calls";
i = "NS";

   .
   .
   .

EXEC SQL prepare x from :s;
EXEC SQL insert into state values (:i, 'New State');
```

Figure 1-6. Declaring a character pointer to input data

The following figure shows how to use an integer pointer to input data to an
INSERT statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   short *i;
   int *o;
   short *s;
EXEC SQL END DECLARE SECTION;

short i_num = 3;
int o_num = 1002;
short s_num = 8;

i = &i_num;
o = &o_num;
s = &s_num;

EXEC SQL connect to 'stores7';
EXEC SQL insert into items values (:*i, :*o, :*s, 'ANZ', 5, 125.00);
EXEC SQL disconnect current;
```

Figure 1-7. Declaring an integer pointer to input data

If you use a host variable that is a pointer to **char** to receive data from a SELECT
statement, you receive a compile-time warning and your results might be
truncated.

Function parameters as host variables

You can use host variables as parameters to functions. You must precede the name
of the host variable with the **parameter** keyword to declare it as a function
parameter. For example, the following figure shows a code fragment with a
Kernighan and Ritchie-style prototype declaration that expects three parameters,
two of which are host variables.

```
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;
{
    select fname into :s from customer
        where customer_num = :id;
  .
  .
  .

}
```

Figure 1-8. Using EXEC SQL to declare host variables as parameters to a Kernighan and Ritchie-Style function declaration

You can also declare parameter host variables with the dollar sign ($) notation. For example, Figure 1-9 shows the function header in Figure 1-8, with the dollar sign ($) notation.

```
f(s, id, s_size)
$parameter char s[20];
$parameter int id;
int s_size;
```

Figure 1-9. Using the Dollar Sign ($) to Declare Host Variables as Parameters to a Function

You can declare parameters in an ANSI-style prototype function declaration as host variables as well. You can also put all parameters to a prototype function declaration inside the EXEC SQL declare section, even if some of the parameters cannot be used as host variables. Figure 1-10 on page 1-19 shows that the function pointer **f** can be included in the EXEC SQL declare section, even though it is not a valid host-variable type and cannot be used as a host variable.

```
int * foo(
EXEC SQL BEGIN DECLARE SECTION;
    PARAMETER char s[20],
    PARAMETER int id,
    PARAMETER int (*f) (double)
EXEC SQL END DECLARE SECTION;
)
{
    select fname into :s from customer
        where customer_num = :id;
  .
  .
  .

}
```

Figure 1-10. Using EXEC SQL to Declare Host Variables as Parameters to ANSI-Style Function Declaration

The functionality that allows inclusion of function parameters inside of the EXEC SQL declare section is in compliance with the requirement that any valid C declaration syntax must be allowed inside the EXEC SQL declare sections to use common header files for C and Informix ESQL/C source files. For more information about how to use common header files between C and Informix ESQL/C source files, see "Defining Host Variables Based on C #defines and typedefs" on page 2-18.

Important: If you want to define Informix ESQL/C host variables that are ANSI-style parameters, you must use the EXEC SQL BEGIN DECLARE SECTION and the EXEC SQL END DECLARE SECTION syntax. You cannot use the $BEGIN DECLARE and $END DECLARE syntax. This restriction is because SQL statements that begin with the dollar sign ($) notion must end with a semicolon (;). However, ANSI syntax requires that each parameter in a parameter list should not end with a semicolon terminator, but with a comma (,) delimiter.

The following limitations apply to using host variables as function parameters:

- 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. If you do, you receive unpredictable results.

Using Host Variables in Windows Environments

This section describes the following topics about IBM Informix ESQL/C host variables that are unique to the Windows environments:

- How to declare host variables with non-ANSI storage-class modifiers
- How global Informix ESQL/C variables are declared

Declaring Variables with Non-ANSI Storage-Class Modifiers

The ANSI C standards define a set of storage-class specifiers for variable declarations. C compilers in Windows environments often support non-ANSI storage-class specifiers. To provide support for these non-ANSI storage-class specifiers in IBM Informix ESQL/C host-variable declarations, the Informix ESQL/C preprocessor supports the form of the ANSI syntax that the following figure shows.
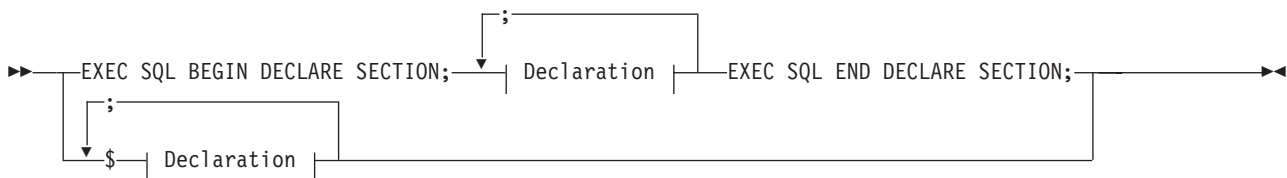
Declaration:
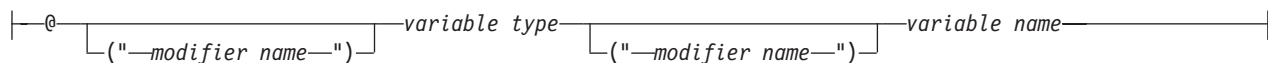
Figure 1-11. ESQL/C Syntax for Non-ANSI Storage-Class Specifiers

Element	Purpose	Restrictions	Syntax
modifier name	Text that you want to pass to the C compiler for translation. This text is usually the name of the storage-class modifier.	The modifier must be valid for your C compiler or be a name that you define in your program.	See your C compiler documentation.
variable name	Identifier name of the ESQL/C host variable	None.	See "Declaring a Host Variable" on page 1-9.
variable type	Data type of the ESQL/C host variable	The type must be a valid C or ESQL/C data type.	See "Declaring a Host Variable" on page 1-9.

For example, the Microsoft Visual C++ compiler supports the **declspec** compiler directive to enable you to declare extended storage-class attributes. This compiler directive has the following syntax:

```
__declspec(attribute) var_type var_name;
```

In this example, *attribute* is a supported keyword (such as **thread**, **dllimport**, or **dllexport**), *var_type* is the data type of the variable, and *var_name* is the variable name.

To enable you to declare Informix ESQL/C host variables as extended storage-class variables, the Informix ESQL/C preprocessor supports the **declspec** directive with the following syntax:

```
@("__declspec(attribute)") var_type var_name;
```

In this example, *attribute*, *var_type*, and *var_name* are the same as in the previous example. You might find it convenient to declare a macro for the **declspec** syntax. The following example declares **threadCount** as an instance-specific integer variable of the thread-extended storage class:

```
#define DLLTHREAD __declspec(thread)
  :
  :

EXEC SQL BEGIN DECLARE SECTION;
   @("DLLTHREAD") int threadCount;
EXEC SQL END DECLARE SECTION;
```

This example creates the DLLTHREAD macro to simplify the declaration of thread-extended storage-class attributes. You can declare similar macros to simplify declaration of variables to be exported (or imported) to the dynamic link library (DLL), as follows:

```
#define DLLEXPORT __declspec(dllexport);
  :
  :

EXEC SQL BEGIN DECLARE SECTION;
   @("DLLEXPORT") int winHdl;
EXEC SQL END DECLARE SECTION;
```

Using Indicator Variables

When an SQL statement returns a value, it returns it in the host variable for the specified column. In some cases, you can associate an *indicator variable* with the host variable to obtain additional information about the value that is returned. If you specify an indicator variable, IBM Informix ESQL/C sets it in addition to returning the value to the host variable.

The indicator variable provides additional information in the following situations:
- If the host variable is associated with a database column or an aggregate function that allows null values, the indicator variable can specify whether the value is null.
- If the host variable is a character array and the column value is truncated by the transfer, the indicator variable can specify the size of the returned value.

The following sections describe how to declare an indicator variable and associate it with a host variable, and also how Informix ESQL/C sets an indicator variable to specify the two preceding conditions.

Declaring Indicator Variables

You declare indicator variables in the same way as host variables, between BEGIN DECLARE SECTION and END DECLARE SECTION statements as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
   -- put indicator variable declarations here
EXEC SQL END DECLARE SECTION;
```

For more information, see "Declaring a Host Variable" on page 1-9.

Indicator variables can be any valid host-variable data type except DATETIME or INTERVAL. Usually, you declare an indicator variable as an integer. For example, suppose your program declares a host variable called **name**. You can declare a **short** integer-indicator variable called **nameind**, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char  name[16];
   short nameind;
EXEC SQL END DECLARE SECTION;
```

You can use non-ASCII (non-English) characters in Informix ESQL/C indicator-variable names if your client locale supports these non-ASCII characters. For more information about how the client locale affects host-variable names, see the *IBM Informix GLS User's Guide*.

Associating an Indicator Variable with a Host Variable

You associate an indicator variable with its host variable in one of the following two ways:
- Prefix the indicator variable with a colon (:) and place the keyword INDICATOR between the host variable name and the indicator variable name as follows:

  ```
  :hostvar INDICATOR :indvar
  ```
- Place a separator symbol between the host variable name and the indicator variable name. The following separator symbols are valid:
  - A colon (:)

  ```
  :hostvar:indvar
  ```
- A dollar sign ($)

  ```
  $hostvar$indvar
  ```

You can use a dollar sign ($) instead of a colon (:), but the colon makes the code easier to read.

You can have one or more white space characters between the host variable and indicator variable. For example, both of the following formats are valid to specify an indicator variable, **hostvarind,** on the **hostvar** host variable:

```
$hostvar:hostvarind
$hostvar :hostvarind
```

Indicating Null Values

When an IBM Informix ESQL/C statement returns a null value to a host variable, the value might not be a meaningful C value. Your program can take one of the following actions:

- If you have defined an indicator variable for this host variable, Informix ESQL/C sets the indicator variable to -1.

  Your program can check the indicator variable for a value of -1.
- If you did not define an indicator variable, the runtime behavior of Informix ESQL/C depends on how you compiled the program:
  - If you compile the program with the **-icheck** preprocessor option, Informix ESQL/C generates an error and sets **sqlca.sqlcode** to a negative value when the database server returns a null value. (See "Syntax of the esql command" on page 2-5.)
  - If you compile the program without the **-icheck** option, Informix ESQL/C does not generate an error when the database server returns a null value. In this case, you can use the **risnull()** function to test the host variable for a null value.

If the value returned to the host variable is not null, Informix ESQL/C sets the indicator variable to 0. If the SQL operation is not successful, the value of the indicator variable is not meaningful. Therefore, you should check the outcome of the SQL statement before you check for a null value in the host variable. For information about exception handling, see Chapter 11, "Exception handling," on page 11-1.

The NULL keyword of an INSERT statement allows you to insert a null value into a table row. As an alternative to the NULL keyword in an INSERT statement, you can use a negative indicator variable with the host variable.

If you want to insert a variable while the indicator is set to NULL (-1), the indicator value will take precedence over the variable value. The value inserted in this case will NULL instead of the value of the host variable.

When you return aggregate function values into a host variable, keep in mind that when the database server performs an aggregate function on an empty table, the result of the aggregate operation is the null value. The only exception to this rule is the **COUNT(*)** aggregate function, which returns a zero (0) in this case.

Important: If you activate the DATASKIP feature of the database server, an aggregate function also returns null if all fragments are offline or if all the fragments that are online are empty.

The DATASKIP feature is unavailable for IBM Informix SE.

Indicating truncated values

When an SQL statement returns a non-null value into a host-variable character array, it might truncate the value to fit into the variable. If you define an indicator variable for this host variable, IBM Informix ESQL/C:

- Sets the SQLSTATE variable to "01004" to signal the occurrence of truncation.

  (For more information about SQLSTATE, see "List of SQLSTATE Class Codes" on page 11-8.) Informix ESQL/C also sets **sqlwarn1** of the **sqlca.sqlwarn** structure to W.

- Sets the associated indicator variable equal to the size in bytes of the SQL host variable before truncation.

If you do not define an indicator variable, Informix ESQL/C still sets SQLSTATE and **sqlca.sqlwarn** to signal the truncation. However, your program has no way to determine how much data was truncated.

If the database server returns a value that is neither truncated nor null, Informix ESQL/C sets the indicator variable to 0.

An indicator variable shows nonzero values for all columns retrieved from a DB2® database, even if only one of the columns was truncated.

An Example Using Indicator Variables

The code segments in Figure 1-12 and Figure 1-13 on page 1-25 show examples of how to use indicator variables with host variables. Both examples use indicator variables to perform the following tasks:

- Determine if truncation has occurred on a character array

  If you define **lname** in the **customer** table with a length that is 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 surname of the company representative with **customer_num = 105** is shorter than 15 characters, Informix ESQL/C truncates only the trailing blanks.

- Check for a null value

  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.

The following figure shows an Informix ESQL/C program that uses the EXEC SQL syntax for the SQL statements.

```
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 INDICATOR :nameind, :comp INDICATOR :compind
   from customer
   where customer_num = 105;
```

Figure 1-12. Using Indicator Variables with EXEC SQL and the Colon (:) Symbol

Figure 1-12 uses the INDICATOR keyword to associate the main and indicator variables. This method complies with the ANSI standard.

The following figure shows an Informix ESQL/C program that uses the dollar sign ($) format for the SQL statements.

```
$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 1-13. Using Indicator Variables with the Dollar Sign ($) Notation

Using ESQL/C Header Files

When you install IBM Informix ESQL/C, the installation script stores the header files in the **$INFORMIXDIR/incl/esql** directory on a UNIX operating system and in the **%INFORMIXDIR%\incl\esql** directory in a Windows environment.

The following table shows the header files provided with the Informix ESQL/C product.

Table 1-4. ESQL/C Header Files

Header File	Contains	Additional Information
datetime.h	Definitions of the Informix ESQL/C **datetime** and **interval** structures, which are the host variables for DATETIME and INTERVAL columns	Chapter 6
decimal.h	Definition of the Informix ESQL/C **decimal** data type, which is the host variable for DECIMAL and MONEY data types	Chapter 5
gls.h	Function prototypes and data structures for the GLS functionality	*IBM Informix GLS User's Guide*
ifxtypes.h	Correctly maps the Informix data types **int1**, **int2**, **int4**, **mint**, **mlong**, **MSHORT**, and **MCHAR** for 32-bit and 64-bit platforms.	"The Integer Host Variable Types" on page 5-2
locator.h	Definition of the Informix ESQL/C locator structure (**ifx_loc_t** or **loc_t**), which is the host variable for byte and text columns	Chapter 7
sqlca.h	Definition of the structure that ESQL/C uses to store error-status codes. The **esql** preprocessor automatically includes this file when it preprocesses your program.	Chapter 11
sqlda.h	Structure definition for value pointers and descriptions of dynamically defined variables	Chapter 15
sqlhdr.h	This file includes the sqlda.h header file, other header files, and function prototypes. The preprocessor automatically includes this file when it preprocesses your program.	"Including Header Files" on page 1-27
sqliapi.h	Function prototypes for internal library APIs. For internal Informix ESQL/C use only.	None

Table 1-4. ESQL/C Header Files (continued)

Header File	Contains	Additional Information
sqlstype.h	Definitions of constants for SQL statements. The DESCRIBE statement uses these constants to describe a dynamically prepared SQL statement.	Chapter 15
sqltypes.h	Defines constants that correspond to ESQL/C and SQL data types. ESQL/C uses these constants when your program contains a DESCRIBE statement.	"Data Type Constants" on page 3-3
sqlxtype.h	Defines constants that correspond to Informix ESQL/C and SQL data types when you are in X/Open mode ESQL/C uses these constants when your program contains a DESCRIBE statement.	"X/Open Data Type Constants" on page 3-6
value.h	Value structures that Informix ESQL/C uses *For internal Informix ESQL/C use only*.	None
varchar.h	Macros that you can use with the VARCHAR data type	Chapter 4

The following figure shows the Informix ESQL/C header files specific to IBM Informix.

Table 1-5. ESQL/C Header Files for IBM Informix

Header File	Contents	Additional Information
collct.h	Definitions of data structures for complex types in Informix ESQL/C	Chapter 9
ifxgls.h	Function prototypes for the GLS application programming interface For internal Informix ESQL/C use only.	None
int8.h	Definition of the structure that stores the INT8 data type	"The int8 Data Type" on page 5-2

The following table shows the Informix ESQL/C header files specific to Windows environments.

Table 1-6. ESQL/C Header Files for Windows Environments

Header File	Contents	Additional Information
sqlproto.h	Function prototypes of all ESQL/C library functions for use with source that is *not* fully ANSI C compliant	"Declaring Function Prototypes" on page 1-27
infxcexp.c	Contains the C code to export the addresses of all C runtime routines that the ESQL client-interface DLL uses	"Using the Same Runtime Routines for Version Independence" on page 2-29
login.h	The definition of the InetLogin and **HostInfoStruct** structures, which enable you to customize configuration information for the application Because this file does not contain ESQL statements, you do not need to include it with the ESQL **include** directive. Use instead the C **#include** preprocessor directive.	"Fields of the InetLogin Structure" on page 1-33

Declaring Function Prototypes

IBM Informix ESQL/C provides the **sqlproto.h** header file to declare function prototypes for all Informix ESQL/C library functions. These function prototypes are required in an Informix ESQL/C source file that you compile with an ANSI C compiler. By default, the **esql** command processor does not include function-prototype declarations. Having the processor include the ANSI-compliant function prototypes for the Informix ESQL/C functions prevents an ANSI C compiler from generating warnings.

Warning: Although you can use an ANSI C compiler, the Informix ESQL/C preprocessor does not fully support ANSI C, so you might not be able to preprocess all programs that follow the ANSI C standards.

Because the **sqlproto.h** file does not contain any Informix ESQL/C statements, you can include this file in either of the following ways:

- With the Informix ESQL/C **include** preprocessor directive:

  ```
  EXEC SQL include sqlproto;
  ```
- With the C **#include** preprocessor directive:

  ```
  #include "sqlproto.h";
  ```

Including Header Files

The Informix ESQL/C preprocessor *automatically* includes the following Informix ESQL/C header files in your program:

- The **sqlhdr.h** file provides cursor-related structures for your Informix ESQL/C program.

  This header file automatically includes the **sqlda.h** and **ifx_types.h** header files.
- The **sqlca.h** file, which allows your program to check the success or failure of your Informix ESQL/C statements with the SQLSTATE or SQLCODE variable

Warning: Although you can now use an ANSI C compiler, the Informix ESQL/C preprocessor does not fully support ANSI C, so you might not be able to preprocess all programs that follow the ANSI C standards.

To include any of the other header files in your Informix ESQL/C program, you must use the **include** preprocessor directive. However, you only need to include an Informix ESQL/C header file if your program refers to the structures or the definitions that the header file defines. For example, if your program accesses datetime data, you *must* include the **datetime.h** header file, as follows:

```
EXEC SQL include datetime.h;
```

Make sure to terminate the line of code with a semicolon. Some additional examples follow:

```
EXEC SQL include varchar.h;
EXEC SQL include sqlda;
$include sqlstype;
```

Tip: You do not have to enter the **.h** file extension for an *Informix ESQL/C* header file; the **esql** preprocessor assumes a **.h** extension.

For information about the **include** directive, see "The include Directive" on page 1-29.

Using ESQL/C Preprocessor Directives

You can use the following capabilities of the Informix ESQL/C preprocessor when you write Informix ESQL/C code:

- The **include** directive expands Informix ESQL/C include files within your program.
- The **define** and **undef** directives create compile-time definitions.
- The **ifdef**, **ifndef**, **else**, **elif**, and **endif** directives specify conditional compilation.

As with embedded SQL statements, you can use either of two formats for Informix ESQL/C preprocessor directives:

- The EXEC SQL keywords:

  ```
  EXEC SQL preprocessor_directive;
  ```

- The dollar sign ($) notation:

  ```
  $preprocessor_directive;
  ```

Important: The EXEC SQL keywords conform to ANSI standards.

In either of these formats, replace *preprocessor_directive* with one of the valid preprocessor directives that the following sections describe. You must terminate these directives with a semicolon (;).

The Informix ESQL/C preprocessor works in two stages. In stage 1, it acts as a preprocessor for the Informix ESQL/C code. In stage 2, it converts all of the embedded SQL code to C code.

In stage 1, the Informix ESQL/C preprocessor incorporates other files in the source file by processing all **include** directives (**$include** and EXEC SQL **include** statements). Also in stage 1, Informix ESQL/C creates or removes compile-time definitions by processing all **define** (**$define** and EXEC SQL **define**) and **undef** (**$undef** and EXEC SQL **undef**) directives.

The remainder of this section describes each of the Informix ESQL/C preprocessor directives in more detail.

The include Directive

The **include** directive allows you to specify a file to *include* within your Informix ESQL/C program. The Informix ESQL/C preprocessor places the contents of the specified file into the Informix ESQL/C source file. Stage 1 of the Informix ESQL/C preprocessor reads the contents of *filename* into the current file at the position of the **include** directive.

You can use the **include** preprocessor directive in either of the following two formats:
- **EXEC SQL include** *filename*;
- **$include** *filename*;

Replace *filename* with the name of the file you want to include in your program. You can specify *filename* with or without quotation marks. If you use a full path name, however, you *must* enclose the path name in quotation marks.

The following example shows how to use full path names in a Windows environment.

```
EXEC SQL include decimal.h;

EXEC SQL include "C:\apps\finances\credits.h";
```

Tip: If you specify the full path name, you must recompile the program if the location of the file changes. Better programming practice specifies search locations with the **esql -I** option and specifies only the file name with the include directive.

If you omit the quotation marks around the file name, Informix ESQL/C changes the file name to lowercase characters. If you omit the path name, the Informix ESQL/C preprocessor checks the preprocessor search path for the file. For more information about this search path, see "Naming the Location of Include Files" on page 2-14.

You can use **include** for the following types of files:
- An Informix ESQL/C header file

  You do not have to use the **.h** file extension for an Informix ESQL/C header file; the compiler assumes that your program refers to a file with a **.h** extension. The following examples show valid statements to include Informix ESQL/C header files:

  ```
  EXEC SQL include varchar.h;
  $include sqlda;
  EXEC SQL include sqlstype;
  ```

  For a complete list of Informix ESQL/C header files, see "Using ESQL/C Header Files" on page 1-25.
- Other user-defined files

  You must specify the exact name of the file that you want to include. The compiler does not assume the **.h** extension when you include a header file that is not an Informix ESQL/C header file.

  The following examples show valid statements to include the files **constant_defs** and **typedefs.h** in a UNIX environment:

  ```
  EXEC SQL include constant_defs;
  EXEC SQL include "constant_defs";
  $include typedefs.h;
  EXEC SQL include "typedefs.h";
  ```

You must use the Informix ESQL/C **include** directive if the file you specify contains embedded SQL statements, or other Informix ESQL/C statements.

Use the standard C **#include** directive to include system header files. The **#include** of C includes a file after Informix ESQL/C preprocessing.

Attention: Embedded INCLUDE statements are not supported within declare sections and can generate misleading errors. For correct usage, see "Excluding Statements Inside C Header Files" on page 2-20.

The define and undef Directives

The IBM Informix ESQL/C preprocessor allows you to create simple variables that are available *only* to the Informix ESQL/C preprocessor. Informix calls these variables *definitions. The* Informix ESQL/C preprocessor manages these definitions with two directives:

define   creates a name-flag definition. The scope of this definition is from the point where you define it to the end of the Informix ESQL/C source file.

undef   removes a name flag that EXEC SQL **define** or **$define** creates.

The Informix ESQL/C preprocessor rather than the C preprocessor (which processes **#define** and **#undef**) processes these directives. The Informix ESQL/C preprocessor creates (**define**) or removes (**undef**) these definitions in stage 1 of preprocessing.

The Informix ESQL/C **define** directive can create definitions with the following formats:

- The format for Boolean symbols is

  ```
  define symbolname;
  ```

  The following examples show the two ways to define a Boolean symbol that is called TRANS:

  ```
  EXEC SQL define TRANS;
  $define TRANS;
  ```

- The format for integer constants is

  ```
  define symbolname value;
  ```

  The following examples show both formats for two integer constants, MAXROWS (with a value of 25), and USETRANSACTIONS (with a value of 1):

  ```
  EXEC SQL define MAXROWS 25;
  $define MAXROWS 25;

  EXEC SQL define USETRANSACTIONS 1;
  $define USETRANSACTIONS 1;
  ```

Important: Unlike the C **#define** statement, the **define** directive does *not* support string constants or macros of statements that receive values at runtime.

You can override **define** and **undef** statements in the source program with the **esql** command line options, **-ED** and **-EU**. For more information about these options, see "Defining and Undefining Definitions While Preprocessing" on page 2-13.

The ifdef, ifndef, elif, else, and endif Directives

The Informix ESQL/C preprocessor supports the following directives for conditional compilation:

ifdef	tests a name and executes subsequent statements if define has created the name.
ifndef	tests a name and executes subsequent statements if define has *not* created the name.
elif	Begins an alternative section to an **ifdef** or **ifndef** condition and checks for the presence of another **define**. It is shorthand for "else if define ".
else	Begins an alternative section to an **ifdef** or **ifndef** condition.
endif	Closes an **ifdef** or **ifndef** condition.

In the following example, the BEGIN WORK statement only compiles if you previously defined the name USETRANSACTIONS with a **define** directive:

```
EXEC SQL ifdef USETRANSACTIONS;
EXEC SQL begin work;
EXEC SQL endif;
```

The following example illustrates the use of the **elif** statement. This sample code will print "USETRANSACTIONS defined".

```
EXEC SQL define USETRANSACTIONS;

    ⋮

   EXEC SQL ifndef USETRANSACTIONS;
      printf("USETRANSACTIONS not defined");
   EXEC SQL elif USETRANSACTIONS;
      printf("USETRANSACTIONS defined");
   EXEC SQL endif;
```

The Informix ESQL/C preprocessor does not support a general **if** directive; it supports only the **ifdef** and **ifndef** statements that test whether a name was defined with **define**.

The Informix ESQL/C preprocessor processes conditional compilation definitions in stage 1 of the preprocessing.

Setting and Retrieving Environment Variables in Windows Environments

You might change the settings of environment variables or create new variables to increase the flexibility of an application. Instead of assuming a particular environment configuration, you can define the environment at runtime. This option can benefit your application in the following ways:

- The application becomes less dependent on a predefined environment.
- Users can enter their user name and password within an application.
- Users can run two applications with different network parameters on the same client computer.
- The same application can run on client computers with different configurations.

The following Informix ESQL/C library functions are available for setting and retrieving environment variables. The library functions are located in Appendix B, "ESQL/C function library," on page B-1.

Function Name	Description	See
ifx_putenv()	Modifies or removes an existing environment variable or creates a new variable	B-146
ifx_getenv()	Retrieves the value of an environment variable	B-71

Important: The **ifx_putenv()** function sets the value of an environment variable in the **InetLogin** structure, and the **ifx_getenv()** function retrieves the value of an environment variable from **InetLogin**. It is recommended that you use these functions to set and retrieve **InetLogin** field values.

For more information about **InetLogin** fields, see "Fields of the InetLogin Structure" on page 1-33.

These functions affect only the environment that is local to the current process. The **ifx_putenv()** function cannot modify the command-level environment. The functions operate only on data structures accessible to the Informix ESQL/C runtime library and not on the environment segment that the operating system creates for the process. When the current process terminates, the environment reverts to the level of the calling process (in most cases, the operating-system level).

The process cannot directly pass on the modified environment to any new processes that **_spawn()**, **_exec()**, or **system()** creates. These new processes do not receive any new variables that **ifx_putenv()** added. You can, however, pass on an environment variable to a new process in the following way:

1. The current process creates an environment variable with the Informix ESQL/C **ifx_putenv()** function.
2. The current process uses the C **putenv()** function to put the environment variable into the operating-system environment segment.
3. The current process starts a new process.
4. The new process uses the C **getenv()** function to retrieve the environment variable from the operating-system environment segment.
5. The new process uses the Informix ESQL/C **ifx_getenv()** function to retrieve the variable into the runtime environment segment.

For environment variable entries, observe the following guidelines:

- If you plan to set any Informix environment variables with **ifx_putenv()**, have the application set all of them before it calls any other Informix ESQL/C library routine, including **ifx_getenv()**, or any SQL statement. The first call to any other Informix ESQL/C library routine or SQL statement requires initialization of the GLS locales. This initialization loads and freezes the values of **CLIENT_LOCALE**, **DB_LOCALE**, and the **DATE**, **TIME**, and **DATETIME** formatting values.

- If **Setnet32** sets an Informix environment variable to a non-null value in the Registry, the **ifx_putenv()** function cannot change the value of the variable to a null string.

  If you specify a null string for an environment variable in an **ifx_putenv()** function call, Informix ESQL/C clears any value set for the environment variable from the runtime environment segment. Then the Registry value for the environment variable is available to the application.

- Do not change an environment variable with **setenv** in the command line or with the C **putenv()** function because a change to the operating-system environment segment has no effect on the ESQL client-interface DLL after application execution begins.

  Instead, use **ifx_putenv()** to change an environment variable in the runtime environment segment.

- To modify the return value of **ifx_getenv()** without affecting the environment table, use **_strdup()** or **strcpy()** to make a copy of the string.

Warning: Never free the pointer to an environment entry that **ifx_getenv()** returns. Also, do not pass **ifx_putenv()** a pointer to a local variable and then exit the function that declares the variable.

InetLogin structure

Important: IBM Informix supports the **InetLogin** structure for compatibility with earlier versions only. For new development, it is recommended that you use the **ifx_getenv()** and **ifx_putenv()** functions instead.

An Informix ESQL/C client application in a Windows environment can use the **InetLogin** structure to set dynamically the configuration information that the application needs.

This section provides the following information about **InetLogin**:
- A description of the **InetLogin** structure, its fields, and header file
- The precedence of configuration information that the client application sends when it establishes a connection
- How to set the **InetLogin** fields directly

Fields of the InetLogin Structure
The **InetLogin** structure is a global C structure that the **login.h** header file declares. To use this structure in your IBM Informix ESQL/C program, you must include **login.h** in your source file (**.ec**). For more information about **login.h**, see Table 1-6 on page 1-27.

Tip: Because **login.h** does not contain Informix ESQL/C statements, you can include the file with the C **#include** or the Informix ESQL/C **include** directive.

The following table defines the fields in the **InetLogin** structure.

Table 1-7. Fields of the InetLogin structure

InetLogin Field	Data Type	Purpose
InfxServer	char[19]	Specifies the value for the **INFORMIXSERVER** environment variable (the default database server)
DbPath	char[129]	Specifies the value for the **DBPATH** environment variable
DbDate	char[6]	Specifies the value for the **DBDATE** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix products
DbMoney	char[19]	Specifies the value for the **DBMONEY** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix products

Table 1-7. Fields of the InetLogin structure (continued)

InetLogin Field	Data Type	Purpose
DbTime	char[81]	Specifies the value for the **DBTIME** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix products
DbTemp	char[81]	Specifies the value for the **DBTEMP** environment variable
DbLang	char[19]	Specifies the value for the **DBLANG** environment variable
DbAnsiWarn	char[1]	Specifies the value for the **DBANSIWARN** environment variable
InformixDir	char[255]	Specifies the value for the **INFORMIXDIR** environment variable
Client_Loc	char *	Specifies the value for the **CLIENT_LOCALE** environment variable
DB_Loc	char *	Specifies the value for the **DB_LOCALE** environment variable
CollChar	char[3]	Specifies the value for the **COLLCHAR** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
Lang	char[81]	Specifies the value for the **LANG** environment variable for the database locale Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
Lc_Collate	char[81]	Specifies the value for the **LC_COLLATE** environment variable for the database locale Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
Lc_CType	char[81]	Specifies the value of the **LC_CTYPE** environment variable for the database locale Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
Lc_Monetary	char[81]	Specifies the value of the **LC_MONETARY** environment variable for the database locale Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
Lc_Numeric	char[81]	Specifies the value of the **LC_NUMERIC** environment variable for the database locale Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
Lc_Time	char[81]	Specifies the value for the **LC_TIME** environment variable for the database locale Provides compatibility for client applications that are based on earlier versions of IBM Informix NLS products
ConRetry	char[4]	Specifies the value of the environment variable **INFORMIXCONRETRY**
ConTime	char[4]	Specifies the value of the environment variable **INFORMIXCONTIME**
DelimIdent	char[4]	Specifies the value of the **DELIMIDENT** environment variable
Host	char[19]	Specifies the value for the HOST network parameter
User	char[19]	Specifies the value for the USER network parameter
Pass	char[19]	Specifies the value for the PASSWORD network parameter

Table 1-7. Fields of the InetLogin structure (continued)

InetLogin Field	Data Type	Purpose
AskPassAtConnect	char[2]	Indicates whether **sqlauth()** should request a password at connection time; should contain the value for yes or no. **AskPassAtConnect** is set if the first character is Y or y.
Service	char[19]	Specifies the value for the SERVICE network parameter
Protocol	char[19]	Specifies the value for the PROTOCOL network parameter
Options	char[20]	*Reserved for future use*
InformixSqlHosts	char[255]	Specifies the value for the **INFORMIXSQLHOSTS** environment variable
FetBuffSize	char[6]	Specifies the value for the **FET_BUF_SIZE** environment variable
CC8BitLevel	char[2]	Specifies the value for the **CC8BITLEVEL** environment variable
EsqlMF	char[2]	Specifies the value for the **ESQLMF** environment variable
GlDate	char[129]	Specifies the value for the **GL_DATE** environment variable
GlDateTime	char[129]	Specifies the value for the **GL_DATETIME** environment variable
DbAlsBc	char[2]	Specifies the value for the **DBALSBC** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix products |
| DbApiCode | char[24] | Specifies the value for the **DBAPICODE** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix products |
| DbAsciiBc | char[2] | Specifies the value for the **DBASCIIBC** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix products |
| DbCentury | char[2] | Specifies the value for the **DBCENTURY** environment variable |
| DbCodeset | char[24] | Specifies the value for the **DBCODESET** environment variable

Provides compatibility for client applications that are based on 4.x versions of IBM Informix Asian Language Support (ALS) products |
| DbConnect | char[2] | Specifies the value for the **DBCONNECT** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix products |
| DbCsConv | char[9] | Specifies the value for the **DBCSCONV** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix products |
| DbCsOverride | char[2] | Specifies the value for the **DBCSOVERRIDE** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix Asian Language Support (ALS) products |
| DbCsWidth | char[12] | Specifies the value for the **DBCSWIDTH** environment variable

Provides compatibility for client applications that are based on earlier versions of IBM Informix products |
| DbFltMsk | char[4] | Specifies the value for the **DBFLTMASK** environment variable |

Table 1-7. Fields of the InetLogin structure (continued)

InetLogin Field	Data Type	Purpose
DbMoneyScale	char[6]	Specifies the value for the **DBMONEYSCALE** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix products
DbSS2	char[5]	Specifies the value for the **DBSS2** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix products
DbSS3	char[5]	Specifies the value for the **DBSS3** environment variable Provides compatibility for client applications that are based on earlier versions of IBM Informix products
OptoFC	char[2]	Not used
OptMSG	char[2]	Not used

All fields in the **InetLogin** structure, *except* **DbAnsiWarn**, **Client_Loc**, and **DB_Loc**, are of data type **char** and are null-terminated strings. The **Client_Loc** and **DB_Loc** fields are character pointers whose data space your Informix ESQL/C program must allocate.

InetLogin field values

Your application must set **InetLogin** values before it executes the SQL statement or Informix ESQL/C library function that needs the configuration information. It is recommended that you use the **ifx_putenv()** and **ifx_getenv()** functions to set and retrieve **InetLogin** field values through environment variables, but you can set the values of the **InetLogin** fields directly.

The following figure shows a dialog box that a client application might use to obtain network parameters from an end user. This application takes the account information that the user enters and sets the appropriate network values in the **InetLogin** structure.
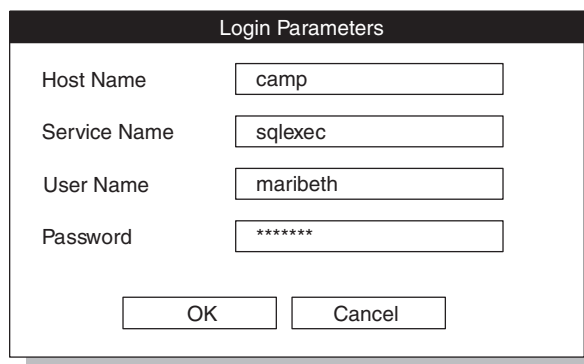
Figure 1-14. User dialog box for login parameters

The following figure shows a code fragment that sets login values in the **InetLogin** structure. The application can obtain these values from the end user through a dialog box (such as the one in Figure 1-14).

```
strcpy(InetLogin.InfxServer, "mainsrvr");

⋮

case IDOK:
    *szDlgString = '\0';
    GetDlgItemText (hdlg, IDC_HOST, szDlgString, cbSzDlgMax);
    strcpy(InetLogin.Host, szDlgString);

    *szDlgString = '\0';
    GetDlgItemText (hdlg, IDC_USER, szDlgString, cbSzDlgMax);
    strcpy(InetLogin.User, szDlgString);
```

Figure 1-15. Code to Prompt the User for InetLogin Values

In the previous figure, if the user enters host information, the fragment sets the
InetLogin.Host and **InetLogin.User** fields for the **mainsrvr** database server to the
user-specified names of the host name and user name. If the user does not enter
host information, Informix ESQL/C uses the HOST and USER Registry values
from the subkey for the **mainsrvr** database server.

Tip: For another example of how to set the InetLogin fields, see the ILOGIN
demonstration program in the **%INFORMIXDIR%\demo\ilogin** directory.

Precedence of Configuration Values

When a client application in a Windows environment requires configuration
information, IBM Informix ESQL/C obtains it from the following locations:

1. The **InetLogin** structure

   If the application uses the **InetLogin** structure, Informix ESQL/C first checks
   for configuration information in this structure. (To set the value of an
   environment variable for the application process, the **ifx_putenv()** function
   changes the value of an **InetLogin** field.)

2. The INFORMIX subkey of the Registry

   If the application has not set the configuration information you want in
   InetLogin, Informix ESQL/C checks for this information in its copy of the
   Registry information. For more information about how to set the Registry, see
   the *IBM Informix Client Products Installation Guide*.

You do not need to define all the values in the **InetLogin** structure. The application
uses the configuration information in the Registry for any values it cannot find in
InetLogin. If you do not set the corresponding Registry value, the application uses
its default value.

Important: The first time that the application requires configuration information,
Informix ESQL/C reads this information from the Registry and stores
it in memory. For subsequent references to Registry information,
Informix ESQL/C accesses this in-memory copy and does not reread
the Registry.

This hierarchy of configuration information is valuable if, for example, you want
the application user to provide a user name and password at runtime, or if an
application has some configuration information that differs from the general values
in the Registry. For example, suppose the application sets the **ConRetry** field of
InetLogin to 2 but does not set the **ConTime** field, as the following code fragment
shows:

```
        strcpy(InetLogin.ConRetry, "2");
        EXEC SQL connect to 'accnts';
```

When Informix ESQL/C establishes the connection to the **accnts** database, it tries to establish the connection twice (instead of the default value of once) but it still uses a connection time of 15 seconds (the default value from the in-memory copy of the Registry information). If **Setnet32** has modified the connection values, Informix ESQL/C uses the modified Registry values instead of the default values.

Tip: Use the **Setnet32** utility to define configuration information in the Registry. For more information about **Setnet32**, see the *IBM Informix Client Products Installation Guide*.

Using Global ESQL/C Variables in a Windows Environment

In earlier versions of the Informix ESQL/C product, Informix ESQL/C provided several global variables to support different features. The following table describes these global variables.

Table 1-8. Global ESQL/C Variables

Global Variable	Description
SQLSTATE	An ANSI-compliant status code as a five-character string (plus null terminator)
	For more information about SQLSTATE, see Chapter 11, "Exception handling," on page 11-1.
SQLCODE **sqlca.sqlcode**	An Informix-specific status code as an integer value
	For more information about SQLCODE, see Chapter 11, "Exception handling," on page 11-1.
sqlca structure	Informix-specific diagnostic information
	For more information about this structure, see Chapter 11, "Exception handling," on page 11-1.
FetBufSize and **BigFetBufSize**	The size of the fetch buffer
	BigFetBufSize is same as **FetBufSize** except for a higher upper limit value of the cursor buffer
	For more information about **FetBufSize** and **BigFetBufSize**, see Chapter 14, "Using Dynamic SQL," on page 14-1.
InetLogin structure	Environment information for the client ESQL/C application
	For more information, see "InetLogin structure" on page 1-33.

In environments, Informix ESQL/C implements the global variables in Table 1-8 as functions, which the **sqlhdr.h** file defines. These functions return values that have the same data types as their global-variable counterparts. Therefore, this change in implementation does not require modification of existing Informix ESQL/C code. You can still use these functions in the same context as their global-variable counterparts.

A Sample IBM Informix ESQL/C Program

The **demo1.ec** program illustrates most of the concepts that this chapter presents, such as include files, identifiers, host variables, and embedded SQL statements. It demonstrates how to use header files, declare and use host variables, and embed SQL statements.

Important: If you are using UNIX, you can find an online version of this and other demonstration programs in the **$INFORMIXDIR/demo/esqlc** directory. If you are using Windows, you can find the demonstration programs in the **%INFORMIXDIR%\demo\esqldemo** directory.

Compiling the demo1 Program

The following command compiles the **demo1** program:

```
esql demo1.ec
```

On UNIX, the name of the executable program defaults to **a.out**.

In Windows environments, the name of the executable program defaults to **demo.exe**.

You can use the **-o** option to assign a different name to the executable program. For more information about the **esql** command, see "Using the esql Command" on page 2-4.

Guide to demo1.ec File

The sample IBM Informix ESQL/C program, **demo1.ec**, uses a *static* SELECT statement. This means that at compile time the program can obtain all of the information that it needs to run the SELECT statement.

The **demo1.ec** program reads from the **customer** table in the **stores7** database the first and surnames of customers whose surname begins with a value less than `'C'`. Two host variables (**:fname** and **:lname**) hold the data from the **customer** table. A cursor manages the rows that the database server retrieves from the table. The database server fetches the rows one at a time. The program then prints the names to standard output.

```
1. #include <stdio.h>
2. EXEC SQL define FNAME_LEN     15;
3. EXEC SQL define LNAME_LEN     15;
4. main()
5. {
6. EXEC SQL BEGIN DECLARE SECTION;
7.    char fname[ FNAME_LEN + 1 ];
8.    char lname[ LNAME_LEN + 1 ];
9. EXEC SQL END DECLARE SECTION;
```

Line 1

The **#include** statement tells the C preprocessor to include the **stdio.h** system header file from the **/usr/include** directory. The **stdio.h** file enables **demo1** to use the standard C language I/O library.

Lines 2 to 3

Informix ESQL/C processes the **define** directives in stage 1 of preprocessing. The directives define the constants FNAME_LEN and LNAME_LEN, which the program uses later in host-variable declarations.

Lines 4 to 9

Line 4 begins the **main()** function, the entry point for the program. The EXEC SQL block declares host variables that are local to the **main()** function that receive data from the **fname** and **lname** columns of the **customer** table. The length of each array is 1 byte greater than the length of the character column from which it receives data. The extra byte stores the null terminator.

```
10.      printf( "DEMO1 Sample ESQL Program running.\n\n");
11.      EXEC SQL WHENEVER ERROR STOP;
12.      EXEC SQL connect to 'stores7';
13.    EXEC SQL DECLARE democursor cursor for
14.       select fname, lname
15.       into :fname, :lname
16.       from customer
17.       where lname < 'C';
18.    EXEC SQL open democursor;
```

Lines 10 to 12

The **printf()** function shows text to identify the program and to notify the user when the program begins to execute. The WHENEVER statement implements a minimum of error handling, causing the program to display an error number and terminate if the database server returns an error after processing an SQL statement. The CONNECT statement initiates a connection to the default database server and opens the **stores7** demonstration database. You specify the default database server in the **INFORMIXSERVER** environment variable, which you must set before an application can connect to *any* database server.

Lines 13 to 17

The DECLARE statement creates a cursor that is called **democursor** to manage the rows that the database server reads from the **customer** table. The SELECT statement within the DECLARE statement determines the type of data that the database server reads from the table. This SELECT statement reads the first and surnames of those customers whose surname (**lname**) begins with a letter less than `'C'`.

Line 18

The OPEN statement opens the **democursor** cursor and begins execution of the SELECT statement.

```
19.      for (;;)
20.        {
21.        EXEC SQL fetch democursor;
22.        if (strncmp(SQLSTATE, "00", 2) != 0)
23.           break;
24.        printf("%s %s\n",fname, lname);
25.        }
26.    if (strncmp(SQLSTATE, "02", 2) != 0)
27.        printf("SQLSTATE after fetch is %s\n", SQLSTATE);
28.    EXEC SQL close democursor;
29.    EXEC SQL free democursor;
```

Lines 19 to 25

This section of code executes a FETCH statement inside a loop that repeats until SQLSTATE is not equal to "00". This condition indicates that either the end-of-data condition or a runtime error has occurred. In each iteration of the loop, the FETCH statement uses the cursor **democursor** to retrieve the next row that the SELECT statement returns and to put the selected data into the host variables **fname** and **lname**. The database server sets status variable SQLSTATE to "00" each time it

fetches a row successfully. If the end-of-data condition occurs, the database server sets SQLSTATE to "02"; if an error occurs, it sets SQLSTATE to a value greater than "02". For more information about error handling and the SQLSTATE status variable, see Chapter 10, "Working with opaque data types," on page 10-1.

Lines 26 to 27

If the class code in SQLSTATE is any value except "02", then the SQLSTATE value for the user is displayed by this **printf()**. This output is useful in the event of a runtime error.

Lines 28 to 29

The CLOSE and FREE statements free the resources that the database server had allocated for the cursor. The cursor is no longer usable.

```
30.     EXEC SQL disconnect current;
31.     printf("\nDEMO1 Sample Program over.\n\n");
32. }
```

Lines 30 to 32

The DISCONNECT CURRENT statement closes the database and terminates the current connection to a database server. The final **printf()** tells the user that the program is over. The right brace (}) on the line 32 marks the end of the **main()** function and of the program.

Chapter 2. Compiling programs

These topics contain the following information:
- Compiling an IBM Informix ESQL/C program
- Using the **esql** command
- Compiling and linking options of the **esql** command
- Windows environment system processor options available to the **esql** command
- Accessing the ESQL Client-Interface in Windows environments

Compiling an IBM Informix ESQL/C program

You use the **esql** command to compile your IBM Informix ESQL/C program. The **esql** command passes your Informix ESQL/C source file to the Informix ESQL/C preprocessor and to the C compiler. It passes along options that are specific to both the Informix ESQL/C preprocessor and the C compiler to preprocess, compile, and link your Informix ESQL/C program.

ESQL/C Preprocessor

To preprocess, compile, and link a program that contains IBM Informix ESQL/C statements, you must pass it through the Informix ESQL/C preprocessor. You use the **esql** command to run the preprocessor on your Informix ESQL/C source file and create an executable file. The **esql** command follows these steps to carry out the conversion:

- In stage one, the Informix ESQL/C preprocessor performs the following steps:
  - Incorporates header files into the source file when it processes all **include** directives (**$include** and EXEC SQL **include** statements)
  - Creates or removes compile-time definitions when it processes all **define** (**$define** and EXEC SQL **define**) and **undef** (**$undef** and EXEC SQL **undef**) directives
- In stage two, the Informix ESQL/C preprocessor processes any conditional compilation directives (**ifdef**, **ifndef**, **else**, **elif**, **endif**) and translates embedded SQL statements to Informix ESQL/C function calls and special data structures.

  Stages one and two mirror the preprocessor and compiler stages of the C compiler. Successful completion of the preprocessing step yields a C source file (**.c** extension). For information about command-line options that affect the preprocessing step, see "Options That Affect Preprocessing" on page 2-11.

The **esql** command processor is installed as part of the Informix ESQL/C product. Before you use **esql**, make sure that:

- the file name of the Informix ESQL/C source file has the **.ec** or the **.ecp**.
- the **INFORMIXDIR** and **PATH** environment variables are set correctly.

  If the **INFORMIXDIR** environment variable is not set in the command window or in the Windows Registry, it will be set internally to the location of the IBM Informix Client SDK dynamically-linked libraries.

  If the **INFORMIXDIR** environment variable is not set in UNIX, an error is returned when compiling any IBM Informix Client Software Development Kit (Client SDK) application.

  For information about how to set the **INFORMIXDIR** and **PATH** variables, see the *Client Products Installation Guide* for your operating system.

For a detailed explanation of the syntax of the **esql** command, see "Using the esql Command" on page 2-4.

Warning: Always link your Informix ESQL/C program with the **esql** program. The lists of libraries that Informix uses can change between releases. Linking with esql assures that your Informix ESQL/C program will link correctly with Informix libraries.

The C code that the Informix ESQL/C preprocessor generates might change from one release of the product to the next. Therefore, do not design Informix ESQL/C programs that depend on how Informix implements the functionality and features of the product in the C code that the Informix ESQL/C preprocessor generates. Instead, develop your programs with the functionality and features of the product that this publication describes.

C Preprocessor and Compiler

The **esql** command does not itself compile and link the Informix ESQL/C program. The **esql** command translates Informix ESQL/C code to C code and then calls the C compiler to compile and link the C code. The C preprocessor preprocesses the C language preprocessing directives. The C compiler performs the compilation, and it also calls a link editor to link the C object files.

Your Informix ESQL/C source file contains commands for the C preprocessor (directives of the form #*directive*). When you use the default order of compilation, these C directives have no effect on Informix ESQL/C statements but take effect in the usual way when the C compiler processes the source file.

If you choose to run the C preprocessor on the Informix ESQL/C source file before the Informix ESQL/C preprocessor, you can use the C language **#defin**e and **typedef** directives to define Informix ESQL/C host variables. For more information, see "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

The C compiler takes the following actions:
* Compiles the C language statements to object code
* Links to Informix ESQL/C libraries and any other files or libraries you specify
* Creates an executable file

Note:
If you use a compiler other than the local C compiler by setting the INFORMIXC environment variable to a non-default value, you might need to override that compiler's default options. For information about the INFORMIXC environment variable, see the*IBM Informix Guide to SQL: Reference*.

For information about command-line options that affect the compilation and linking steps, see "Compiling and Linking Options of the esql Command" on page 2-23.

Default Compilation Order

After you have created an IBM Informix ESQL/C program file, you run the **esql** command on that file. By default, the Informix ESQL/C preprocessor runs first and translates the embedded SQL statements in the program into Informix ESQL/C function calls that communicate with the database server. The Informix ESQL/C

preprocessor produces a C source file and calls the C compiler. The C compiler then preprocesses and compiles your source file and links any other C source file, object file, or library file the same way as any other C program. If **esql** does not encounter errors in one of these steps, it generates an executable file. You can run the compiled Informix ESQL/C program as you would any C program. When the program runs, it calls the Informix ESQL/C library procedures; the library procedures set up communications with the database server to carry out the SQL operations.

The following figure illustrates the process by which an Informix ESQL/C program becomes an executable program.

ESQL/C source   ESQL/C preprocessor   C source program   C language   Executable
program   stages 1 and 2   with SQL statements   preprocessor   program
                                          converted toESQL/C   and compiler
                                        calls
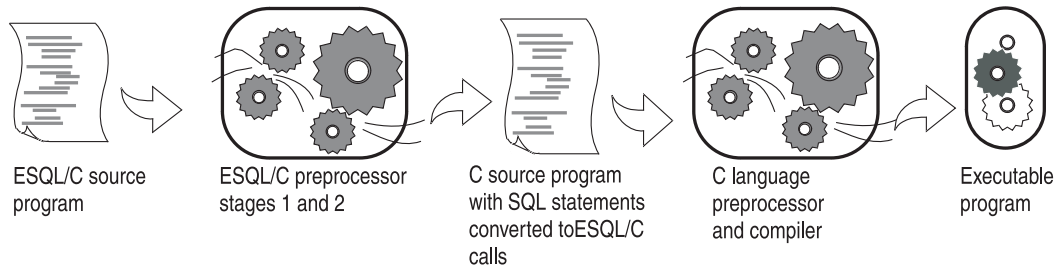
Figure 2-1. Relationship Between IBM Informix ESQL/C and C

> **Important:** Keep in mind that with the default order of compilation, **esql** handles Informix ESQL/C preprocessor directives before it calls the C compiler. Therefore, the Informix ESQL/C directives take effect before the C compiler performs any preprocessing. You cannot access Informix ESQL/C definitions within C preprocessor directives, nor can you use the C preprocessor to perform conditional compilation of Informix ESQL/C statements.

Running the C Preprocessor First

With IBM Informix ESQL/C, you change the default order of processing when you compile your Informix ESQL/C program. Informix ESQL/C allows you to run the C preprocessor on the Informix ESQL/C source file first, and then pass that file to the Informix ESQL/C preprocessor. This feature enables your Informix ESQL/C program to access variables made available by C preprocessor directives. For more information about this option, see "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

Using the esql Command

To create an executable C program from an Informix ESQL/C source file, use the **esql** command. The IBM Informix installation script installs the **esql** command as part of the Informix ESQL/C product. This section describes what the **esql** command can do and how you use it.

The **esql** command performs the following steps:
1. Converts the embedded SQL statements to C language code.
2. Converts your ESQL/C source files to C language source files.
3. Compiles the files that result with the C compiler to create an object file.
4. Creates the resource compiler and links any resource files (**.res**) that you specify on the **esql** command line for Windows.
5. Links the object file with the Informix ESQL/C libraries and your own libraries.

Requirements for Using esql

Before you use **esql**, make sure that:

* The file name of your Informix ESQL/C source files have the **.ec** extension. You can also use the **.ecp** extension if you want the C preprocessor to run before the Informix ESQL/C preprocessor.

* You have set the environment variable **INFORMIXDIR** correctly and the **PATH** environment variable includes the path to the bin directory of the INFORMIXDIR directory (**$INFORMIXDIR/bin** on the UNIX operating system and **%INFORMIXDIR%\bin** in Windows environments).

   For a complete description of **INFORMIXDIR**, see the *IBM Informix Guide to SQL: Reference* or the *IBM Informix Client Products Installation Guide* for your operating system.

Syntax of the esql command

The following topics describe the syntax of the **esql** command. This section organizes the command-line options by the processing phase that they affect:

* Preprocessing options determine how **esql** translates the embedded SQL statements.

* Compilation options affect the compilation phase, when the C compiler translates the C source to object code.

* Linking options affect the linking phase, when the C compiler links the object code to produce an executable file.

```
                      (1)
►►──esql──┬─────────────────────────┬──┬──────┬──┬──────────────────┬──┬───────┬──►
          │◄──────────────────────┐ │  └──-e──┘  └──-log──logfile────┘  └──-libs─┘
          ├──-ansi──────────────────┤
          ├──-cc──ccargs────────────┤
          ├──-EDname──┬───────────┬──┤
          │           └──=value────┘  │
          ├──-EUname─────────────────┤
          ├──-g─────────────────────┤
          ├──-G──┬──────┬───────────┤
          │      └──-nln─┘          │
          ├──-Ipathname─────────────┤
          ├──-icheck────────────────┤
          ├──-local─────────────────┤
          ├──-nowarn────────────────┤
          ├──source.ec──────────────┤
          └──otherarg───────────────┘

►──┬──────────────────┬──┬──────────┬──┬──────────┬──┬──────┬──┬────────────┬──►◄
   └──-o──outfile──────┘  └──-thread─┘  └──-static─┘  └──-V──┘  └──-version───┘
```

```
                                                    (2)                ┌─-xopen─┐
├─┬─ UNIX-only arguments ─┬─────────────────────────────────────────────────────►◄
  │                       (3)                     │
  └─ Windows-only arguments ─┘
```
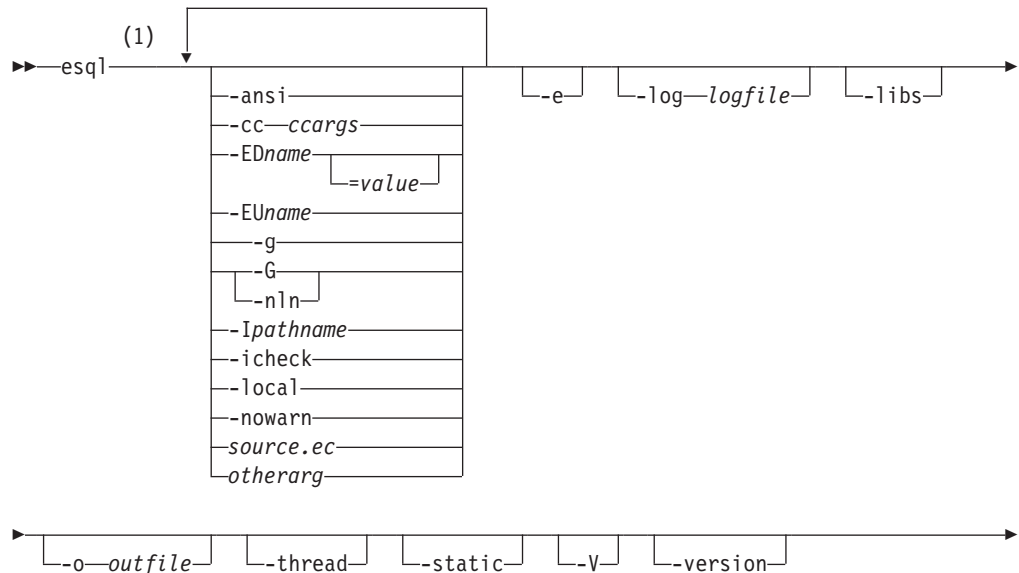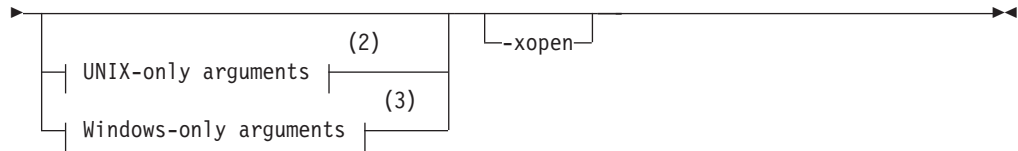
Notes:

1 Order of the arguments that follow **esql** at this level does not matter. For example **-e** can come before **-local**.

2 Allowed only if **esql** is running on a UNIX operating system.

3 Allowed only if **esql** is running on a Windows operating system.

-ansi Causes **esql** to warn you if the source file uses Informix extensions to ANSI-standard SQL syntax. This argument only affects source files to the right of it on the command line. See "Checking for ANSI-Standard SQL Syntax" on page 2-12.

-cc *ccargs*

Passes *ccargs* to the C compiler without interpreting or changing them. The variable *ccargs* represents all of the arguments between the **-cc** and the next occurrence of any of these arguments:

- **-l** (Windows only)
- **-r** (Windows only)
- **-f** (Windows only)
- any file name except those that are arguments for an option

See "Pass options to the C compiler" on page 2-23.

-e Preprocesses only, no compiling or linking. The ESQL/C preprocessor produces a C source file with a **.c** extension. See "Preprocessing Without Compiling or Linking" on page 2-12.

-ED*name*

Creates a definition for *name*. The effect is the same as if the source file contained an ESQL/C **define** directive for *name*. If =*value* is included, the definition is set to *value*. For details, see "Defining and Undefining Definitions While Preprocessing" on page 2-13..

-EU*name*

Undefines the Informix ESQL/C definition named *name*. The effect is as if the source file included an Informix ESQL/C **undef** directive for that name. For details, see "Defining and Undefining Definitions While Preprocessing" on page 2-13..

-g Reverses the effects of the last **-G** option for source files to the right of this option on the command line. See "Numbering Lines" on page 2-15.

-G Normally **#line** directives are added to the C source code so that the C compiler can direct you to the correct line in the Informix ESQL/C file when it detects an error in the C file. The **-G** option turns off this feature for the Informix ESQL/C source files that follow it on the command line. Use the **-g** argument to turn the feature back on. The **-nln** argument is a synonym for **-G**. See "Numbering Lines" on page 2-15.

-I*pathname*

Adds *pathname* to the search path for Informix ESQL/C and C include

files. The search path is used when searching for the files named in **include** and **#include** directives. See "Naming the Location of Include Files" on page 2-14.

-icheck
Tells **esql** to add code that generates an error if a null value is returned to a host variable that does not have an indicator variable associated with it. This argument only affects source files to the right of it on the command line. See "Checking for Missing Indicator Variables" on page 2-14.

-local Specifies that the static cursor names and static statement IDs that you declare in a source file are local to that file. If you do not use the **-local** option, cursor names and statement IDs, by default, are global entities. This argument only affects source files to the right of it on the command line. See "Setting the Scope of Cursor Names and Statement IDs" on page 2-15.

-log *logfile*
Sends the error and warning messages generated by the Informix ESQL/C preprocessor to the specified file instead of to standard output. This option affects only preprocessor errors and warnings. See "Redirecting Errors and Warnings" on page 2-16.

-libs Prevents all compiling and linking and instead shows the names of all the libraries that would be linked based on the other options.

-nln Synonym for **-G**.

-nowarn
Suppresses warning messages from the preprocessor. Error messages are still issued. This argument only affects the preprocessing of source files to the right of it on the command line. See "Suppressing Warnings" on page 2-16.

-o *outfile*
Specifies the name of the output file that will be created by the compiler. See "Naming the Executable" on page 2-23.

otherarg
Any argument that **esql** does not recognize or deal with directly is passed to the C compiler. This allows you to include libraries, resource files, C compiler options, and similar arguments on the command line. If an argument that you want to pass to the C compiler conflicts with one of the **esql** arguments, use the **-cc** option to protect it from **esql**. See "Pass options to the C compiler" on page 2-23.

source.ec
An Informix ESQL/C source file with the default suffix **.ec**.

-thread
Tells the Informix ESQL/C preprocessor to create thread-safe code. See "Specifying Versions of Informix General Libraries" on page 2-28.

-static Links Informix static libraries instead of the default Informix shared libraries. See "Specifying Versions of Informix General Libraries" on page 2-28.

-V Prints the version information for your Informix ESQL/C preprocessor then exits. If this argument is given then all other arguments are ignored.

-version

    Prints the build and version information for your Informix ESQL/C preprocessor then exits. If this argument is given then all other arguments are ignored.

-xopen

    Generates warning messages for SQL statements that use Informix extensions to the X/Open standard. It also indicates that dynamic SQL statements use the X/Open set of codes for data types (when using GET DESCRIPTOR and SET DESCRIPTOR statements or an **sqlda** structure). See "Using X/Open Standards" on page 2-16.

UNIX-only arguments:

Notes:

1   Order of the arguments at this level does not matter.

-cp   Causes **esql** to run the C preprocessor before the Informix ESQL/C preprocessor when processing *source.ec* files. The SQL keywords in the file are protected from interpretation by the C preprocessor and the protection is removed after the C preprocessor runs. This argument only affects source files to the right of it on the command line. See "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

-glu   Compile such that your application can use GLU (GLS for Unicode). For details see "Using the GLS for Unicode (GLU) Feature" on page 2-16.

-np   Prevents the protecting of SQL statements in source files that are processed by the C preprocessor before being processed by the Informix ESQL/C preprocessor. This argument only affects source files to the right of it on the command line. See "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

-nup   No unprotect mode. The SQL keyword protection is not removed after the C preprocessor is run. The compilation stops after the C preprocessor and the results are put in a file with the extension **.icp**. See "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

-onlycp

    This mode is like the **-cp** mode in that it forces the C preprocessor to run first before the Informix ESQL/C preprocessor. However, the processing stops after the C preprocessor runs, leaving the result in a **.icp** file. See "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

source.ecp

    An Informix ESQL/C source file with the special suffix **.ecp**. It is treated as if it were a normal Informix ESQL/C file that was preceded with the **-cp** option. See "Running the C Preprocessor Before the ESQL/C Preprocessor" on page 2-17.

Windows-only arguments:

```
    ┌─────────────────────────┐
    ▼                         │
├───┬──-l──largs───┬──────────┴──┬─@respfile─┬──┬─-dcmdl─┬──┬─-f──filename─┬──┬─-mserr─┬──►
    ├──-lw:width───┤             └───────────┘  └────────┘  └──────────────┘  └────────┘
    └──-ts:width───┘
```

```
                                       (1)
├──┬────┬──┬────┬──┬─────┬──┬──-cpu:──┬──────┬──alpha─┬──────────┬──►
   └─-n─┘  └─-p─┘  └─-mc─┘  │         │      ├──i386──┤          │
                           │         │      └──mips──┘          │
                           │         ├──-pa──────────────────────┤
                           │         ├──-pi──────────────────────┤
                           │         └──-pm──────────────────────┘
```

```
                 (1)                                (1)
├──┬──-runtime:──────┬──┬─libc─┬──────┬──┬──-target:──────┬──dll─┬──┬──►
   │            (1)  │  ├─s────┤      │  │                └─exe─┘  │
   └──-rt:───────────┘  ├─libcmt──────┤  ├──-wd───────────────────┤
                        ├─m───────────┤  └──-we───────────────────┘
                        ├─msvcrt──────┤
                        └─d───────────┘
```

```
                    (1)
├──┬──-subsystem:────────┬──┬─console─┬─────┬──────────────────────┤
   │               (1)   │  ├─c───────┤     │
   └──-ss:───────────────┘  ├─windows─┤     │
                            └─w───────┘     │
   ├──-Sc──────────────────────────────────┤
   └──-Sw──────────────────────────────────┘
```
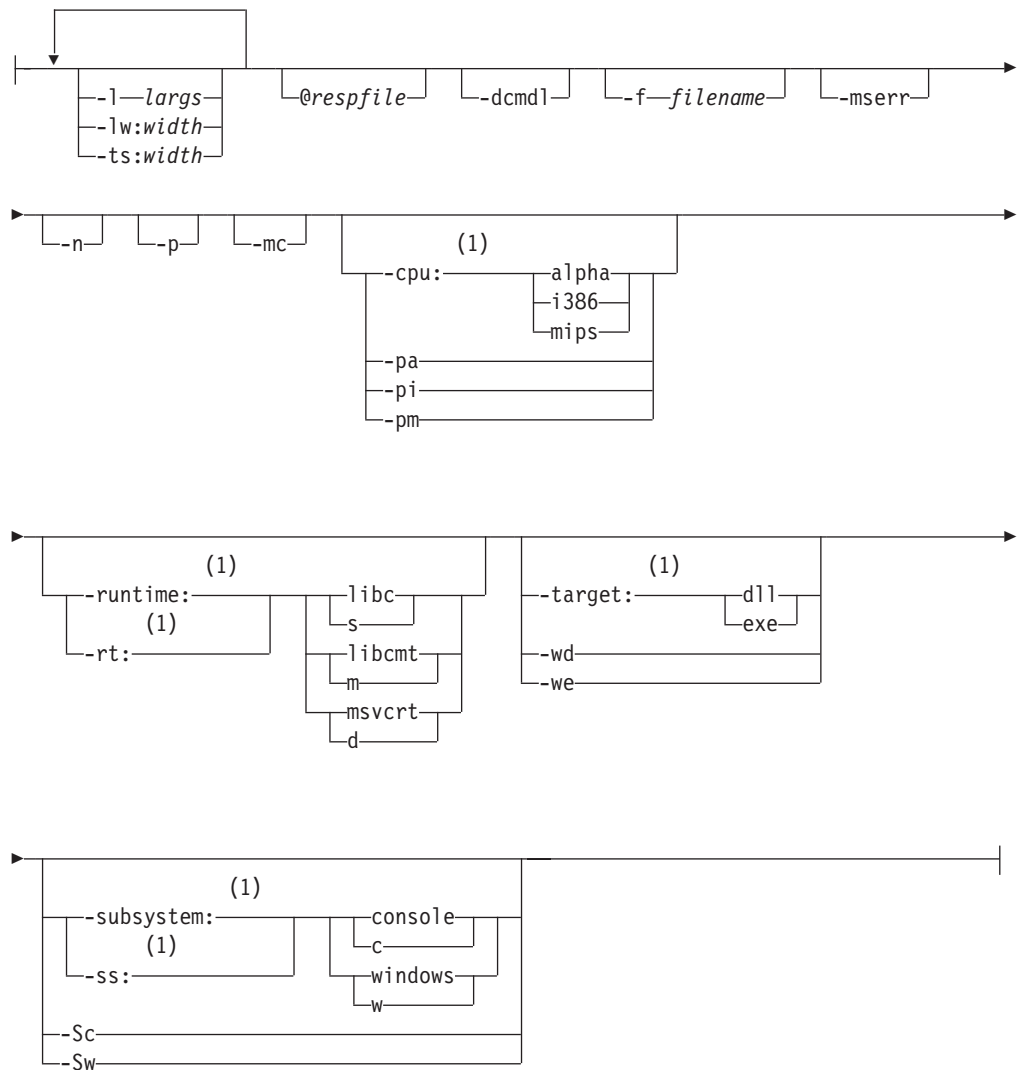
Notes:

1   There should be no space between this option and the part that follows.

@ *respfile*   Specifies a file containing additional options. For details see "Creating a Response File" on page 2-25

-bc   Tells the preprocessor to use the Borland C compiler instead of the Microsoft Visual C++ compiler. See "Specify a particular C compiler (Windows)" on page 2-23.

-cpu: This argument has no effect if you are using Borland C to compile. This argument tells esql what type of processor you would like the executable to be optimized for. There are three possible values:

         alpha   For processors that are compatible with the Alpha architecture.

         i386   For processors that are compatible with the Intel386 architecture. This is the default.

mips	For processors that use the MIPS32 or MIPS64 instructions set architecture (ISA).
-dcmdl	Shows the command line used to start the C compiler. This lets you visually verify the options that are used.
-f *filename*	Specifies the name of a file that contains the names of additional Informix ESQL/C source files.
-l *largs*	Passes *largs* to the linker without interpreting or changing them. The *largs* is all of the arguments between the **-cc** and a **-r** option or the end of the line. See "Passing Arguments to the Linker" on page 2-28.
-lw:*width*	When an Informix ESQL/C source file is converted into a C source file this argument causes lines in the C source file to be wrapped at the column position that *width* indicates. This argument only affects source files to the right of it on the command line. See "Line Wrapping" on page 2-22.
-mc	Tells the preprocessor to use the Microsoft Visual C++ compiler to compile and link. See "Specify a particular C compiler (Windows)" on page 2-23.
-mserr	Provides Microsoft-style messages and warnings.
-n	Prevents **esql** from printing a version message when it runs.
-p	Synonym for **-e**.
-pa	Synonym for **-cpu:alpha**.
-pi	Synonym for **-cpu:i386**.
-pm	Synonym for **-cpu:mips**.
-rt:	Synonym for **-runtime:**.
-runtime:	Determines what C runtime libraries will be linked with the executable. An indicator of the library type must follow this option with no space in between. The type must be one of the following:

d	Links a multithreaded shared library. This is the default library that is used if **-runtime:** is not given. You can also use the library name in place of **d**. If you are using Microsoft Visual C++ to compile, the library name is **msvcrt**. If you are using Borland C, it is **cw32mti**.
m	Links a static multithreaded shared library. You can also use the library name in place of **m**. If you are using Microsoft Visual C++ to compile, the library name is **libcmt**. If you are using Borland C, it is **cw32mt**. Cannot be used with the **-static** option.
s	Links a static single-threaded library. You can also use the library name in place of **s**. If you are using Microsoft Visual C++ to compile, the library name is **libc**. If you are using Borland C, it is **cw32**. Cannot be used with the **-static** option.
t	This option can be used only if you are using Borland C. It links the static multithreaded library. You can also use the library name **cw32i** in place of **t**. Cannot be used with the **-static** option.

-Sc	Synonym for **-subsystem:console**.
-ss:	Synonym for **-subsystem:**
-Sw	Synonym for **-subsystem:windows**.
-subsystem:	Determines what subsystem will be linked into the executable. An indicator of the subsystem type must follow this option with no space in between. The type must be one of the following:

> **console**
> This is the default type. It creates a console application. This indicator can be abbreviated as **c**.

> **windows**
> Creates a Windows application. This indicator can be abbreviated as **w**.

> The **-subsystem:** option can be abbreviated **-ss:**.

-target:	Determines what type of file will be created. An indicator of the target type must follow this option with no space in between. The indicator must be one of the following:

> **dll** A Dynamic Load Library (DLL) file will be created.

> **exe** This is the default type. A regular executable file will be created.

-ts:*width*	Tells the preprocessor to define tab stops every *width* columns when creating the C source file. By default, the preprocessor sets tab stops every eighth column. See "Setting Tab Stops" on page 2-23.
-v	Synonym for **-V**.
-wd	Synonym for **-target:dll**.
-we	Synonym for **-target:exe**.

Options That Affect Preprocessing

An Informix ESQL/C program must be preprocessed before a C compiler can compile it. The Informix ESQL/C preprocessor converts the embedded SQL statements to C language code.

You can use all the preprocessor options that the following sections describe for preprocessing only or for preprocessing, compiling, and linking.

Checking the Version Number
Use the **-V** option to obtain the IBM Informix version number and serial number for your Informix ESQL/C product, as the following example shows:
```
esql -V
```

The **-version** option provides more detailed version information that might be needed when dealing with IBM Informix technical support.

Associating Options with Files
Many preprocessor options affect only files that are displayed to the right of the option on the command line. For example in this command line:
```
esql -G source1.ec -ansi source2.ec
```

The **-G** option affects the **source1.ec** file, and both the **-ansi** and the **-G** options affect the **source2.ec** file.

Preprocessing Without Compiling or Linking

By default, the **esql** command causes the IBM Informix ESQL/C program to be preprocessed, compiled, and linked. The output of the **esql** command is an executable file. You can specify the **-e** option to suppress the compilation and linking of your Informix ESQL/C program. With this option, **esql** only causes preprocessing of the code. The output of this command is a C source file (**.c** extension).

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

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

The preceding command would generate a C source file that is called **demo1.c**. The following **esql** command preprocesses **demo1.ec**, checks for Informix extensions to ANSI-standard syntax, and does not use line numbers:

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

Generating Thread-Safe Code

You can use the **-thread** option to instruct the preprocessor to generate thread-safe code.

You must use the **THREADLIB** environment variable with this option to specify which thread package to use when you compile your application.

For Windows environments, the IBM Informix general libraries (**libgen**, **libos**, **libgls**, **libafs**, and **libsql)** are shared, thread-safe DLLs. Therefore, the **esql** command links the shared, thread-safe DLLs automatically. You do not set the **THREADLIB** environment variable when you compile multithreaded applications in a Windows environment.

Checking for ANSI-Standard SQL Syntax

When you compile an IBM Informix ESQL/C program, you can instruct the preprocessor to check for Informix extensions to ANSI-standard SQL syntax in one of two ways:

- You can set the **DBANSIWARN** environment variable.

  After you set the **DBANSIWARN** environment variable, every time you compile or run an Informix ESQL/C program, Informix ESQL/C checks for ANSI compatibility. For information about how to set **DBANSIWARN**, see the *IBM Informix Guide to SQL: Reference*. For details about how to check for runtime warnings, see Chapter 10 of this publication. For details on how to set environment variables, see the *IBM Informix Client Products Installation Guide* for your operating system.

- You can specify the **-ansi** option at compile time whenever you want to check for ANSI compatibility.

  The **-ansi** option does not cause Informix ESQL/C to check for ANSI compatibility at runtime.

With the **-ansi** option, the Informix ESQL/C preprocessor generates a warning message when it encounters an Informix extension to ANSI SQL syntax. The following **esql** command preprocesses, compiles, and links 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 compile a program with both the **-ansi** and **-xopen** options, the Informix ESQL/C preprocessor generates warning messages for Informix extensions to both ANSI and X/Open SQL syntax.

Defining and Undefining Definitions While Preprocessing

You can use the **-ED** and **-EU** options to create or remove definitions during IBM Informix ESQL/C preprocessing.

To create a global definition, use one of the following forms of the **-ED** option:
- Use the **-ED***name* syntax to define a Boolean symbol, as follows:
  ```
  esql –EDENABLE_CODE define_ex.ec
  ```
- Use the **-ED***name=value* syntax to define an integer constant, as follows:
  ```
  esql –EDMAXLENGTH=10 demo1.ec
  ```

The **-ED***name* is equivalent to the **define** preprocessor directive (**$define** or EXEC SQL **define**) with *name* at the top of your Informix ESQL/C program.

To remove or undefine a definition globally for the entire source file, use the following syntax for the **-EU** option:
```
–EUname
```

The **-EU** option has a global effect over the whole file, regardless of other **define** directives for *name*.

Important: Do not put a space between **ED** or **EU** and the symbol name.

As with the **define** and **undef** statements, the Informix ESQL/C preprocessor processes the -ED and -EU options in stage 1 of preprocessing (before it preprocesses the code in your source file). For more information about preprocessor definitions, see "The define and undef Directives" on page 1-30.

Figure 2-2 shows a code fragment that uses conditional compilation (the **ifdef** and **ifndef** directives).

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

main()
{
   .
   .
   .
EXEC SQL ifdef ENABLE_CODE;
printf("First block enabled");
EXEC SQL endif ENABLE_CODE;
   .
   .
   .
EXEC SQL ifndef ENABLE_CODE;
EXEC SQL define ENABLE_CODE;
EXEC SQL endif ENABLE_CODE;
   .
   .
   .
EXEC SQL ifdef ENABLE_CODE;
printf("Second block enabled");
EXEC SQL endif ENABLE_CODE;
}
```

Figure 2-2. ESQL/C Excerpt That Uses ifdef, ifndef, and endif

For the code fragment shown in Figure 2-2, the following esql command line does not generate code because the command line undefines the ENABLE_CODE definition for the entire source file:

```
esql –EUENABLE_CODE define_ex.ec
```

Checking for Missing Indicator Variables

If you include the **-icheck** option, the Informix ESQL/C preprocessor generates code in your program that returns a runtime error if an SQL statement returns a null value to a host variable that does not have an associated indicator variable. For example, the following command tells the preprocessor to insert code that checks for null values into the **demo1.ec** program:

```
esql –icheck demo1.ec
```

If you do not use the **-icheck** option, Informix ESQL/C does not generate an error if the database server passes a null value to a host variable without an indicator variable. For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

Naming the Location of Include Files

The **-I** preprocessor option allows you to name a directory where the preprocessor searches for IBM Informix ESQL/C and C include files. This option is valid for both the Informix ESQL/C and the C preprocessors as follows:

• The Informix ESQL/C preprocessor (**esql**) processes only Informix ESQL/C include files.

   You specify these include files with the **include** preprocessor directive **$include** or EXEC SQL **include**.

• The C preprocessor (**cc**) processes only the C include files.

   You specify these files with the **#include** preprocessor statement. Because the C preprocessing begins after the Informix ESQL/C compilation is completed, the C include files are processed *after* the Informix ESQL/C include files.

The preprocessor passes the **-I** option to the C compiler for processing of C include files (those that a **#include** preprocessor statement specifies). The syntax for the **-I** option is as follows:

```
esql -Idirectory esqlcprogram.ec
```

The *directory* can be on a mounted remote file system if the standard C library functions **fopen()**, **fread()**, and **fclose()** can access them.

The following **esql** command names the UNIX directory **/usr/johnd/incls** as a directory to search for Informix ESQL/C and C include files within the **demo1** program:

```
esql -I/usr/johnd/incls demo1.ec
```

Each **-I** option lists a single directory. To list several directories, you must list multiple **-I** options on the command line.

To search in both the **C:\dorrie\incl** and **C:\johnd\incls** directories in a Windows environment, you would need to issue the following command:

```
esql -IC:dorrie\incl -IC:\johnd\incls demo1.ec
```

When the preprocessor reaches an **include** directive, it looks through a *search path* for the file to include. It searches directories in this sequence:

1. The current directory
2. The directories that **-I** preprocessor options specify (in the order in which you specify them on the command line)
3. The directory **$INFORMIXDIR/incl/esql** on a UNIX operating system and the **%INFORMIXDIR%\incl\esql** in a Windows environment (where **$INFORMIXDIR** and **%INFORMIXDIR%** represent the contents of the environment variable of that name)
4. The directory **/usr/include**

For more information about Informix ESQL/C **include** files, see "The include Directive" on page 1-29.

Numbering Lines

By default, the Informix ESQL/C preprocessor puts **#line** directives in the **.c** file so that if an error is detected by the C compiler it will direct you to the line that generated the problem C code in the Informix ESQL/C source file. If you instead want to be directed to the problem line in the C file itself you can turn off line numbering by using the **-G** option. The **-G** option prevents the generation of **#line** directives in source code files that follow it on the command line. To turn line numbers back on, use the **-g** option. Files that follow the **-g** option will have **#line** directives generated.

Setting the Scope of Cursor Names and Statement IDs

By default, Informix ESQL/C defines cursor names and statement IDs as global entities. If you use the **-local** option, static cursor names and static statement IDs that you declare in a file are *local* to that file. To create the local name, Informix ESQL/C adds a unique tag (two to nine characters long) to the cursor names and statement IDs in an Informix ESQL/C program. If the combined length of the cursor name (or statement ID) and the unique tag exceeds 128 characters, you receive a warning message.

The **-local** option is provided primarily for compatibility with applications that were created in previous versions of Informix ESQL/C. Do not use this option

when you compile new applications. Do not mix files compiled with and without the **-local** flag. If you mix them, you might receive unpredictable results.

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

Redirecting Errors and Warnings

By default, esql directs error and warning messages it generates to standard output. If you want the errors and warnings to be put into a file, use the **-log** option with the file name. For example, the following esql command compiles the program **demo1.ec** and sends the errors to the **err.out** file:

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

This option only affects the error warnings that the Informix ESQL/C preprocessor generates. The warnings from the compile and link stages still go to the standard error output, which is **stderr** on UNIX, for example.

Suppressing Warnings

By default, the preprocessor generates warning messages when it processes an Informix ESQL/C file. To suppress these warning messages, use the **-nowarn** option. This option has no effect on error messages.

Using the GLS for Unicode (GLU) Feature

GLS for Unicode (GLU) is a feature that allows your application to use the International Components for Unicode (ICU) libraries instead of the usual GLS libraries when handling Unicode. The main advantage of using the ICU libraries is that they take the locale into account when collating Unicode characters, the GLS libraries do not.

To enable GLU:

1. Compile your application using the **-glu** option to the **esql** command.
2. Set the **GL_USEGLU** environment variable to 1 (one) in the environment of both client and database server.
3. Choose a locale that uses either Unicode or GB18030-2000 as a code set.

Using X/Open Standards

The **-xopen** option tells the IBM Informix ESQL/C preprocessor that your program uses X/Open standards. When you specify this option, the preprocessor performs the following two tasks:

• It checks for Informix extensions to X/Open-standard syntax.

  If you include Informix extensions to X/Open-standard syntax in your code, the preprocessor generates warning messages.

• It uses the X/Open set of codes for SQL data types.

  Informix ESQL/C uses these codes in a dynamic management structure (a system-descriptor area or an **sqlda** structure) to indicate column data types. Informix defines these codes in the **sqlxtype.h** header file.

If you use X/Open SQL in an Informix ESQL/C program, you must recompile any other source files in the same application with the **-xopen** option.

If you compile a program with both the **-xopen** and **-ansi** options, the Informix ESQL/C preprocessor generates warning messages for Informix extensions to both X/Open and ANSI SQL syntax.

Running the C Preprocessor Before the ESQL/C Preprocessor

The compilation of an IBM Informix ESQL/C source file can follow either the default order, where the Informix ESQL/C preprocessor runs first on the source file, or it can allow the C preprocessor to run on the source file before the Informix ESQL/C preprocessor.

The default sequence of compilation for an Informix ESQL/C source file is as follows:

1. The Informix ESQL/C preprocessor performs the following steps to create a **.c** file from the Informix ESQL/C source file:
   - Incorporates Informix ESQL/C header files into the source file when it processes all **include** directives (**$include** and EXEC SQL **include** statements)
   - Creates or removes compile-time definitions when it processes all **define** (**$define** and EXEC SQL **define**) and **undef** (**$undef** and EXEC SQL **undef**) directives
   - Processes any conditional compilation directives (**ifdef**, **ifndef**, **else**, **elif**, **endif**) and translates embedded SQL statements to Informix ESQL/C function calls and special data structures

2. The C preprocessor takes the following actions:
   - Incorporates C header files into the source file when it processes all C **include** directives (**#include**)
   - Creates or removes compile-time definitions when it processes all C language **define** (**#define**) and **undef** (**#undef**) directives
   - Processes C **conditional compilation** directives (**#ifdef**, **#ifndef**, **#else**, **#elif**, **#endif**)

3. The C compiler, assembler, and linker work in the usual way, as they do for a C source file, translating the C code file into an executable program.

This default order of compilation is restrictive because you cannot use **#defines** or **typedefs** defined in C system header files or custom C header files to define Informix ESQL/C host variables and constants, nor can you use them for conditional compilation of Informix ESQL/C code. With the default order of compilation, C header files do not get included into the Informix ESQL/C source code until after the Informix ESQL/C preprocessor has run, making these definitions unavailable to the Informix ESQL/C preprocessor.

Options for Running the C Preprocessor First

You can make the C preprocessor run on an Informix ESQL/C source file first, thus expanding any C-dependent **typedefs** or **#defines** inside the Informix ESQL/C source file before the Informix ESQL/C preprocessor is run on that source file. You can do this in any of the following ways:

- Pass the **-cp** or the **-onlycp** option to the esql command.

   Both force the C preprocessor to run first, but in the case of the **-cponly** option, the compilation stops after the C preprocessor is run, and the result is put in a **.icp** file.

- Create the Informix ESQL/C source file with a **.ecp** extension.

   This triggers the **-cp** option on the file automatically.

- Set the **CPFIRST** environment variables to TRUE (uppercase only).

- Use the **eprotect.exe** utility.

CPFIRST Environment Variable (UNIX)

The **CPFIRST** environment variable specifies whether the C preprocessor should run before the IBM Informix ESQL/C preprocessor on all Informix ESQL/C source files. Set the environment variable to TRUE (uppercase only) to run the C preprocessor on all Informix ESQL/C source files, irrespective of whether the **-cp** option is passed to the **esql** command, and irrespective of whether the source file has the **.ec** or the **.ecp** extension.

Using the eprotect.exe Utility (Windows)

Windows users can use the **eprotect.exe** utility to run the preprocessor on an IBM Informix ESQL/C source file. The **eprotect.exe** utility protects all of the SQL keywords against interpretation by the C preprocessor. The **eprotect.exe -u** option will remove SQL keyword protection.

To change the preprocessor order for an Informix ESQL/C source file on Windows:

1. Run the following command:

   ```
   %INFORMIXDIR%\lib\eprotect.exe filename.ec filename.c
   ```

   This protects all of the SQL keywords against interpretation by the C Preprocessor and writes the result to the file *filename*.**c**.

2. Run the following command

   ```
   cl /E filename.c   > filename2.c
   ```

   This runs the C Preprocessor on the source file *filename*.**c** and writes the result to the file *filename2*.**c**

3. Run the following command:

   ```
   %INFORMIXDIR%\lib\eprotect.exe -u filename2.c filename.ec
   ```

   This removes SQL keyword protection and stores the result in *filename*.**ec**.

4. Run **esql** on the source file to compile it.

The Order of Compilation When the C Preprocessor Runs First

When a user chooses to run the C preprocessor on an IBM Informix ESQL/C source file before the Informix ESQL/C preprocessor, the file undergoes the following order of compilation.

1. The **eprotect** utility runs on the source file to protect all SQL keywords against interpretation by the C preprocessor.
2. The C preprocessor runs on the source file.
3. The **eprotect** utility runs on the output of the C preprocessor with the **-u** mode to remove SQL keyword protection.
4. The Informix ESQL/C preprocessor runs on the output of the C preprocessor, which no longer has any SQL keyword protection.
5. The output of the Informix ESQL/C preprocessor undergoes compilation and linking by the C compiler and linker to produce an executable file.

For details on the phases of SQL keyword protection and unprotection, see "SQL Keyword Protection" on page 2-21.

Defining Host Variables Based on C #defines and typedefs

When the C preprocessor runs on a source file, it expands the contents of all C header files where they are included within the source file. This expansion makes it possible to use host variables in IBM Informix ESQL/C source files based on types and **#define** and **typedef** statements in C header files. The examples given

here indicate some of the advantages of sharing header files. In the example in Figure 2-3, the same C header file is used by both Informix ESQL/C and C source files.

```
/*header file i.h*/
#define LEN 15
typedef struct customer_t{
    int    c_no;
    char   fname[LEN];
    char   lname[LEN];
    } CUST_REC;
:

/*cust.ec*/
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
#include "i.h"
EXEC SQL END DECLARE SECTION;

int main()
{

EXEC SQL BEGIN DECLARE SECTION;
    CUST_REC cust;
EXEC SQL END DECLARE SECTION;
:

}


/*name.c*/
#include "i.h"
int main ()
{...
    CUST_REC cust;
:

}
```

Figure 2-3. ESQL/C and C Excerpt That Share a C Header File

In the example in Figure 2-4, the Informix ESQL/C source file contains a host variable based on a type defined in the **time.h** system-header file.

```
/*time.ec*/

#include <time.h>

main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
    time_t time_hostvar;
EXEC SQL END DECLARE SECTION;
:

}
```

Figure 2-4. ESQL/C Excerpt That Uses a Host Variable of the Type Defined in a C System-Header File

A C header file can be included anywhere inside an Informix ESQL/C source file. However, to define host variables in Informix ESQL/C files based on **#defines** and **typedefs** defined in C header files, you must include the C header file within the EXEC SQL declare section.

Contrast the example in Figure 2-5 on page 2-20, which leads to error -33051: `Syntax error on identifier or symbol 'name_hostvar` with the example in Figure 2-3 on page 2-19 which does not. The only difference is that in the example in Figure 2-3, the C header file with the **#define** and the **typedef** that is used in the EXEC SQL declare section is included within that declare section.

```
/*header file i.h*/
#define LEN 15
typedef struct customer_t{
    int    c_no;
    char   fname[LEN];
    char   lname[LEN];
    } CUST_REC;
:
:

/*cust.ec*/
#include "i.h"

int main()
{

EXEC SQL BEGIN DECLARE SECTION;
    CUST_REC cust;
:
:

}
:
:

...Leads to error –33051...
```

Figure 2-5. ESQL/C Excerpt That Defines a Host Variable Based on a C Header File Included Outside the Declare Section

Allowing All Valid C Declaration Syntax Inside the EXEC SQL Declare Section

When the IBM Informix ESQL/C preprocessor runs on a file, it expands all the contents of header files inside the Informix ESQL/C source file where the header file was included in the source file. Therefore, one consequence of including C header files inside the EXEC SQL declare section is that all types of C declaration syntax are included in the EXEC SQL declare section after the pass through the C preprocessor. You can now include all valid C declaration syntax in the EXEC SQL declare section in the EXEC SQL declare section. However, you can only declare host variables based on certain types described in "Host-Variable Data Types" on page 1-10.

Excluding Statements Inside C Header Files

If the Informix ESQL/C preprocessor has problems with certain statements in C header files, you can exclude specific lines from the preprocessing that the Informix ESQL/C preprocessor performs as shown in the example in Figure 2-6.

```
/*header file i.h*/
#ifndef ESBDS /*define empty macros, if included by a C\
   source*/
#define ESBDS
#define ESEDS
#endif
.
.

ESEDS
statement that you do not want ESQL/C  preprocessor to see
ESBDS


/*name.ec*/
#define ESBDS "EXEC SQL BEGIN DECLARE SECTION;"
#define ESEDS "EXEC SQL END DECLARE SECTION"
main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
   #include "i.h"
EXEC SQL END DECLARE SECTION;
.
.


}
```

Figure 2-6. ESQL/C and C Excerpt That Uses a Common C Header File

SQL Keyword Protection

If the code in the IBM Informix ESQL/C files is passed unprotected to the C
preprocessor before it is passed to the Informix ESQL/C preprocessor, certain SQL
keywords might be analyzed by the C preprocessor, which causes unintended
results. In the following example, the SQL keyword NULL is replaced by the C
preprocessor with the value zero, which creates a valid SQL statement, but one
which inserts a value into the **orders** table other than the value that the
programmer intended:

```
EXEC SQL insert into orders (shipcharge) values (NULL);
```

When a user gives the option to run the C preprocessor before the Informix
ESQL/C preprocessor, the utility **eprotect** runs before the C preprocessor runs on
the Informix ESQL/C source file. The **eprotect** utility adds a prefix to any SQL
keyword that occurs in an SQL statement with the prefix **SQLKEYWORD_**. This
prefix is affixed on all SQL keywords inside SQL statements that begin with the
EXEC SQL directive and end with a semicolon. When the Informix ESQL/C source
file that contains the select statement mentioned earlier is passed to the C
preprocessor, the SELECT statement has the following form:

```
EXEC SQL SQLKEYWORD_insert SQLKEYWORD_into orders (order_num)
SQLKEYWORD_values (SQLKEYWORD_NULL);
```

After the C preprocessor runs on an Informix ESQL/C source file, the **esql**
command runs the **eprotect** utility with the **-u** mode, which removes all the
SQLKEYWORD_ prefixes before it runs the Informix ESQL/C preprocessor on the
output of the C preprocessor.

SQL Keyword Protection and the Dollar Sign ($) Symbol

All SQL statements within IBM Informix ESQL/C source files can either begin with
the EXEC SQL key words or with the $ prefix. All of the following pairs of
statements are equivalent:

```
EXEC SQL BEGIN DECLARE SECTION;
$BEGIN DECLARE SECTION;

EXEC SQL connect to 'database9';
$connect to 'database9';

EXEC SQL select fname into :hostvar1 from table1;
$ select fname into :hostvar1 from table1;
```

However, the $ symbol can also occur at the beginning of **typedef** definitions such as in the following example:

```
$int *ip = NULL;
```

In cases such as the preceding **typedef** example, program logic might require that the C preprocessor substitute the value zero in the place of the keyword NULL. Not allowing the C preprocessor to make value substitutions in such cases would lead to errors. Therefore the **eprotect** utility does not add a prefix to the **SQLKEYWORD_** prefix on SQL keywords that are displayed in SQL statements that begin with the dollar sign ($) symbol.

Important: If you want to run the C preprocessor on your Informix ESQL/C source file before the Informix ESQL/C preprocessor, and if you do not want the C preprocessor to substitute values for the SQL keywords in SQL statements that occur in your source file, you must begin each SQL statement with the keywords EXEC SQL, and not with the dollar sign ($) symbol.

Preprocessor Options Specific to Windows Environments

The following additional preprocessing options are available to you if you use IBM Informix ESQL/C in a Windows environment.

Line Wrapping

The Informix ESQL/C preprocessor translates one embedded SQL statement as one C line. Long lines can cause problems for some debuggers and editors. You can use the **-lw** option to tell the preprocessor to wrap output lines at a specific column position. For example, the following **esql** command tells the preprocessor to wrap lines at column 75:

```
esql -lw:75 demo.ec
```

If you omit the **-lw** option, the preprocessor does not perform line wrapping.

Changing Error and Warning Displays

By default, the IBM Informix ESQL/C preprocessor generates error and warning messages when it processes an Informix ESQL/C file. It displays these errors and warnings in the console window. You can change the display of error and warning messages with the following command-line options:

- Use the **-nowarn** option to suppress warning messages. This option has no effect on error messages.
- Use the **-mserr** option to display error and warning messages in Microsoft Error Message format. Some text editors understand this format and can use it to go to the line in the Informix ESQL/C source file that caused the error or warning.

Setting Tab Stops

By default, the IBM Informix ESQL/C preprocessor formats the C source file with tab stops at every eighth column position. You can use the **-ts** option to set different tab stops. For example, the following **esql** command tells the preprocessor to set tab stops every four characters:

```
esql -ts:4 demo.ec
```

Compiling and Linking Options of the esql Command

The following sections describe the compiling and linking options of the **esql** command.

Naming the Executable

You can explicitly specify the name of the executable file with the **-o** option. For example, the following esql command produces an executable file called **inpt**:

```
esql -o inpt custinpt.ec ordinpt.ec
```

If **esql** is running on a Windows operating system the name of the target file defaults to the name of the first Informix ESQL/C source file on the esql command line. The extension is changed to either **.exe** or **.dll** depending on the type of target being generated.

If **esql** is running on a UNIX operating system the name of the target file defaults to whatever is the default for your C compiler, usually **a.out**.

Setting the Type of Executable Created (Windows)

The **esql** command can be used to compile regular executables and also Dynamic Link Libraries (DLLs). Use the **-target:** option to tell esql which type of output you want. The **-target:** option only tells **esql** how to compile your application. If you compile to a DLL, your source code must be written as a DLL or the compile will fail.

Pass options to the C compiler

The **esql** command processor passes any unrecognized arguments in the command line to the C compiler. For example, because **esql** does not recognize **/Zi** as an option, the following **esql** command passes the **/Zi** option to the C compiler:

```
esql /Zi demo1.ec
```

If you want to pass C compiler options that have the same names as IBM Informix ESQL/C processor options, precede them with the **-cc** option. The **esql** command ignores any options between the **-cc** and the next occurrence of any of these arguments:

- -l (Windows only)
- -r (Windows only)
- -f (Windows only)
- An Informix ESQL/C source file.

Specify a particular C compiler (Windows)

ESQL/C in Windows environments supports the following C compilers:

- Microsoft Visual C++, Version 2.x or later
- Borland C++, Version 5

Either the Microsoft C compiler or the Borland C compiler must be on your computer before you can compile an Informix ESQL/C program. The default C compiler option, **-mc**, starts the Microsoft compiler. To choose the Borland compiler, use the **-bc** option.

Compile without linking

By default, the IBM Informix ESQL/C command processor preprocesses, compiles, and links the Informix ESQL/C program and creates either an executable file or a DLL. To suppress the linking of your Informix ESQL/C program specify the **-c** option. With this option, **esql** only preprocesses and compiles the code. The output of this command is a C object file (**.obj** extension) for each C source file (**.c**) or Informix ESQL/C source file (**.ec**).

For example, to preprocess and compile the Informix ESQL/C source file **demo1.ec**, use the following command:

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

The preceding command generates a C object file called **demo1.obj**. The following **esql** command preprocesses **demo1.ec**, checks for Informix extensions to X/Open-standard syntax, and suppresses warning messages:

```
esql -c -xopen -nowarn demo1.ec
```

Important: If you specify the conflicting options **-c** and **-o**, the preprocessor ignores the **-o** option and processes the **-c** option. The preprocessor reports the conflict in an error message.

Special Compile Options for Windows Environments

You can give the following additional compile options to the **esql** command if you are running IBM Informix ESQL/C in a Windows environments.

Using a Project File

The **-f** option enables you to specify the name of a project file at the **esql** command line. The *filename* that follows **-f** is a project file that contains the names of the IBM Informix ESQL/C source (**.ec**) files to compile.

For example, suppose the project file, **project.txt**, contains the following lines:

```
first.ec
second.ec
third.ec
```

In this example, **first.ec**, **second.ec**, and **third.ec** represent the names of Informix ESQL/C source files that you want to compile.

The following **esql** command uses the **project.txt** project file to specify the three source files to compile and link:

```
esql -f project.txt
```

The preceding **esql** command is the equivalent of the following **esql** command:

```
esql first.ec second.ec third.ec
```

You can accomplish the same task with a response file. For more information about response files, see "Creating a Response File" on page 2-25.

Creating a Response File

You can specify the command-line arguments for the IBM Informix ESQL/C command processor in a *response file* and specify the file name for the Informix ESQL/C processor. The Informix ESQL/C response file is a text file with Informix ESQL/C command-line options and file names, separated by a space, a new line, a carriage return, a line feed, or a combination of these characters.

The following example shows the syntax that specifies a response file called **resp.esq**:

```
esql @resp.esq
```

The response file, **resp.esq**, might contain the following lines:

```
-we
first.ec
second.ec
third.ec
-r foo.rc
```

Use of this response file is the equivalent of the following **esql** command:

```
esql -we -f project.txt -r foo.rc
```

In this example, **project.txt** is a project file that contains the file names **first.ec**, **second.ec**, and **third.ec** on separate lines, as the previous shows.

You might use a response file for the following reasons:

- The command line is limited to 1,024 characters. If your **esql** options exceed this length, you must use a response file.
- If you use one or more sets of **esql** options regularly, you can save yourself a great deal of typing by putting each set in a different response file. Instead of typing the options, you can list the appropriate response file in the **esql** command.

Implicit Options Invoked by the esql Preprocessor in Windows Environments

The Informix ESQL/C command processor implicitly passes compiler and linker flags to the supported C compilers. Table 2-1 on page 2-26 lists the implicit options that **esql** passes when you use the indicated **esql** options. If you choose to create your own build file, use the indicated flags as appropriate for your application.

Important: The **esql** command does not implicitly pass any options to the resource compiler.

Table 2-1. Implicitly Passed Compiler Options

Compiler	Module Type	esql Options	Implicit Options Compiler	Linker
Microsoft Visual C++, Version 2.x or later	executable	-target:exe -we	-c -I%INFORMIXDIR%\ incl\esql /D_*systype* /D_*proctype* /*threadtype* /DWIN32	-DEF:deffile -OUT:target -MAP -SUBSYSTEM:*systype* %INFORMIXDIR%\lib\isqlt09a.lib %INFORMIXDIR%\lib\igl4g303.lib %INFORMIXDIR%\lib\iglxg303.lib %INFORMIXDIR%\lib\igo4g303.lib *libset*
	dll	-target:dll -wd	-c -I%INFORMIXDIR%\ incl\esql /D_*systype* /D_*proctype* /*threadtype* /DWIN32	-DLL -DEF:deffile -OUT:*target* -MAP -SUBSYSTEM:*systype* %INFORMIXDIR%\lib\isqlt09a.lib %INFORMIXDIR%\lib\igl4g303.lib %INFORMIXDIR%\lib\iglxg303.lib %INFORMIXDIR%\lib\igo4g303.lib *libset*
Borland C++, Version 5	executable	-target:exe -we	-c -I%INFORMIXDIR%\ incl\esql -e*target* -*subtype* -*libtlog* -*libtlg*	-c -Tpe -M -DEF:*deffile* -*subsystem* %INFORMIXDIR%\lib\igl4b303.lib %INFORMIXDIR%\lib\iglxb303.lib %INFORMIXDIR%\lib\igo4b303.lib c0*t*32.obj *libset*
	dll	-target:dll -wd	-c -I%INFORMIXDIR%\ incl\esql -e*target* -*subtype* -*libtlog* -*libtlg*	-c -Tpd -M -DEF:*deffile* -*subsystem* %INFORMIXDIR%\lib\igl4b303.lib %INFORMIXDIR%\lib\iglxb303.lib %INFORMIXDIR%\lib\igo4b303.lib c0d32.obj *libset*

The italicized terms in the compiler and linker options represent the following definitions.

Term	Definition
deffile	Name of a **.def** file (The -DEF option executes only if you specify a **.def** file on the command line.)
libset	Library set (depends on whether the application is WINDOWS or CONSOLE).
libtlg	-D_RTLDLL for a dynamic library or " " for a shared library
libtlog	-WM for a multithread library or " " for a single-thread library
proctype	Type of processor (X86)
subsystem	ap for a console subsystem or aa for a Windows subsystem
subtype	WC for a console executable, W for a Windows executable file, WCD for a console DLL, or WD for a Windows DLL
systype	Type of subsystem (WINDOWS or CONSOLE)
t	X for a console subsystem and W for a Windows subsystem
target	Name of the executable file (name of first **.ec** file or the name specified by the **-o** command-line option)

threadtype   Type of thread option (ML, MT, MD, depending on the value of the **-runtime** command-line option)

For more information about the **-target**, **-wd**, and **-we** command-line options, see "Syntax of the esql command" on page 2-5

The library set that the linker uses depends on whether you are creating a Windows or console application. Table 2-2 lists the library sets that the indicated esql options use.

Table 2-2. Library Sets That the Linker Uses

Compiler	esql Options	Library Sets That the Linker Uses
Microsoft Visual C++, Version 2.x or later	-subsystem:windows -Sw -ss:w	advapi32.lib wsock32.lib user32.lib winmm.lib gdi32.lib comdlg32.lib winspool.lib
	-subsystem:console -Sc -ss:c	netapi32.lib wsock32.lib user32.lib winmm.lib
Borland C++, Version 5	-subsystem:windows -Sw -ss:w	cw32mti.lib import32.lib
	-subsystem:console -Sc -ss:c	cw32mti.lib import32.lib

Linking Options

The C compiler performs the linking phase of an IBM Informix ESQL/C compile. This section describes the **esql** command-line arguments that affect how this linking occurs.

General Linking Options

The following linking options affect both UNIX and Windows environments:
- Linking other C source and object files
- Specifying versions of IBM Informix general libraries

The following sections describe how to specify these options.

Linking Other C Source and Object Files: You can list the following types of files on the **esql** command line to indicate that you want the link editor to link to the resulting object file:

- C source files in the form *otherCsrc*.**c**

  If you list files with the **.c** extensions, **esql** passes them through to the C compiler, which compiles them to object files (**.o** extensions) and links these object files.

- C object files in the form *otherCobj*.**o** on a UNIX operating system or *otherCobj*.**obj** in a Windows environment

If you list files with **.o** or **.obj** extensions, **esql** passes them through to the C
compiler, which links these object files. The link editor links the C object files
with the appropriate IBM Informix ESQL/C library functions.

- Library files, either your own libraries or system libraries that are compatible
  with the linker
- Module definitions (.**def**)
- Resource files, either compiled (**.res**) or uncompiled (**.rc**)

> **Tip:** If you specify uncompiled resource files, **esql** passes them to the resource
> compiler and links the resulting **.res** file to the Informix ESQL/C
> application.

The Informix ESQL/C command preprocessor passes these files directly to the
linker. It also links the libraries it needs to support the Informix ESQL/C function
library. You can use the **-libs** option to determine which libraries **esql**
automatically links, as follows:

```
esql -libs
```

Specifying Versions of Informix General Libraries: By default, the **esql**
command links the shared libraries for the IBM Informix general libraries: **libgen**,
libos, **libgls**, **libafs**, and **libsql**. To use shared libraries, your computer must
support shared memory.

You can use the following command-line options to change which versions of the
Informix general libraries the preprocessor links with your program:

- The **-thread** option tells the preprocessor to link the thread-safe versions of the
  Informix shared libraries.
- The **-static** option tells the preprocessor to link the static libraries for the
  Informix general libraries in a UNIX environment. If you use the **-static** option,
  you cannot set the **IFX_LONGID** environment variable. You must recompile
  with **libos.a**.

You can combine these options to tell the preprocessor to link in the thread-safe
versions of the Informix static libraries. For more information about this topic, see
"The esql command" on page 13-3.

Special Linking Options for Windows

The following sections give linking options that you can only use in Windows
environments.

Passing Arguments to the Linker: On the **esql** command line, you can list linker
arguments by prefacing them with the **-l** processor option. The **esql** command
processor passes to the linker all arguments after the **-l** option, up to whichever of
the following items it encounters first:

- The **-r** option to specify resource compiler options
- The end of the command line

Passing Arguments to the Resource Compiler: On the **esql** command line, you
can list resource compiler arguments by prefacing them with the **-r** processor
option. The IBM Informix ESQL/C command processor passes to the resource
compiler all arguments after the **-r**, up to the end of the command line. The
processor then runs the resource compiler to create a **.res** file, which it then passes
to the linker. If you specify the **-r** option but do not specify an associated *resfile***.rc**,
esql uses the name for the target and appends the **.rc** extension.

ESQL/C Dynamic Link Libraries: For Windows environments, the IBM Informix ESQL/C product includes the following dynamic link libraries (DLLs):

- The ESQL client-interface DLL (**isqlt09a.dll**) contains the Informix ESQL/C library functions that the Informix ESQL/C preprocessor needs to translate embedded SQL statements and other internal functions that are needed at runtime. For more information see "Accessing the ESQL Client-Interface DLL in Windows Environments" on page 2-30.
- The **esqlauth.dll** DLL provides runtime verification of the connection information that the client application sends to the database server. When your application requests a connection, Informix ESQL/C calls the **sqlauth()** function, which **esqlauth.dll** defines. For more information about **sqlauth()**, see "Connection Authentication Functionality in a Windows Environment" on page 12-8.
- The Registry DLL, **iregt07b.dll**, is used by the **Setnet32** utility and the IBM Informix Connect library to set and access configuration information in the Registry.
- The **igl4b304.d ll**, **igo4g303.dll**, and **iglxg303.dll** DLLs are required for Global Language Support (GLS). For more information about code-set conversion, see the *IBM Informix GLS User's Guide*.

Informix DLLs are located in the **%INFORMIXDIR%\bin** directory. **%INFORMIXDIR%** is the value of the **INFORMIXDIR** environment variable.

Using the Same Runtime Routines for Version Independence: If your application was compiled with a version of Microsoft Visual C++ earlier than 4.x, you must export your C runtime library to the ESQL client-interface DLL (**isqlt09a.dll**). The ESQL client-interface DLL uses your runtime routines to make sure all the pieces of your application are compiled with the same runtime version. Any application that is linked to your application and calls IBM Informix ESQL/C library routines or SQL statements must also use your C runtime library.

To export a C runtime library, include the following line in your code before the first call to an Informix ESQL/C library routine or SQL statement:

```
#include "infxcexp.c";
```

The **infxcexp.c** file contains the C code to export the addresses of all C runtime routines that the ESQL client-interface DLL uses. This file is in the **%INFORMIXDIR%\incl\esql** directory, which the **esql** command processor automatically searches when it compiles a program. If you do not use the esql command processor, add the **%INFORMIXDIR%\incl\esql** directory to the compiler search path before you compile.

You must include the **infxcexp.c** file only once, in the **main()** routine (once per process), before the first Informix ESQL/C library call or SQL statement in the program. The code in this file exports your runtime library to the ESQL runtime DLL (**isqlt09a.dll**) so that they use the same C runtime code. Exporting your runtime routines enables the ESQL runtime routines to allocate memory (**malloc()**), return the pointer to a C program, and let the program free the memory (**free()**). It also enables a C program to open a file and to pass the handle (or file pointer) to the ESQL runtime routines for read-write access.

Accessing the ESQL Client-Interface DLL in Windows Environments

A DLL is a collection of functions and resources that can be shared by applications. It is similar to a runtime library in that it stores functions that many applications need. It differs, however, from a runtime library in the way that it is linked to the calling application.

Libraries that are linked at compile time are *static-link libraries*. The libraries such as **libc** and **libcmt** (used with the Microsoft Visual C++, Version 2.x) are static-link libraries. Whenever you link one of these Microsoft Visual C++ (Version 2.x) libraries to your application, the linker copies the code from the appropriate static-link library to the executable file (**.exe**) for your application. By contrast, when you link dynamically, no code is copied to the executable file of your application. Instead, your functions are linked at runtime.

Static-link libraries are effective in an environment where no multitasking is required. However, they become inefficient when more than one application calls the same functions. For example, if two applications that are running simultaneously in a Windows environment call the same static-link function, two copies of the function is in memory. This situation is inefficient.

But if a function is dynamically linked, the Windows system first checks memory to see if a copy of the function already is there. If a copy exists, the Windows system uses that copy rather than making another copy. If the function does not yet exist in memory, the Windows system links or copies the function into memory from the DLL.

The IBM Informix ESQL/C library functions, and other internal functions, are contained in the ESQL client-interface DLL. To use these functions in your Informix ESQL/C application, you must perform the following tasks:

- Access the import library for the ESQL client-interface DLL
- Locate the ESQL client-interface DLL

Accessing the Import Library

The *import* library of the DLL is provided to enable your IBM Informix ESQL/C application to access the ESQL client-interface DLL. The linker uses an import library to locate functions that are contained in the DLL. It contains references that reconcile function names used in an application with the library module that contains the function.

When you link a static library to your application, the linker copies program code from your static-link libraries to the executable file. However, if you link an import library to your application, the linker does not copy the program code when it links the executable file. Instead, the linker stores the information needed to locate the functions in the DLL. When you run your application, this location information serves as a dynamic link to the DLL.

The ESQL client-interface library provides location information for the Informix ESQL/C function calls. The **esql** command processor automatically links the import and Windows libraries for the DLL whenever you use it to compile and link your Informix ESQL/C program.

Locating a DLL

During the development of your application, the IBM Informix ESQL/C software (such as the **esql** command processor) must be able to access object libraries and import libraries. However, DLLs must be accessible when the application is running. Consequently, Windows must be able to locate them on your hard disk.

Search directories for your DLL in the following order:
1. The directory from which you loaded the application
2. The Windows environment system directory, **SYSTEM**
3. The current directory (where the executable file exists or the working directory that the Program Item Properties value for the icon specifies)
4. Directories that your **PATH** environment variable lists

Note: For the most recent information about your particular Windows operating system, see the Dynamic-Link Library Search Order documentation at www.microsoft.com.

Building an Application DLL

You can tell the IBM Informix ESQL/C processor to build an Informix ESQL/C program as a DLL (**.dll** file) with the **-target** (or **-wd**) command-line option. Such an Informix ESQL/C program is called an *application DLL*.

To build an Informix ESQL/C program as a DLL, follow the guidelines for general-purpose DLLs. For more information, see your system documentation. Compile the Informix ESQL/C source file with the **-target:dll** (or **-wd**) to create the application DLL.

For an example of how to build an application DLL, see the WDEMO demonstration program in the **%INFORMIXDIR%\demo\wdemo** directory. The Informix ESQL/C source file for the sample application DLL is called **wdll.ec**. To compile this DLL, use the following **esql** command:

```
esql –subsystem:windows –target:dll wdll.ec
```

The source code for the WDEMO executable file is in the **wdemo.exe** file.

Chapter 3. Informix ESQL/C data types

These topics contain information about the correspondence between SQL and C data types and how to handle data types in an IBM Informix ESQL/C program. These topics contain the following information:

- Choosing the appropriate data type for a host variable
- Converting from one data type to another
- Functions for working with nulls and different data types

Choosing data types for host variables

When you access a database column in your Informix ESQL/C program, you must declare a host variable of the appropriate C or Informix ESQL/C data type to hold the data. Table 3-1 on page 3-1 lists the SQL data types of the IBM Informix and the corresponding Informix ESQL/C data types that you can declare for host-variables. Table 3-2 on page 3-3 lists the additional SQL data types available with IBM Informix and the Informix ESQL/C data types that you can use as host variables for those types of columns. Both figures include a reference to the section or chapter in this book where you can obtain more information about the host-variable data type. For more information about the SQL data types that you can assign to database columns, see the *IBM Informix Guide to SQL: Reference*.

Table 3-1. Corresponding SQL and Host Variable Data Types

SQL Data Type	ESQL/C Predefined Data Type	C Language Type	See
BIGINT	**BIGINT**	8-byte integer	Chapter 5, "Working with numeric data types," on page 5-1
BIGSERIAL	**BIGINT**	8-byte integer	Chapter 5, "Working with numeric data types," on page 5-1
BOOLEAN	**boolean**		Table 3-8 on page 3-8
BYTE	**ifx_loc_t** or **loc_t**		Chapter 7, "Working with simple large objects," on page 7-1

Table 3-1. Corresponding SQL and Host Variable Data Types (continued)

SQL Data Type	ESQL/C Predefined Data Type	C Language Type	See
CHAR(*n*) CHARACTER(*n*)	**fixchar** [*n*] or **string** [*n*+1]	char [*n* + 1] or **char** *	Chapter 4, "Working with character and string data types," on page 4-1
DATE	**date**	4-byte integer	Chapter 6, "Working with time data types," on page 6-1
DATETIME	**datetime** or **dtime_t**		Chapter 6, "Working with time data types," on page 6-1
DECIMAL DEC NUMERIC MONEY	**decimal** or **dec_t**		Chapter 5, "Working with numeric data types," on page 5-1
FLOAT DOUBLE PRECISION		**double**	Chapter 6, "Working with time data types," on page 6-1
INT8	**int8** or **ifx_int8_t**		Chapter 5, "Working with numeric data types," on page 5-1
INTEGER INT		4-byte integer	Chapter 5, "Working with numeric data types," on page 5-1
INTERVAL	**interval** or **intrvl_t**		Chapter 6, "Working with time data types," on page 6-1
LVARCHAR	**lvarchar**	char [*n* + 1] or **char** *	Chapter 4, "Working with character and string data types," on page 4-1
NCHAR(*n*)	**fixchar** [*n*] or **string** [*n*+1]	char [*n* + 1] or **char** *	Chapter 4, "Working with character and string data types," on page 4-1
NVARCHAR(*m*)	**varchar**[*m*+1] or **string** [*m*+1]	char [*m*+1]	Chapter 4, "Working with character and string data types," on page 4-1
SERIAL		4-byte integer	Chapter 5, "Working with numeric data types," on page 5-1
SERIAL8	**int8** or **ifx_int8_t**		Chapter 5, "Working with numeric data types," on page 5-1
SMALLFLOAT REAL		**float**	Chapter 5, "Working with numeric data types," on page 5-1
SMALLINT		2-byte integer	Chapter 5, "Working with numeric data types," on page 5-1
TEXT	**loc_t**		Chapter 7, "Working with simple large objects," on page 7-1
VARCHAR(*m,x*)	**varchar**[*m*+1] or **string** [*m*+1]	char d[*m*+1]	Chapter 4, "Working with character and string data types," on page 4-1

Table 3-2. Corresponding SQL and Host Variable Data Types Specific to IBM Informix

SQL Data Type	ESQL/C Predefined Data Type	See
BLOB	**ifx_lo_t**	Chapter 8, "Working with smart large objects," on page 8-1
CLOB	**ifx_lo_t**	Chapter 8, "Working with smart large objects," on page 8-1
LIST(*e*)	**collection**	Chapter 8, "Working with smart large objects," on page 8-1
MULTISET(*e*)	**collection**	Chapter 9, "Working with complex data types," on page 9-1
Opaque data type	**lvarchar**, **fixed binary**, or **var binary**	Chapter 10, "Working with opaque data types," on page 10-1
ROW(...)	**row**	Chapter 9, "Working with complex data types," on page 9-1
SET(*e*)	**collection**	Chapter 9, "Working with complex data types," on page 9-1

Data Type Constants

The IBM Informix ESQL/C **sqltypes.h** header file contains integer constants for both SQL and Informix ESQL/C data types. Some Informix ESQL/C library functions require data type constants as arguments. You can also compare these data type constants in dynamic SQL programs to determine the type of column that the DESCRIBE statement described. The Informix ESQL/C code excerpt in Figure 3-1 compares the **sqltype** element of an **sqlvar** structure to a series of SQL data type constants to determine what types of columns a DESCRIBE statement returned.

```
for (col = udesc->sqlvar, i = 0; i < udesc->sqld; col++, i++)
   {
   switch(col->sqltype)
      {
      case SQLSMFLOAT:
      col->sqltype = CFLOATTYPE;
      break;

      case SQLFLOAT:
        col->sqltype = CDOUBLETYPE;
        break;

      case SQLMONEY:
      case SQLDECIMAL:
        col->sqltype = CDECIMALTYPE;
        break;

      case SQLCHAR:
        col->sqltype = CCHARTYPE;
        break;

      default:
        /*  The program does not handle INTEGER,
         *  SMALL INTEGER, DATE, SERIAL or other
         *  data types.  Do nothing if we see
         *  an unsupported type.
         */
        return;
        }
```

Figure 3-1. Code Excerpt Using SQL Data Type Constants

For more information about the use of data type constants with the DESCRIBE statement, see Chapter 15, "Determining SQL statements," on page 15-1.

SQL Data Type Constants

Table 3-3 shows the SQL data type constants for the IBM Informix. Table 3-4 on page 3-5 shows the SQL data type constants for the additional data types that are available with the IBM Informix.

Table 3-3. Constants for Informix SQL Column Data Types

SQL Data Type	Defined Constant	Integer Value
CHAR	SQLCHAR	0
SMALLINT	SQLSMINT	1
INTEGER	SQLINT	2
FLOAT	SQLFLOAT	3
SMALLFLOAT	SQLSMFLOAT	4
DECIMAL	SQLDECIMAL	5
SERIAL	SQLSERIAL	6
DATE	SQLDATE	7
MONEY	SQLMONEY	8
DATETIME	SQLDTIME	10
BYTE	SQLBYTES	11
TEXT	SQLTEXT	12
VARCHAR	SQLVCHAR	13
INTERVAL	SQLINTERVAL	14
NCHAR	SQLNCHAR	15
NVARCHAR	SQLNVCHAR	16
INT8	SQLINT8	17
BIGSERIAL	SQLBIGSERIAL	53
LVARCHAR	SQLLVARCHAR	43
BOOLEAN	SQLBOOL	45
BIGINT	SQLINFXBIGINT	52
BIGSERIAL	SQLBIGSERIAL	53

Table 3-4. Constants for Informix SQL Column Data Types That Are Specific to IBM Informix

SQL Data Type	Defined Constant	Integer Value
SET	SQLSET	19
MULTISET	SQLMULTISET	20
LIST	SQLLIST	21
ROW	SQLROW	22
Varying-length opaque type	SQLUDTVAR	40
Fixed-length opaque type	SQLUDTFIXED	41
SENDRECV (client-side only)	SQLSENDRECV	44

Important: The SENDRECV data type has an SQL constant but can only be used in an Informix ESQL/C program. You cannot define a database column as type SENDRECV.

ESQL/C Data Type Constants

You assign an IBM Informix ESQL/C data type to a host variable in an Informix ESQL/C program. The following table shows these constants.

Table 3-5. Constants for ESQL/C Host-Variable Data Types

ESQL/C Data Type	Constant	Integer Value
char	CCHARTYPE	100
short int	CSHORTTYPE	101
int4	CINTTYPE	102
long	CLONGTYPE	103
float	CFLOATTYPE	104
double	CDOUBLETYPE	105
dec_t or decimal	CDECIMALTYPE	107
fixchar	CFIXCHARTYPE	108
string	CSTRINGTYPE	109
date	CDATETYPE	110
dec_t or decimal	CMONEYTYPE	111
datetime or dtime_t	CDTIMETYPE	112
ifx_loc_t or loc_t	CLOCATORTYPE	113
varchar	CVCHARTYPE	114
intrvl_t or interval	CINVTYPE	115
char	CFILETYPE	116
int8	CINT8TYPE	117
collection	CCOLTYPE	118
lvarchar	CLVCHARTYPE	119
fixed binary	CFIXBINTYPE	120
var binary	CVARBINTYPE	121
boolean	CBOOLTYPE	122
row	CROWTYPE	123

You can use these Informix ESQL/C data types as arguments for some of the functions in the Informix ESQL/C library. For example, both the **rtypalign()** and **rtypmsize()** functions require data type values as arguments.

X/Open Data Type Constants

If your programs conform to the X/Open standards (compile with the **-xopen** option), you must use the data type values that the following table shows. IBM Informix defines the constants for these values in the **sqlxtype.h** header file.

Table 3-6. Constants for Informix SQL Column Data Types in an X/Open Environment

SQL Data Type	Defined Constant	X/Open Integer Value
CHAR	XSQLCHAR	1
DECIMAL	XSQLDECIMAL	3
INTEGER	XSQLINT	4
SMALLINT	XSQLSMINT	5
FLOAT	XSQLFLOAT	6

Header Files for Data Types

To use an SQL data type, your program must include the appropriate IBM Informix ESQL/C header file. Table 3-7 shows the relationship between host-variable data types and Informix ESQL/C header files for all database servers. Table 3-8 on page 3-8 shows the relationship between host-variable data types and Informix ESQL/C header files that are specific to IBM Informix with Universal Data Option.

Table 3-7. SQL Data Types and ESQL/C Header Files

SQL Data Type	ESQL/C or C Data Type	ESQL/C Header File
BLOB	**ifx_lo_t**	**locator.h**
BOOLEAN	**boolean**	Defined automatically
BYTE	**ifx_loc_t** or **loc_t**	**locator.h**
CHAR(*n*) CHARACTER(*n*)	**fixchar array[*n*]** or **string array[*n*+1]**	Defined automatically
DATE	**date**	Defined automatically
DATETIME	**datetime** or **dtime_t**	**datetime.h**
DECIMAL DEC NUMERIC MONEY	**decimal** or **dec_t**	**decimal.h**
FLOAT DOUBLE PRECISION	**double**	Defined automatically
INT8	**int8**	**int8.h**
INTEGER INT	4-byte integer	Defined automatically
INTERVAL	**interval** or **intrvl_t**	**datetime.h**
LVARCHAR	**lvarchar array[*n + 1*]** where *n* is the length of the longest string that might be stored in the LVARCHAR field.	Defined automatically
MULTISET(e)	**collection**	Defined automatically
NCHAR(*n*)	**fixchar array[*n*]** or **string array[*n*+1]**	Defined automatically
NVARCHAR(*m*)	**varchar[*m*+1]** *or* **string array[*m*+1]**	Defined automatically
SERIAL	4-byte integer	Defined automatically
SERIAL8	**int8**	**int8.h**
BIGINT	BIGINT	Defined automatically
BIGSERIAL	BIGINT	Defined automatically
SMALLFLOAT REAL	**float**	Defined automatically
SMALLINT	**short int**	Defined automatically
TEXT	**loc_t**	**locator.h**
VARCHAR(*m,x*)	**varchar[*m*+1]** or **string array[*m*+1]**	Defined automatically

Table 3-8. SQL Data Types and ESQL/C Header Files That Are Specific to IBM Informix

SQL Data Type	ESQL/C or C Data Type	ESQL/C Header File
BLOB	**ifx_lo_t**	**locator.h**
CLOB	**ifx_lo_t**	**locator.h**
LIST(*e*)	**collection**	Defined automatically
Opaque data type	**lvarchar** or **fixed binary** or **var binary**	User-defined header file that contains definition of internal structure for opaque type
ROW(...)	**row**	Defined automatically
SET(e)	**collection**	Defined automatically

Data Conversion

When a discrepancy exists between the data types of two values, IBM Informix ESQL/C attempts to convert one of the data types. The process of converting a value from one data type to another is called *data conversion*.

The following list names a few common situations in which data conversion can occur:

- **Comparison.** Data conversion can occur 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.

  For example, to compare a CHAR value and a numeric value, Informix ESQL/C converts the CHAR value to a numeric value before it performs the comparison.

- **Fetching and inserting.** Data conversion can occur if you fetch or insert values with host variables and database columns of different data types.

- **Arithmetic operations.** Data conversion can occur if a numeric value of one data type operates on a value of a different data type.

For more information about fetching and inserting, see "Fetching and Inserting with Host Variables." For more information about arithmetic operations, see "Performing Arithmetic Operations" on page 3-10.

Fetching and Inserting with Host Variables

If you try to *fetch* a value from a database column into a host variable that you do not declare according to the correspondence shown in Table 3-1 on page 3-1, IBM Informix ESQL/C attempts to convert the data types. Similarly, if you try to *insert* a value from a host variable into a database column, Informix ESQL/C might need to convert data types if the host variable and database column do not use the correspondences in Table 3-1. Informix ESQL/C converts the data types only if the conversion is meaningful.

This section provides the following information about data conversion for fetching and inserting values with host variables:

- How Informix ESQL/C converts between numeric and character data
- How Informix ESQL/C converts floating-point numbers to strings
- How Informix ESQL/C converts BOOLEAN values to characters
- How Informix ESQL/C converts DATETIME and INTERVAL values
- How Informix ESQL/C converts between VARCHAR columns and character data

Converting Numbers and Strings

Before IBM Informix ESQL/C can convert a value from one data type to another, it must determine whether the conversion is meaningful. Table 3-9 shows possible conversions between numeric data types and character data types. In this figure, `N` represents a value with a numeric data type (such as DECIMAL, FLOAT, or SMALLINT) and `C` represents a value with a character data type (such as CHAR or VARCHAR).

If conversion is not possible, either because it makes no sense or because the target variable is too small to accept the converted value, Informix ESQL/C returns values that the **Results** column in Table 3-9 describes.

Table 3-9. Data Conversion Problems and Results

Conversion	Problem	Results
C Æ C	Does not fit	Informix ESQL/C truncates the string, sets a warning (**sqlca.sqlwarn.sqlwarn1** to `W` and SQLSTATE to `01004`), and sets any indicator variable to the size of the original string. For more information, see "Fetching and Inserting Character Data Types" on page 4-7.
N Æ C	None	Informix ESQL/C creates a string for the numeric value; it uses an exponential format for very large or very small numbers. For more information, see "Converting Floating-Point Numbers to Strings" on page 3-10.
N Æ C	Does not fit	Informix ESQL/C fills the string with asterisks, sets a warning (**sqlca.sqlwarn.sqlwarn1** to `W` and SQLSTATE to `01004`), and sets any indicator variable to a positive integer. When the fractional part of a number does not fit in a character variable, Informix ESQL/C rounds the number. Asterisks are displayed only when the integer part does not fit.
C Æ N	None	Informix ESQL/C determines the numeric data type based on the format of the character value; if the character contains a decimal point, Informix ESQL/C converts the value to a DECIMAL value.
C Æ N	Not a number	The number is undefined; Informix ESQL/C sets **sqlca.sqlcode** and SQLSTATE to indicate a runtime error.
C Æ N	Overflow	The number is undefined; Informix ESQL/C sets **sqlca.sqlcode** and SQLSTATE to indicate a runtime error.
N Æ N	Does not fit	Informix ESQL/C attempts to convert the number to the new data type. For information about possible errors, see the *IBM Informix Guide to SQL: Reference*.
N Æ N	Overflow	The number is undefined; Informix ESQL/C sets **sqlca.sqlcode** and SQLSTATE to indicate a runtime error.

In Table 3-9, the phrase *Does not fit* means that the size of the data from the source variable or column exceeds the size of the target column or variable.

Converting Floating-Point Numbers to Strings

IBM Informix ESQL/C can automatically convert floating-point column values (data type of DECIMAL(*n*), FLOAT, or SMALLFLOAT) between database columns and host variables of character type **char**, **varchar**, **string**, or **fixchar**. When Informix ESQL/C converts a floating-point value to a character string whose buffer is not large enough to hold the full precision, Informix ESQL/C rounds the value to fit it in the character buffer. For more information, see "Implicit Data Conversion" on page 5-8.

Converting BOOLEAN Values to Characters

The database server can automatically convert BOOLEAN values between database columns and host variables of the **fixchar** date type. The following table shows the character representations for the BOOLEAN values.

BOOLEAN Value	Character Representation
'\01'	'T'
'\00'	'F'

For more information about the BOOLEAN data type, see Chapter 5, "Working with numeric data types," on page 5-1.

Converting DATETIME and INTERVAL Values

IBM Informix ESQL/C can automatically convert DATETIME and INTERVAL values between database columns and host variables of character type **char**, **string**, or **fixchar**. Informix ESQL/C converts a DATETIME or INTERVAL value to a character string and then stores it in a host variable. For more information, see "Implicit Data Conversion" on page 6-7.

You can use Informix ESQL/C library functions to explicitly convert between DATE and DATETIME values. For more information, see "Data conversion for datetime and interval values" on page 6-8.

Converting Between VARCHAR and Character Data Types

IBM Informix ESQL/C can automatically convert VARCHAR values between database columns and host variables of character type **char**, **string**, or **fixchar**. For more information, see "Fetching and Inserting VARCHAR Data" on page 4-8.

Performing Arithmetic Operations

When IBM Informix ESQL/C performs an arithmetic operation on two values, it might need to convert data types if the two values do not have data types that match. This section provides the following information about data conversion for arithmetic operations:

- How Informix ESQL/C converts numeric values
- How Informix ESQL/C handles operations that involve floating-point values

Converting Numbers to Numbers

If two values of different numeric data types operate on one another, Informix ESQL/C converts the values to the data type that Table 3-10 indicates and then performs the operation.

Table 3-10. Data Types for Which ESQL/C Carries Out Numeric Operations

Operands	DEC	FLOAT	INT	SERIAL	SMALLFLOAT	SMALLINT
DEC	DEC	DEC	DEC	DEC	DEC	DEC
FLOAT	DEC	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
INT	DEC	FLOAT	INT	INT	FLOAT	INT
SERIAL	DEC	FLOAT	INT	INT	FLOAT	INT
SMALLFLOAT	DEC	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
SMALLINT	DEC	FLOAT	INT	INT	FLOAT	INT

Table 3-10 shows that if Informix ESQL/C performs an operation between an operand with a data type of FLOAT and a second operand with a data type of DECIMAL (DEC), Informix ESQL/C generates a result that has a DECIMAL data type. For more information about DECIMAL values, see the next section, "Using Operations That Involve a Decimal Value."

Using Operations That Involve a Decimal Value

The following table shows the numeric data types. Database columns use the SQL data types, and Informix ESQL/C host variables use the corresponding Informix ESQL/C data types.

SQL Data Type	ESQL/C Data Type
INTEGER	4-byte integer
SMALLINT	short integer
DECIMAL	decimal
MONEY	decimal
FLOAT	double
SMALLFLOAT	float

When Informix ESQL/C performs arithmetic operations on operands with numeric data types and one of the operands has a decimal value (an SQL data type of DECIMAL or an Informix ESQL/C data type of **decimal**), Informix ESQL/C converts each operand and the result to a decimal value.

An SQL DECIMAL data type has the format DECIMAL(p,s), where p and s represent the following parameters:

- The *p parameter is the precision, which* is the total number of significant digits in a real number.

  For example, the number 1237.354 has a precision of seven.

- The *s parameter is the scale, which* is the number of digits that represent the fractional part of the real number.

  For example, the number 1237.354 has a scale of three. If the DECIMAL data type includes a scale parameter (DECIMAL(p,s)), it holds fixed-point decimal numbers. If the DECIMAL data type omits a scale parameter (DECIMAL(p)), it holds floating-point decimal numbers.

The Informix ESQL/C **decimal** data type keeps track of precision and scale differently from the SQL DECIMAL data type. For simplicity, this section uses the format of the SQL DECIMAL data type to describe how Informix ESQL/C performs data conversion for arithmetic operations that involve a decimal value.

However, this same data-conversion information applies to arithmetic operations that involve an Informix ESQL/C **decimal** host variable.

For more information about how the **decimal** data type tracks precision and scale, see "The decimal structure" on page 5-5.

Converting the Non-DECIMAL Numeric Operand: Informix ESQL/C converts all operands that are not already DECIMAL (or **decimal**) to DECIMAL before it performs the arithmetic operation. The following table shows the precision and scale that Informix ESQL/C uses for the non-DECIMAL operand.

Operand Type	Convert To
FLOAT	DECIMAL(17)
SMALLFLOAT	DECIMAL(9)
INTEGER	DECIMAL(10,0)
SMALLINT	DECIMAL(5,0)

Informix ESQL/C does not consider leading or trailing zeros as significant digits. Leading or trailing zeros do not contribute to the determination of precision and scale. If the operation is addition or subtraction, Informix ESQL/C adds trailing zeros to the operand with the smaller scale until the scales are equal.

Obtaining the DECIMAL Data Type of the Arithmetic Result: The precision and scale of the arithmetic result depend on the precision and scale of the operands and on whether one of the operands is a floating-point decimal, as follows:

- When one of the operands is a floating-point decimal, the arithmetic result is a floating-point decimal.

  For example, for an arithmetic operation between a fixed-point decimal of DECIMAL(8,3) and a FLOAT value, Informix ESQL/C converts the FLOAT value to a floating-point decimal of DECIMAL(17). The arithmetic result has a data type of DECIMAL(17).

- When both of the operands are fixed-point decimals, the arithmetic result is also a fixed-point decimal.

  Table 3-11 summarizes the rules for arithmetic operations on operands with definite scale (fixed-point decimals). In Table 3-11, p_1 and s_1 are the precision and scale of the first operand, and p_2 and s_2 are the precision and scale of the second operand.

Table 3-11. Precision and Scale of Fixed-Decimal Arithmetic Results

Operation	Precision and Scale of Result	
Addition and Subtraction	Precision: Scale:	$MIN(32, MAX(p_1 - s_1, p_2 - s_2) + MAX(s_1, s_2) + 1)$ $MAX(s_1, s_2)$
Multiplication	Precision: Scale:	$MIN(32, p_1 + p_2)$ $s_1 + s_2$; If $(s_1 + s_2) >$ precision, the result is a floating-point decimal number (no scale value).
Division	Precision: Scale:	32 Result is a floating-point decimal number. The sum: $32 - p_1 + s_1 - s_2$ cannot be negative.

If the data type of the result of an arithmetic operation requires the loss of significant digits, Informix ESQL/C reports an error.

Data-Type Alignment Library Functions

The following Informix ESQL/C library functions provide machine-independent size and alignment information for different data types and help you work with null database values.

Function Name	Description	See
The risnull() function)	Checks whether a C variable is null	B-191
The rsetnull() function	Sets a C variable to null	B-197
The rtypalign() function	Aligns data on correct type boundaries	B-206
The rtypmsize() function	Gives the byte size of SQL data types	B-209
The rtypname() function	Returns the name of a specified SQL data type	B-211
The rtypwidth() function	Returns the minimum number of characters that a character data type needs to avoid truncation	B-213

When you compile your Informix ESQL/C program with the **esql** command, **esql** calls on the linker to link these functions to your program. The library functions are located in Appendix B, "ESQL/C function library," on page B-1.

Chapter 4. Working with character and string data types

These topics explain how to use character data types in an IBM Informix ESQL/C program. The topics contain the following information:

- An overview of the character data types
- Some issues to consider when you insert data from character host variables into the database
- The syntax of Informix ESQL/C library functions that you can use to manipulate the character data type

For information about all the data types available for use in an Informix ESQL/C program, see Chapter 3, "Informix ESQL/C data types," on page 3-1. For information about SQL data types, see the *IBM Informix Guide to SQL: Reference*.

Character data types

Informix ESQL/C supports five data types that can hold character data that you retrieve from and send to the database. If you use a character data type (such as the SQL data types CHAR and VARCHAR) for your database column, you can choose any of the following data types for your host variable:

- The C character data type: **char**
- One of the Informix ESQL/C predefined data types: **fixchar**, **string**, **varchar**
- The **lvarchar** data type

If you use locale-sensitive character data types (NCHAR or NVARCHAR), you have the same choice of character data types for your associated host variables. For more information about how to declare host variables for the NCHAR and NVARCHAR data types, see the *IBM Informix GLS User's Guide*.

The following two conditions determine which character data type to use:

- Whether you want Informix ESQL/C to terminate the character data with the null character
- Whether you want Informix ESQL/C to pad the character data with trailing blanks

Table 4-1 summarizes the attributes of each of the character data types.

Table 4-1. ESQL/C Character Data Types

ESQL/C Character Data Type	Null Terminated	Contains Trailing Blanks
char	Y	Y
fixchar		Y
string	Y	Returns a trailing blank *only* if the column contains an empty string.
varchar	Y	Y
lvarchar	Y	

This section describes the characteristics of these data types and the differences among them.

The char Data Type

The **char** data type is the C data type that holds character data. When an application reads a value from a CHAR column into a host variable of type **char**, Informix ESQL/C pads this value with trailing blanks up to the size of the host variable. It leaves just one place for the null character that terminates the host array. The behavior is the same if an application reads a value from a VARCHAR (or NVARCHAR) column into a host variable of the **char** data type.

Declare a **char** data type with a length of [*n* + 1] (where *n* is the size of the column with values that you want read) to allow for the null terminator. Use the following syntax to declare a host variable of the **char** data type:

```
EXEC SQL BEGIN DECLARE SECTION;
   char ch_name[n + 1];
EXEC SQL END DECLARE SECTION;
```

The fixchar Data Type

The **fixchar** data type is an Informix ESQL/C data type that holds character data that does *not* append a null terminator. When an application reads a value from a CHAR column into a host variable of type **fixchar**, Informix ESQL/C pads this value with trailing blanks up to the size of the host variable. Informix ESQL/C does not append any null character. The behavior is the same if an application reads a value from a VARCHAR (or NVARCHAR) column into a host variable of the **fixchar** data type.

Important: Do not use the **fixchar** data type with VARCHAR, or NVARCHAR, data. With a **fixchar**, even if the length of the data is shorter than the size of the **fixchar**, the database server stores all n characters of the **fixchar**, including any blanks at the end of the string. Unless the blanks have significance, storing them defeats the space savings that the VARCHAR data type provides.

Declare a **fixchar** host variable as an array with *n* components (where *n* is the size of the column with values that you want read). Use the following syntax to declare a host variable of the **fixchar** data type:

```
EXEC SQL BEGIN DECLARE SECTION;
   fixchar fch_name[n];
EXEC SQL END DECLARE SECTION;
```

Important: You can copy a null-terminated C string into a **fixchar** variable if space is available for the null character. However, this is not good practice. When the database server inserts this value into a column, it also inserts the null terminator. As a result, later searches of the table might fail to find the value.

The string Data Type

The **string** data type is an Informix ESQL/C data type that holds character data that is null terminated and does not contain trailing blanks. However, if a string of blanks (that is, ' ') is stored in a database field and selected into a host variable of the **string** data type, the result is a single blank character.

When an application reads a value from a CHAR column into a host variable of the **string** data type, it strips the value of any trailing blanks and appends a null terminator. The behavior is the same if an application reads a value from a VARCHAR column into a host variable of the **string** data type.

The one exception to this rule is that if the BLANK_STRINGS_NOT_NULL environment variable is set to 1 or any other value, like 0 or 2, the string host variable stores an empty string as a single blank followed by a null terminator. If this environment variable is not set, string host variables store an empty string as a null string.

```
EXEC SQL BEGIN DECLARE SECTION;
   string buffer[16];
EXEC SQL END DECLARE SECTION;
   .
   .
   .
EXEC SQL select lname into :buffer from customer
   where customer_num = 102;
```

Declare the **string** data type with a length of [*n* + 1] (where *n* is the size of the column with values that you want read) to allow for the null terminator. In the preceding code fragment, the **lname** column in the **customer** table is 15 bytes so the **buffer** host variable is declared as 16 bytes. Use the following syntax to declare a host variable of the **string** data type:

```
EXEC SQL BEGIN DECLARE SECTION;
   string str_name[n + 1];
EXEC SQL END DECLARE SECTION;
```

The varchar Data Type

The **varchar** data type is an Informix ESQL/C data type that holds character data of varying lengths. When an application reads a value from a CHAR column into a host variable of type **varchar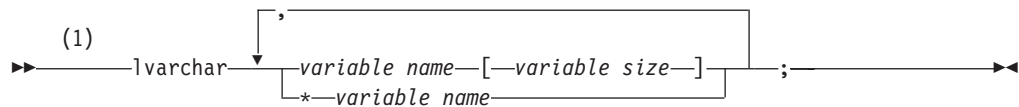**, Informix ESQL/C preserves any trailing blanks and terminates the array with a null character. The behavior is the same if an application reads a value from a VARCHAR column into a host variable of the **varchar** data type.

Declare the **varchar** data type with a length of [*n* +1] (where *n* is the maximum size of the column with values that you want read) to allow for the null terminator. Use the following syntax to declare a host variable of the **varchar** data type:

```
EXEC SQL BEGIN DECLARE SECTION;
   varchar varc_name[n + 1];
EXEC SQL END DECLARE SECTION;
```

Informix includes the **varchar.h** header file with the Informix ESQL/C libraries. This file defines the names and macro functions shown in Table 4-2.

Table 4-2. VARCHAR Size Macros

Name of Macro	Description
MAXVCLEN	The maximum number of characters that you can store in a VARCHAR column. This value is 255.
VCLENGTH(s)	The length to declare the host variable.
VCMIN(s)	The minimum number of characters that you can store in the VARCHAR column. Can range from 1 to 255 bytes but must be smaller than the maximum size of the VARCHAR.
VCMAX(s)	The maximum number of characters that you can store in the VARCHAR column. Can range from 1 to 255 bytes.
VCSIZ(min, max)	The encoded size value, based on *min* and *max*, for the VARCHAR column.

These macros are useful when your program uses dynamic SQL. After a DESCRIBE statement, the macros can manipulate size information that the database server stores in the LENGTH field of the system-descriptor area (or the **sqllen** field of the **sqlda** structure). Your database server stores size information for a VARCHAR column in the **syscolumns** system catalog table.

The **varchar.ec** demonstration program obtains **collength** from the **syscolumns** system catalog table for the **cat_advert** column (of the **stores7** database). It then uses the macros from **varchar.h** to display size information about the column. This sample program is in the **varchar.ec** file in the **demo** directory. Figure 4-1 shows the **main()** function for the **varchar.ec** demonstration program.

```
/*
   * varchar.ec *

   The following program illustrates the use of VARCHAR macros to
   obtain size information.
*/

EXEC SQL include varchar;

char errmsg[512];

main()
{
    mint vc_code;
    mint max, min;
    mint hv_length;

    EXEC SQL BEGIN DECLARE SECTION;
      mint vc_size;
    EXEC SQL END DECLARE SECTION;

    printf("VARCHAR Sample ESQL Program running.\n\n");

    EXEC SQL connect to 'stores7';
    chk_sqlcode("CONNECT");

    printf("VARCHAR field 'cat_advert':\n");
    EXEC SQL select collength into $vc_size from syscolumns
      where colname = "cat_advert";
    chk_sqlcode("SELECT");
    printf("\tEncoded size of VARCHAR (from syscolumns.collength) = %d\n",
      vc_size);

    max = VCMAX(vc_size);
    printf("\tMaximum number of characters = %d\n", max);

    min = VCMIN(vc_size);
    printf("\tMinimum number of characters = %d\n", min);

    hv_length = VCLENGTH(vc_size);
    printf("\tLength to declare host variable = char(%d)\n", hv_length);

    vc_code = VCSIZ(max, min);
    printf("\tEncoded size of VARCHAR (from VCSIZ macro) = %d\n", vc_code);

    printf("\nVARCHAR Sample Program over.\n\n");
}
```

Figure 4-1. The varchar.ec Demonstration Program

When the IFX_PAD_VARCHAR environment variable is set to 1, the client sends
the VARCHAR data type with padded trailing spaces. When this environment is
not set (the default), the client sends the VARCHAR data type value without
trailing spaces. The IFX_PAD_VARCHAR environment variable must be set only at
the client side and is supported only with IBM Informix ESQL/C Version 9.53 and
2.90 or later and IBM Informix Version 9.40 or later.

The lvarchar Data Type

The **lvarchar** data type is an IBM Informix ESQL/C data type that holds character
data of varying lengths. The **lvarchar** data type is implemented as a variable
length user-defined type that is similar to the **varchar** data type except that it can
support strings of greater than 256 bytes and has the following two uses:

• To hold a value for an LVARCHAR column in the database.

   When an application reads a value from an LVARCHAR column into a host
   variable of the **lvarchar** data type, Informix ESQL/C preserves any trailing

blanks and terminates the array with a null character. The behavior is the same if an application reads a value from a VARCHAR column into a host variable of the **lvarchar** data type.

- To represent the string or external format of opaque data types.

  For more information, see "Accessing the External Format of an Opaque Type" on page 10-3.

Important: You cannot retrieve or store smart large objects (CLOB or BLOB data types) from or to an **lvarchar** host variable.

To declare an **lvarchar** host variable for a character column (CHAR, VARCHAR, or LVARCHAR), use the **lvarchar** keyword as the variable data type, as the following syntax shows.

Notes:

1   Informix extension

Element	Purpose	Restrictions
variable name	Name of an **lvarchar** variable of a specified size	
variable size	Number of bytes to allocate for an **lvarchar** variable of specified size	Integer value can be between 1 and 32,768 (32 KB).
*variable name	Name of an **lvarchar** pointer variable for data of unspecified length	Not equivalent to a C char pointer (char *). Points to an internal ESQL/C representation for this type. You must use the **ifx_var()** functions to manipulate data. For more information, see "The lvarchar pointer and var binary library functions" on page 10-22.

Figure 4-2 shows declarations for three **lvarchar** variables that hold values for LVARCHAR columns.

```
EXEC SQL BEGIN DECLARE SECTION;
    lvarchar *a_polygon;
    lvarchar circle1[CIRCLESZ],           circle2[CIRCLESZ];
EXEC SQL END DECLARE SECTION;
```

Figure 4-2. Sample lvarchar Host Variables

Important: To declare a lvarchar host variable for the external format of an opaque data type, use the syntax described in "Declaring lvarchar Host Variables" on page 10-4.

A lvarchar Host Variable of a Fixed Size

If you do not specify the size of a lvarchar host variable, the size is equivalent to a 1-byte C-language **char** data type. If you specify a size, the **lvarchar** host variable is equivalent to a C-language **char** data type of that size. When you specify a fixed-size **lvarchar** host variable, any data beyond the specified size will be truncated when the column is fetched. Use an indicator variable to check for truncation.

Because a lvarchar host variable of a known size is equivalent to a C-language **char** data type, you can use C-language character string operations to manipulate them.

The lvarchar Pointer Host Variable

When the **lvarchar** host variable is a pointer, the size of the data that the pointer references can range up to 2 gigabytes. The **lvarchar** pointer host variable is designed to insert or select user-defined or opaque types that can be represented in a character string format.

You must use the **ifx_var()** functions to manipulate a lvarchar pointer host variable.

For more information about the **ifx_var()** functions, see Appendix B, "ESQL/C function library," on page B-1. For more information about how to use the Informix ESQL/C **lvarchar** host variable with opaque data types, see Chapter 10, "Working with opaque data types," on page 10-1.

Fetching and Inserting Character Data Types

You can transfer character data between CHAR and VARCHAR columns and character (**char**, **string**, **fixchar**, **varchar**, or **lvarchar**) host variables with either of the following operations:

- A fetch operation transfers character data from a CHAR or VARCHAR column to a character host variable.
- An insert or update operation transfers character data from a character host variable to a CHAR, VARCHAR, or LVARCHAR column.

If you use locale-sensitive character data types (NCHAR or NVARCHAR), you can also transfer character data between NCHAR or NVARCHAR columns and character host variables. For more information about how to declare host variables for the NCHAR and NVARCHAR data types, see the *IBM Informix GLS User's Guide*.

Fetching and Inserting CHAR Data

When an application uses a character host variable to fetch or insert a CHAR value, Informix ESQL/C must ensure that the character value fits into the host variable or database column.

Fetching CHAR Data

An application can fetch data from a database column of type CHAR or VARCHAR into a character (**char**, **string**, **fixchar**, **varchar**, or **lvarchar**) host variable. If the column data does not fit into the character host variable, Informix ESQL/C truncates the data. To notify the user of the truncation, Informix ESQL/C performs the following actions:

- It sets the **sqlca.sqlwarn.sqlwarn1** warning flag to W and the SQLSTATE variable to 01004.
- It sets any indicator variable that is associated with the character host variable to the size of the character data in the column.

For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

Inserting CHAR Data

An application can insert data from a character host variable (**char**, **string**, **fixchar**, **varchar**, or **lvarchar**) into a database column of type CHAR. If the value is shorter than the size of the database column then the database server pads the value with blanks up to the size of the column.

If the value is longer than the size of the column the database server truncates the value if the database is non-ANSI. No warning is generated when this truncation occurs. If the database is ANSI and the value is longer than the column size then the insert fails and this error is returned:

```
-1279: Value exceeds string column length.
```

Although **char**, **varchar**, **lvarchar**, and **string** host variables contain null terminators, Informix ESQL/C never inserts these characters into a database column. (Host variables of type **fixchar** should never contain null characters.)

If you use the locale-sensitive character data type, NCHAR, you can insert a value from a character host variable into an NCHAR column. Insertion into NCHAR columns follows the same behavior as insertion into CHAR columns. For more information about how to declare host variables for the NCHAR data type, see the *IBM Informix GLS User's Guide*.

Do not use the **fixchar** data type for host variables that insert character data into ANSI-compliant databases. For more information, see "Fetching and Inserting with an ANSI-Compliant Database" on page 4-12.

Fetching and Inserting VARCHAR Data

When an application uses a character host variable to fetch or insert a VARCHAR value, Informix ESQL/C must ensure that the character value fits into the host variable or database column. When Informix ESQL/C calculates the length of a source item, it does not count trailing spaces. The following sections describe how Informix ESQL/C performs the conversion of VARCHAR data to and from **char**, **fixchar**, and **string** character data types.

These conversions also apply to NVARCHAR data. For more information about the NVARCHAR data type, see the *IBM Informix GLS User's Guide*.

Fetching VARCHAR Data

Table 4-3 shows the conversion of VARCHAR data when an application fetches it into host variables of **char**, **fixchar**, **lvarchar**, and **string** character data types.

Table 4-3. Converting the VARCHAR Data Type to ESQL/C Character Data Types

Source Type	Destination Type	Result
VARCHAR	**char**	If the source is longer, truncate and null terminate the value, and set any indicator variable. If the destination is longer, pad the value with trailing spaces and null terminate it.
VARCHAR	**fixchar**	If the source is longer, truncate the value and set any indicator variable. If the destination is longer, pad the value with trailing spaces.
VARCHAR	**string**	If the source is longer, truncate and null terminate the value, and set any indicator variable. If the destination is longer, null terminate the value.
VARCHAR	**lvarchar**	If the source is longer, truncate and set any indicator variable. If the destination is longer, null terminate it.

Table 4-4 shows examples of conversions from VARCHAR column data to character host variables that Informix ESQL/C might perform during a fetch. In this figure, a plus (+) symbol represents a space character and the value in the **Length** column includes any null terminators.

Table 4-4. Examples of VARCHAR Conversion During a Fetch

Source Type	Contents	Length	Destination 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
VARCHAR(10)	Fairfield+	10	lvarchar(11)	Fairfield+	0
VARCHAR(9)	Fairfield	9	lvarchar(5)	Fair\0	9

Inserting VARCHAR Data

When an application inserts a value from a **char**, **varchar**, **lvarchar**, or **string** host variable into a VARCHAR column, Informix ESQL/C also inserts any trailing blanks. Informix ESQL/C does not, however, add trailing blanks.

If the value is longer than the maximum size of the column, the database server truncates the value if the database is non-ANSI. No warning is generated when this truncation occurs. If the database is ANSI and the value is longer than the maximum column size then the insert fails and this error is returned:

`-1279: Value exceeds string column length.`

Although **char**, **varchar**, **lvarchar**, and **string** host variables contain null terminators, Informix ESQL/C never inserts these characters into a database column. (Host variables of type **fixchar** should never contain null characters.) If an application inserts a **char**, **varchar**, **lvarchar**, or **string** value into a VARCHAR column, the database server keeps track of the end of the value internally.

Table 4-5 shows the conversion of VARCHAR data when an application inserts it from host variables of **char**, **fixchar**, **lvarchar**, and **string** character data types.

Table 4-5. Converting ESQL/C Character Data Types to the VARCHAR Data Type

Source Type	Destination Type	Result
char	VARCHAR	If the source is longer than the max VARCHAR, truncate the value and set the indicator variable. If the max VARCHAR is longer than the source, the length of the destination equals the length of the source (not including the null terminator of the source).
fixchar	VARCHAR	If the source is longer than the max VARCHAR, truncate the value and set the indicator variable. If the max VARCHAR is longer than the source, the length of the destination equals the length of the source.
string	VARCHAR	If the source is longer than the max VARCHAR, truncate the value and set the indicator variable. If the max VARCHAR is longer than the source, the length of the destination equals the length of the source (not including the null terminator of the source).
lvarchar	VARCHAR	If the source is longer than the max VARCHAR, truncate the value and set the indicator variable. If the max VARCHAR is longer than the source, the length of the destination equals the length of the source.

If you use the locale-sensitive character data type, NVARCHAR, you can insert a value from a character host variable into an NVARCHAR column. Insertion into NVARCHAR columns follows the same behavior as insertion into VARCHAR columns. For more information about how to declare host variables for the NVARCHAR data type, see the *IBM Informix GLS User's Guide*.

Table 4-6 shows examples of conversions from character host variables to VARCHAR column data that Informix ESQL/C might perform during an insert. In this figure, a plus (+) symbol represents a space character.

Table 4-6. Examples of VARCHAR Conversion During an Insert

Source Type	Contents	Length	Destination Type	Contents	Length
char(10)	Fairfield\0	10	VARCHAR(4)	Fair	4
char(10)	Fairfield\0	10	VARCHAR(11)	Fairfield	9
char(12)	Fairfield++\0	12	VARCHAR(9)	Fairfield	9
char(13)	Fairfield+++\0	13	VARCHAR(6)	Fairfi	6
char(11)	Fairfield+\0	11	VARCHAR(11)	Fairfield+	10
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+	10
string(9)	Fairfield\0	9	VARCHAR(4)	Fair	4
string(9)	Fairfield\0	9	VARCHAR(11)	Fairfield	9

Fetching and Inserting lvarchar Data

When an application uses a lvarchar host variable to fetch or insert a data value, IBM Informix ESQL/C must ensure that the value fits into the host variable or database column.

Fetching lvarchar Data

An application can fetch data from a database column of type LVARCHAR into a character (**char**, **string**, **fixchar**, **varchar**, or **lvarchar**) host variable. If the column data does not fit into the host variable, Informix ESQL/C truncates the data. To notify the user of the truncation, Informix ESQL/C performs the following actions:

- It sets the **sqlca.sqlwarn.sqlwarn1** warning flag to W and the SQLSTATE variable to 01004.
- It sets any indicator variable that is associated with the character host variable to the size of the character data in the column.

For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

Inserting LVARCHAR Data

An application can insert data from a character host variable (**char**, **string**, **fixchar, varchar**, or **lvarchar**) into a database column of type LVARCHAR.

If the value is longer than the maximum size of the column the database server truncates the value if the database is non-ANSI. No warning is generated when this truncation occurs. If the database is ANSI and the value is longer than the maximum column size then the insert fails and this error is returned:

-1279: Value exceeds string column length.

If the host variable you use for the insert is a **char** or **varchar,** the database server casts the type to **lvarchar**.

When you write data to an LVARCHAR column, the database server imposes a limit of 32 KB on the column. If the host variable is a lvarchar data type and the data exceeds 32 KB, the database server returns an error. If the column has an

input support function, it must use any data beyond 32 KB, if necessary, to prevent the database server from returning the error.

Fetching and Inserting with an ANSI-Compliant Database

For an ANSI-compliant database, when you use a character host variable in an INSERT statement or in the WHERE clause of an SQL statement (SELECT, UPDATE, or DELETE), the character value in the host variable must be null terminated. Therefore, use the following data types for character host variables:

- **char**, **string**, or **varchar**
- **lvarchar**

For example, the following insertion is valid because the first and **last** host variables are of type **char**, which is null terminated:

```
EXEC SQL BEGIN DECLARE SECTION;
   char first[16], last[16];
EXEC SQL END DECLARE SECTION;
   .
   .
   .
stcopy("Dexter", first);
stcopy("Haven", last);
EXEC SQL insert into customer (fname, lname)
   values (:first, :last);
```

The **stcopy()** function copies the null terminator into the host variable and the **char** data type retains the null terminator.

Do not use the **fixchar** data type for host variables because it does not include a null terminator on the string. For an ANSI-compliant database, the database server generates an error under either of the following conditions:

- If you try to insert a string that is not null terminated.
- If you use a string that is not null terminated in a WHERE clause.

Character and string library functions

The IBM Informix ESQL/C library contains the following character-manipulation functions. You can use these functions in your C programs to manipulate single characters and strings of bytes and characters, including variable-length expressions of the following data types:

- **varchar**
- fixed-size **lvarchar**

The internal structure referenced by the lvarchar pointer data type is different than the character representation of a fixed-size **lvarchar** variable. You must use the **ifx_var()** functions to manipulate **lvarchar** pointer variables. For more information about the **ifx_var()** functions, see "The lvarchar pointer and var binary library functions" on page 10-22.

The functions whose names begin with **by** act on and return fixed-length strings of bytes. The functions whose names begin with **rst** and **st** (except **stchar**) operate on and return null-terminated strings. The **rdownshift()** and **rupshift** functions also operate on null-terminated strings but do not return values. When you compile your Informix ESQL/C program with the **esql** preprocessor, it calls on the linker to link these functions to your program. The following list provides brief descriptions of the character and string library functions and refers you to the pages where detailed information for each function is given.

Function Name	Description	See
bycmpr()	Compares two groups of contiguous bytes	B-10
bycopy()	Copies bytes from one area to another	B-12
byfill()	Fills an area you specify with a character	B-13
byleng()	Counts the number of bytes in a string	B-14
ldchar()	Copies a fixed-length string to a null-terminated string	B-170
rdownshift()	Converts all letters to lowercase	B-176
rstod()	Converts a **string** to a **double** value	B-200
rstoi()	Converts a **string** to a **short integer** value	B-201
rstol()	Converts a **string** to a 4-byte integer value	B-202
rupshift()	Converts all letters to uppercase	B-209
stcat()	Concatenates one string to another	B-232
stchar()	Copies a null-terminated string to a fixed-length string	B-233
stcmpr()	Compares two strings	B-234
stcopy()	Copies one string to another string	B-235
stleng()	Counts the number of bytes in a string	B-235

Chapter 5. Working with numeric data types

IBM Informix database servers support the following numeric data types:

- Integer data types: SMALLINT, INTEGER, INT8, SERIAL, SERIAL8
- The Boolean data type
- Fixed-point data types: DECIMAL and MONEY
- Floating-point data types: SMALLFLOAT and FLOAT

These topics contain information about working with numeric data types:

- Informix ESQL/C data types to use as host variables for SQL numeric data types
- Characteristics of Informix ESQL/C numeric data types
- Formatting masks, which you can use to format numeric data types
- Informix ESQL/C library functions that you can use to manipulate numeric data types

For information about all the data types available for use in an Informix ESQL/C program, see Chapter 3, "Informix ESQL/C data types," on page 3-1.

The integer data types

The database server supports the following data types for integer values.

SQL Integer Data Type	Number of Bytes	Range of Values
SMALLINT	2	-32767 to 32767
INTEGER, INT, SERIAL	4	-2,147,483,647 to 2,147,483,647
INT8, BIGINT, SERIAL8, BIGSERIAL	8	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807

The C language supports the **short int** and **long int** data types for integers. The size of these C integer data types depends on the hardware and operating system of the computer you use. For example, a **long** data type could be 4 bytes long on a 32-bit platform but 8 bytes long on a 64-bit platform.

Therefore, when you declare an integer host variable, you must ensure that this host variable is large enough for all possible values of the SQL integer data type

with which the variable is associated. For more information about how to implement integer data types on your system, check with your system administrator or your C documentation.

The Integer Host Variable Types

The following data types are provided for specifying integer host variables of specific lengths.

Data Type	Length
int1	One-byte integer
int2	Two-byte integer
int4	Four byte integer
mint	Native integer data type for the machine
mlong	Native **long** integer data type for the machine, the size of which is equal to that of the pointer for the machine. The **mlong** data type is mapped to the **long** data type on Windows 32-bit and UNIX and LINUX 32-bit and 64-bit platforms. It is mapped to the **__int64** data type on Windows 64-bit platforms.
MSHORT	Native **short** integer data type for the machine
MCHAR	Native **char** data type for the machine

> **Warning:** The preceding integer data types are reserved. Your programs must not use typedef or $typedef statements to define these data types.

The integer host variable data types are defined in the **ifxtypes.h** file, which is automatically included in your program when you compile it with the **esql** script.

> **Important:** Many of the Informix ESQL/C library functions have been changed to declare the Informix integer data types rather than the machine specific types such as **int**, **short**, and **long**. It is recommended that you use the Informix integer types when you call Informix ESQL/C library functions.

The INT8 and SERIAL8 SQL Data Types

Informix ESQL/C supports the SQL INT8 and SERIAL8 data types with the **int8** data type. The **int8** data type is a machine-independent method that represents numbers in the range $-(2^{63}-1)$ to $2^{63}-1$.

For a complete description of the INT8 and SERIAL8 SQL data types, see the *IBM Informix Guide to SQL: Reference*. This section describes how to manipulate the Informix ESQL/C data type, **int8**.

The int8 Data Type

Use the Informix ESQL/C **int8** data type to declare host variables for database values of type INT8 and SERIAL8. Table 5-1 shows the fields of the structure **ifx_int8_t**, which represents an INT8 or SERIAL8 value.

Table 5-1. Fields of the ifx_int8_t Structure

Field Name	Field Type	Purpose
data	*unsigned 4-byte integer*[INT8SIZE]	An array of integer values that make up the 8-byte integer value. When the INT8SIZE constant is defined as 2, this array contains two unsigned 4-byte integers. *The actual data type of an unsigned 4-byte integer can be machine specific.*
sign	short integer	A short integer to hold the sign (null, negative, or positive) of the 8-byte integer. The actual data type of a 2-byte integer can be machine specific.

The **int8.h** header file contains the **ifx_int8** structure and a **typedef** called **ifx_int8_t**. Include this file in all C source files that use any **int8** host variables as shown in the following example:

```
EXEC SQL include int8;
```

You can declare an **int8** host variable in either of the following ways:

```
EXEC SQL BEGIN DECLARE SECTION;
   int8 int8_var1;
   ifx_int8_t int8_var2;
EXEC SQL BEGIN DECLARE SECTION;
```

The int8 Library Functions

You must perform all operations on **int8** type numbers through the Informix ESQL/C library functions for the **int8** data type. Any other operations, modifications, or analyses can produce unpredictable results. The Informix ESQL/C library provides functions that allow you to manipulate **int8** numbers and convert **int8** type numbers to and from other data types. The following table describes these functions.

Function Name	Description	See
Manipulation Functions		
ifx_getserial8()	Returns an inserted SERIAL8 value	B-73
ifx_int8add()	Adds two int8 numbers	B-75
ifx_int8cmp()	Compares two int8 numbers	B-76
ifx_int8copy()	Copies an int8 number	B-78
ifx_int8div()	Divides two int8 numbers	B-87
ifx_int8mul()	Multiplies two int8 numbers	B-89
ifx_int8sub()	Subtracts two int8 numbers	B-90
Type Conversion Functions		
ifx_int8cvasc()	Converts a C char type value to an **int8** type value	B-80
ifx_int8cvdbl()	Converts a C double type value to an **int8** type value	B-80
ifx_int8cvdec()	Converts a C decimal type value to a **int8** type value	B-81
ifx_int8cvflt()	Converts a C **float** type value to an **int8** type value	B-83
ifx_int8cvint()	Converts a C int type value to an **int8** type value	B-84

Function Name	Description	See
ifx_int8cvlong()	Converts a C 4-byte integer type value to an int8 type value	B-86
ifx_int8toasc()	Converts **an int8** type value to a text string	B-92
ifx_int8todbl()	Converts an int8 type value to a C **double** type value	B-94
ifx_int8todec()	Converts aint8 type value to a **decimal** type value	B-96
ifx_int8toflt()	Converts an **int8** type value to a C **float** type value	B-99
ifx_int8toint()	Converts an int8 type value to a C **int** type value	B-101
ifx_int8tolong()	Converts an int8 type value to a C 4-byte integer type value	B-103

For information about the syntax and behavior of these library functions, see Appendix B, "ESQL/C function library," on page B-1.

The BOOLEAN data type

Informix ESQL/C uses the **boolean** data type to support the SQL BOOLEAN data type. For a complete description of the SQL BOOLEAN data type, see the *IBM Informix Guide to SQL: Reference*. This section describes how to manipulate the Informix ESQL/C **boolean** data type.

You can declare a **boolean** host variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   boolean flag;
EXEC SQL BEGIN DECLARE SECTION;
```

In an Informix ESQL/C program, the following values are the only valid values that you can assign to **boolean** host variables:

TRUE	'\1'
FALSE	'\0'
NULL	Use the **rsetnull()** function with the CBOOLTYPE as the first argument

If you want to assign the character representations of 'T' or 'F' to a BOOLEAN column, you must declare a **fixchar** host variable and initialize it to the desired character value. Use this host variable in an SQL statement such as the INSERT or UPDATE statement. The database server converts the **fixchar** value to the appropriate BOOLEAN value.

The following code fragment inserts two values into a BOOLEAN column called **bool_col** in the **table2** table:

```
EXEC SQL BEGIN DECLARE SECTION;
   boolean flag;
   fixchar my_boolflag;
   int id;
EXEC SQL END DECLARE SECTION;

id = 1;
```

```
flag = '\0';  /* valid boolean assignment to FALSE */
EXEC SQL insert into table2 values (:id, :flag); /* inserts FALSE */

id = 2;
rsetnull(CBOOLTYPE, (char *) &flag); /* valid BOOLEAN assignment
                                      * to NULL */
EXEC SQL insert into table2 values (:id, :flag); /* inserts NULL */

id = 3;
my_boolflag = 'T' /* valid character assignment to TRUE */
EXEC SQL insert into table2 values (:id, :my_boolflag); /* inserts TRUE
                                                        */
```

The decimal data type

Informix ESQL/C supports the SQL DECIMAL and MONEY data types with the **decimal** data type. The **decimal** data type is a machine-independent method that represents numbers of up to 32 significant digits, with valid values in the range 10^{-129} to 10^{+125}.

The DECIMAL data type can take the following two forms:

- DECIMAL(*p*) floating point

  When you define a column with the DECIMAL(*p*) data type, it has a total of *p* (< = 32) significant digits. DECIMAL(*p*) has an absolute value range between 10^{-130} and 10^{124}.

- DECIMAL(*p,s*) fixed point

  When you define a column with the DECIMAL(*p,s*) data type, it has a total of *p* (< = 32) significant digits (the precision) and *s* (< = *p*) digits to the right of the decimal point (the scale).

For a complete description of the DECIMAL data type, see the *IBM Informix Guide to SQL: Reference*.

The decimal structure

Use the **decimal** data type to declare host variables for database values of type DECIMAL. A structure of type **decimal** represents a value in a **decimal** host variable, as follows:

```
#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.h** header file contains the **decimal** structure and the **typedef dec_t**. Include this file in all C source files that use any **decimal** host variables with the following **include** directive:

```
EXEC SQL include decimal;
```

The **decimal** structure stores the number in *pairs* of digits. Each pair is a number in the range 00 to 99. (Therefore, you can think of a pair as a base-100 digit.) Table 5-2 shows the four parts of the **decimal** structure.

Table 5-2. Fields in the decimal Structure

Field	Description
dec_exp	The *exponent* of the normalized **decimal** type number. The normalized form of this number has the decimal point at the left of the left-most digit. This exponent represents the number of digit pairs to count from the *left* to position the decimal point (or as a power of 100 for the number of base-100 numbers).
dec_pos	The *sign* of the **decimal** type number. The **dec_pos** field can assume any one of the following three values: 1: when the number is zero or greater 0: when the number is less than zero −1: when the value is null
dec_ndgts	The *number of digit pairs* (number of base-100 significant digits) in the **decimal** type number. This value is also the number of entries in the **dec_dgts** array.
dec_dgts[]	A character array that holds the significant digits of the normalized decimal type number, assuming **dec_dgts[0] ! = 0**. Each byte in the array contains the next significant base-100 digit in the **decimal** type number, proceeding from **dec_dgts[0]** to **dec_dgts[dec_ndgts]**.

Table 5-3 shows some sample **decimal** values.

Table 5-3. Sample decimal Values

	decimal Structure Field Values			
Value	dec_exp	dec_pos	dec_ndgts	dec_dgts[]
-12345.6789	3	0	5	dec_dgts[0] = 01 dec_dgts[1] = 23 dec_dgts[2] = 45 dec_dgts[3] = 67 dec_dgts[4] = 89
1234.567	2	1	4	dec_dgts[0] = 12 dec_dgts[1] = 34 dec_dgts[2] = 56 dec_dgts[3] = 70
-123.456	2	0	4	dec_dgts[0] = 01 dec_dgts[1] = 23 dec_dgts[2] = 45 dec_dgts[3] = 60
480	2	1	2	dec_dgts[0] = 04 dec_dgts[1] = 80
.152	0	1	2	dec_dgts[0] = 15 dec_dgts[1] = 20
-6	1	0	1	dec_dgts[0] = 06

You can use the **deccvasc** demonstration program to experiment with how Informix ESQL/C stores **decimal** numbers.

The decimal Library Functions

You must perform all operations on **decimal** type numbers through the following Informix ESQL/C library functions for the **decimal** data type. Any other operations, modifications, or analyses can produce unpredictable results. For more information about the syntax and behavior of these functions, see the referenced pages.

Function Name	Description	See
Manipulation Functions		
decadd()	Adds two decimal numbers	B-15
deccmp()	Compares two decimal numbers	B-17
deccopy()	Copies a decimal number	B-18
decdiv()	Divides two decimal numbers	B-27
decmul()	Multiplies two decimal numbers	B-32
decround()	Rounds a decimal number	B-34
decsub()	Subtracts two decimal numbers	B-35
dectrunc()	Truncates a decimal number	B-43
Type Conversion Functions		
deccvasc()	Converts a C char type value to a **decimal** type value	B-19
deccvdbl()	Converts a C double type value to a **decimal** type value	B-21
deccvint()	Converts a C int type value to a **decimal** type value	B-24
deccvlong()	Converts a C 4-byte integer type value to a decimal type value	B-25
dececvt()	Converts a decimal value to an ASCII string	B-28
decfcvt()	Converts a decimal value to an ASCII string	B-28
dectoasc()	Converts **a decimal** type value to an ASCII string	B-36
dectodbl()	Converts a decimal type value to a C **double** type value	B-39
dectoint()	Converts a decimal type value to a C **int** type value	B-40
dectolong()	Converts a decimal type value to a C 4-byte integer type value	B-42

For information about the function **rfmtdec()**, which allows you to format a decimal number, see "Numeric-Formatting Functions" on page 5-13. For additional information about **decimal** values, see "Using Operations That Involve a Decimal Value" on page 3-11

The Floating-Point Data Types

The database server supports the following data types for floating-point values.

SQL Floating-Point Data Type	ESQL/C or C Language Type	Range of Values
SMALLFLOAT, REAL	**float**	Single-precision values with up to 9 significant digits
FLOAT, DOUBLE PRECISION	**double**	Double-precision values with up to 17 significant digits
DECIMAL(*p*)	**decimal**	Absolute value range between 10^{-130} and 10^{124}

Declaring float Host Variables

When you use the C **float** data type (for SMALLFLOAT values), be aware that most C compilers pass **float** to a function as the **double** data type. If you declare the function argument as a **float**, you might receive an incorrect result. For example, in the following excerpt, **:hostvar** might produce an incorrect value in **tab1**, depending on how your C compiler handles the **float** data type when your program passes it as an argument.

```
main()
{
   double dbl_val;

   EXEC SQL connect to 'mydb';
   ins_tab(dbl_val);
   .
   .
   .
}

ins_tab(hostvar)
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER double hostvar;
EXEC SQL END DECLARE SECTION;
{
   EXEC SQL insert into tab1 values (:hostvar, ...);
}
```

For more information about the SQL floating point data types, see the *IBM Informix Guide to SQL: Reference*

Implicit Data Conversion

When an Informix ESQL/C program fetches a floating-point column value into a character host variable (**char**, **fixchar**, **varchar**, or **string**), it includes only the number of decimal digits that can fit into the character buffer. If the host variable is too small for the full precision of the floating-point number, Informix ESQL/C rounds the number to the precision that the host variable can hold.

In the following code fragment, an Informix ESQL/C program retrieves the value 1234.8763512 from a FLOAT column that is called **principal** into the **prncpl_strng** character host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   char prncpl_strng[15]; /* character host variable */
EXEC SQL END DECLARE SECTION;
   .
   .
```

```
EXEC SQL select principal into :prncpl_strng from loan
   where customer_id = 1098;
printf("Value of principal=%s\n", prncpl_strng);
```

Because the **prncpl_strng** host variable is a buffer of 15 characters, Informix ESQL/C is able to put all decimal digits into the host variable and this code fragment produces the following output:

```
Value of principal=1234.876351200
```

However, if the preceding code fragment declares the **prncpl_strng** host variable as a buffer of 10 characters, Informix ESQL/C rounds the FLOAT value to fit into **prncpl_strng** and the code fragment produces the following output:

```
Value of principal=1234.8764
```

Informix ESQL/C assumes a precision of 17 decimal digits for FLOAT or SMALLFLOAT values. For DECIMAL(*n*,*m*), Informix ESQL/C assumes *m* decimal digits.

Formatting Numeric Strings

A *numeric-formatting mask* specifies a format to apply to some numeric value. This mask is a combination of the following formatting characters:

*   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 changes leading zeros to blanks. Use this character to specify the maximum leftward extent of a field.

<   This character left-justifies the numbers in the display field. It changes leading zeros to a null string.

,   This character indicates the symbol that separates groups of three digits (counting leftward from the units position) in the whole-number part of the value. By default, this symbol is a comma. You can set the symbol with the **DBMONEY** environment variable. In a formatted number, this symbol appears only if the whole-number part of the value has four or more digits.

.   This character indicates the symbol that separates the whole-number part of a money value from the fractional part. By default, this symbol is a period. You can set the symbol with the **DBMONEY** environment variable. You can have only one period in a format string.

-       This character is a literal. It appears as a minus sign when *expr1* is less than zero. When you group several minus signs in a row, a single minus sign floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.

+   This character is a literal. It appears 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 plus signs in a row, a single plus or minus sign floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.

(		This character is a literal. It appears as a left parenthesis to the left of a negative number. It is one of the pair of accounting parentheses that replace a minus sign for a negative number. When you group several in a row, a single left parenthesis floats to the rightmost position that it can occupy; it does not interfere with the number and its currency symbol.
)		This is one of the pair of accounting parentheses that replace a minus sign for a negative value.
$		This character displays the currency symbol that appears at the front of the numeric value. By default, the currency symbol is the dollar sign ($). You can set the currency symbol with the **DBMONEY** environment variable. When you group several dollar signs in a row, a single currency symbol floats to the rightmost position that it can occupy; it does not interfere with the number.

Any other characters in the formatting mask are reproduced literally in the result.

When you use the following characters within a formatting mask, the characters *float*; that is, multiple occurrences of the character at the left of the pattern in the mask appear as a single character as far to the right as possible in the formatted number (without removing significant digits):

```
-
+
(
)
$
```

For example, if you apply the mask $$$,$$$.## to the number 1234.56, the result is $1,234.56.

When you use **The rfmtdec() function, The rfmtdouble() function,** or **The rfmtlong() function** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, the numeric-formatting functions use the currency symbols that the client locale defines. The default local, U.S. English, defines currency symbols as if you set **DBMONEY** to "$,.". For a discussion of **DBMONEY**, see the *IBM Informix Guide to SQL: Reference*. For more information about locales, see the *IBM Informix GLS User's Guide*.

Table 5-4 on page 5-10 shows sample format strings for numeric expressions. The character b represents a blank or space.

Table 5-4. Sample Format Patterns and Their Results

Formatting Mask	Numeric Value	Formatted Result
"#####"	0	bbbbb 00000 bbbb$ *****
"&&&&&"	0	(null string)
"$$$$$"	0	
"*****"	0	
"<<<<<"	0	

Table 5-4. Sample Format Patterns and Their Results (continued)

Formatting Mask	Numeric Value	Formatted Result
"##,###"	12345	12,345 b1,234 bbb123 bbbb12
"##,###"	1234	bbbbb1 bbbbb1 bbbbbb
"##,###"	123	
"##,###"	12	
"##,###"	1	
"##,###"	-1	
"##,###"	0	
"&&,&&&"	12345	12,345
"&&,&&&"	1234	01,234
"&&,&&&"	123	000123
"&&,&&&"	12	000012
"&&,&&&"	1	000001
"&&,&&&"	-1	000001
"&&,&&&"	0	000000
"$$,$$$"	12345	***** (overflow)
"$$,$$$"	1234	$1,234 bb$123 bbb$12 bbbb$1
"$$,$$$"	123	bbbb$1 bbbbb$ DM1,234
"$$,$$$"	12	
"$$,$$$"	1	
"$$,$$$"	-1	
"$$,$$$"	0	
"$$,$$$"	1234	
(**DBMONEY** set to DM)		
"**,***"	12345	12,345
"**,***"	1234	*1,234
"**,***"	123	***123
"**,***"	12	****12
"**,***"	1	*****1
"**,***"	0	******
"##,###.##"	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	bbbbb1.00
"&&,&&&.&&"	.67	000000.67
"&&,&&&.&&"	1234.56	01,234.56
"&&,&&&.&&"	123.45	000123.45
"&&,&&&.&&"	0.01	000000.01
"$$,$$$.$$"	12345.67	********* (overflow)
"$$,$$$.$$"	1234.56	$1,234.56 bbbbb$.00
"$$,$$$.##"	0.00	$1,234.00 bbbbb$.00
"$$,$$$.##"	1234.00	$1,234.00
"$$,$$$.&&"	0.00	
"$$,$$$.&&"	1234.00	

Table 5-4. Sample Format Patterns and Their Results (continued)

Formatting Mask	Numeric Value	Formatted Result
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"--#,###.##"	-12.34	b-bbb12.34
"---,###.##"	-12.34	bb-bb12.34
"---,-##.##"	-12.34	bbbb-12.34
"---,--#.##"	-12.34	bbbb-12.34
"--#,###.##"	-1.00	b-bbbb1.00
"---,--#.##"	-1.00	bbbbb-1.00
"-##,###.##"	12345.67	b12,345.67
"-##,###.##"	1234.56	bb1,234.56
"-##,###.##"	123.45	bbbb123.45
"-##,###.##"	12.34	bbbbb12.34
"--#,###.##"	12.34	bbbbb12.34
"---,###.##"	12.34	bbbbb12.34
"---,-##.##"	12.34	bbbbb12.34
"---,---.##"	1.00	bbbbbb1.00
"---,---.--"	-.01	bbbbbb-.01
"---,---.&&"	-.01	bbbbbb-.01
"-$$$,$$$.&&"	-12345.67	-$12,345.67
"-$$$,$$$.&&"	-1234.56	-b$1,234.56
"-$$$,$$$.&&"	-123.45	-bbb$123.45
"--$$,$$$.&&"	-12345.67	-$12,345.67
"--$$,$$$.&&"	-1234.56	b-$1,234.56
"--$$,$$$.&&"	-123.45	b-bb$123.45
"--$$,$$$.&&"	-12.34	b-bbb$12.34
"--$$,$$$.&&"	-1.23	b-bbbb$1.23
"----,--$.&&"	-12345.67	-$12,345.67
"----,--$.&&"	-1234.56	b-$1,234.56
"----,--$.&&"	-123.45	bbb-$123.45
"----,--$.&&"	-12.34	bbbb-$12.34
"----,--$.&&"	-1.23	bbbbb-$1.23
"----,--$.&&"	-.12	bbbbb-$.12
"$***,***.&&"	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	b($1,234.56)
"(($$,$$$.&&)"	-123.45	b(bb$123.45)
"(($$,$$$.&&)"	-12.34	b(bbb$12.34)
"(($$,$$$.&&)"	-1.23	b(bbbb$1.23)
"(((((,(($.&&)"	-12345.67	($12,345.67)
"(((((,(($.&&)"	-1234.56	b($1,234.56)
"(((((,(($.&&)"	-123.45	bbb($123.45)
"(((((,(($.&&)"	-12.34	bbbb($12.34)
"(((((,(($.&&)"	-1.23	bbbbb($1.23)
"(((((,(($.&&)"	-.12	bbbbbb($.12)

Table 5-4. Sample Format Patterns and Their Results (continued)

Formatting Mask	Numeric Value	Formatted Result
"($$$,$$$.&&)"	12345.67	b$12,345.67
"($$$,$$$.&&)"	1234.56	bb$1,234.56
"($$$,$$$.&&)"	123.45	bbbb$123.45
"(($$,$$$.&&)"	12345.67	b$12,345.67
"(($$,$$$.&&)"	1234.56	bb$1,234.56
"(($$,$$$.&&)"	123.45	bbbb$123.45
"(($$,$$$.&&)"	12.34	bbbbb$12.34
"(($$,$$$.&&)"	1.23	bbbbbb$1.23
"(((((,((($.&&)"	12345.67	b$12,345.67
"(((((,((($.&&)"	1234.56	bb$1,234.56
"(((((,((($.&&)"	123.45	bbbb$123.45
"(((((,((($.&&)"	12.34	bbbbb$12.34
"(((((,((($.&&)"	1.23	bbbbbb$1.23
"(((((,((($.&&)"	.12	bbbbbbb$.12
"<<<,<<<"	12345	12,345
"<<<,<<<"	1234	1,234
"<<<,<<<"	123	123
"<<<,<<<"	12	12

Numeric-Formatting Functions

Special functions are provided that allow you to format numeric expressions for display. These formatting functions apply a given formatting mask to a numeric value to allow you to line up decimal points, right- or left-justify the number, enclose a negative number in parentheses, and so on. The Informix ESQL/C library includes the following functions that support formatting masks for numeric values.

Function Name	Description	See
rfmtdec()	Converts a decimal value to a string	B-181
rfmtdouble()	Converts a double value to a string	B-183
rfmtlong()	Converts a 4-byte integer value to a string	B-185

For information about the syntax and behavior of these library functions, see Appendix B, "ESQL/C function library," on page B-1.

Chapter 6. Working with time data types

These topics explain how to use **date**, **datetime**, and **interval** data types in an IBM Informix ESQL/C program. It contains the following information:

- An overview of the Informix ESQL/C **date** data type
- The syntax of the Informix ESQL/C library functions that you can use to manipulate the **date** data type
- An overview of the Informix ESQL/C **datetime** and **interval** data types and how to use them
- The syntax of Informix ESQL/C library functions that you can use to manipulate the **datetime** and **interval** data types

For information about all the data types that are available for use in an Informix ESQL/C program, see Chapter 3, "Informix ESQL/C data types," on page 3-1. For information about SQL data types, see the *IBM Informix Guide to SQL: Reference*.

The SQL DATE data type

Informix ESQL/C supports the SQL DATE data type with the Informix ESQL/C **date** data type for host variables. The **date** data type stores internal DATE values. It is implemented as a 4-byte integer 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 SQL DATE data type, see the *IBM Informix Guide to SQL: Reference*.

Formatting date strings

A *date-formatting mask* specifies a format to apply to some date value. This mask is a combination of the following formats.

dd	Day of the month as a two-digit number (01 through 31)
ddd	Day of the week as a three-letter abbreviation (Sun through Sat)
mm	Month as a two-digit number (01 through 12)
mmm	Month as a three-letter abbreviation (Jan through Dec)
yy	Year as a two-digit number (00 through 99)

| yyyy | Year as a four-digit number (0001 through 9999) |
| ww | Day of the week as a two-digit number (00 for Sunday, 01 for Monday, 02 for Tuesday ... 06 for Saturday) |

Any other characters in the formatting mask are reproduced literally in the result.

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in a numeric-formatting mask. For more information, see the *IBM Informix GLS User's Guide*.

When you use **The rfmtdate() function** or **The rdefmtdate() function** to format DATE values, the function uses the date end-user formats that the **GL DATE** or **DB DATE** environment variable specifies. If neither of these environment variables is set, these date-formatting functions use the date end-user formats for the locale. The default locale, U.S. English, uses the format *mm/dd/yyyy*. For a discussion of **GL DATE** and **DB DATE**, see the *IBM Informix GLS User's Guide*.

DATE Library Functions

The following date-manipulation functions are in the Informix ESQL/C library. They convert dates between a string format and the internal DATE format.

Function Name	Description	See
rdatestr()	Converts an internal DATE to a character string format	B-171
rdayofweek()	Returns the day of the week of a date in internal format	B-172
rdefmtdate()	Converts a specified string format to an internal DATE	B-174
rfmtdate()	Converts an internal DATE to a specified string format	B-178
rjulmdy()	Returns month, day, and year from a specified DATE	B-193
rleapyear()	Determines whether specified year is a leap year	B-195
rmdyjul()	Returns an internal DATE from month, day, and year	B-196
rstrdate()	Converts a character string format to an internal DATE	B-204
rtoday()	Returns a system date as an internal DATE	B-205

When you compile your Informix ESQL/C program with the **esql** command, **esql** automatically links these functions into your program.

The SQL DATETIME and INTERVAL Data Types

Informix ESQL/C supports two data types that can hold information about time values:

- The **datetime** data type, which encodes an instant in time as a calendar date and a time of day.
- The **interval** data type, which encodes a span of time.

Table 6-1 summarizes these two time data types.

Table 6-1. ESQL/C Time Data Types

SQL Data Type	ESQL/C Data Type	C typedef Name	Sample Declaration
DATETIME	**datetime**	dtime_t	EXEC SQL BEGIN DECLARE SECTION;
			datetime year to day sale;
			EXEC SQL END DECLARE SECTION;
INTERVAL	**interval**	intrvl_t	EXEC SQL BEGIN DECLARE SECTION;
			interval hour to second test_num;
			EXEC SQL END DECLARE SECTION;

The header file **datetime.h** contains the **dtime_t** and **intrvl_t** structures, along with a number of macro definitions that you can use to compose qualifier values. Include this file in all C source files that use any **datetime** or **interval** host variables:

```
EXEC SQL include datetime;
```

The **decimal.h** header file defines the type **dec_t**, which is a component of the **dtime_t** and **intrvl_t** structures.

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**. Avoid the following declarations:

```
EXEC SQL BEGIN DECLARE SECTION;
   datetime year;                        /* will cause an error */
   datetime year to day year, today;                    /* ambiguous */
EXEC SQL END DECLARE SECTION;
```

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 that its qualifier implies. Once you know the precision and scale, you know the storage format. For example, if you define a table column as DATETIME YEAR TO DAY, it 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)**.

If the default precision of the underlying decimal value is not appropriate, you can specify a different precision. For example, if you have a host variable of type **interval**, with the qualifier **day to day**, the default precision of the underlying decimal value is two digits. If you have intervals of one hundred or more days, this precision is not adequate. You can specify a precision of three digits as follows:

```
interval day(3) to day;
```

For more information about the DATETIME and INTERVAL data types, see the *IBM Informix Guide to SQL: Reference*.

The datetime Data Type

Use the **datetime** data type to declare host variables for database values of type DATETIME. You specify the accuracy of the **datetime** data type with a *qualifier*. For example, the qualifier in the following declaration is **year to day**:

```
datetime year to day sale;
```

As a host variable, a **dtime_t**. structure represents a **datetime** value:

```
typedef struct dtime {
   short dt_qual;
   dec_t dt_dec;
} dtime_t;
```

The **dtime** structure and **dtime_t** typedef have two parts. Table 6-2 lists these parts.

Table 6-2. Fields in the dtime Structure

Field	Description
dt_qual	Qualifier of the datetime value
dt_dec	Digits of the fields of the datetime value This field is a **decimal** value.

Declare a host variable for a DATETIME column with the **datetime** data type followed by an optional qualifier, as the following example shows:

```
EXEC SQL include datetime;
:
:

EXEC SQL BEGIN DECLARE SECTION;
   datetime year to day holidays[10];
   datetime hour to second wins, places, shows;
   datetime column6;
EXEC SQL END DECLARE SECTION;
```

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 with the macros shown in Table 6-4 on page 6-5.

The interval Data Type

Use the **interval** data type to declare host variables for database values of type INTERVAL. You specify the accuracy of the **interval** data type with a qualifier. The qualifier in the following declaration is **hour to second**:

```
interval hour to second test_run;
```

As a host variable, an **intrvl_t**. represents an **interval** value:

```
typedef struct intrvl {
   short in_qual;
   dec_t in_dec;
} intrvl_t;
```

The **intrvl** structure and **intrvl_t** typedef have two parts. Table 6-3 lists these parts.

Table 6-3. Fields in the intrvl Structure

Field	Description
in_qual	Qualifier of the interval value
in_dec	Digits of the fields of the interval value This field is a **decimal** value.

To declare a host variable for an INTERVAL column, use the **interval** data type followed by an optional qualifier, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
   interval day(3) to day accrued_leave, leave_taken;
   interval hour to second race_length;
   interval scheduled;
EXEC SQL END DECLARE SECTION;
```

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 with the macros described in the following section.

Macros for datetime and interval Data Types

In addition to the **datetime** and **interval** data structures, the **datetime.h** file defines the macro functions shown in Table 6-4 for working directly with qualifiers in binary form.

Table 6-4. Qualifier Macros for datetime and interval Data Types

Name of Macro	Description
TU_YEAR	Time unit for the YEAR qualifier field
TU_MONTH	Time unit for the MONTH qualifier field
TU_DAY	Time unit for the DAY qualifier field
TU_HOUR	Time unit for the HOUR qualifier field
TU_MINUTE	Time unit for the MINUTE qualifier field
TU_SECOND	Time unit for the SECOND qualifier field
TU_FRAC	Time unit for the leading qualifier field of FRACTION
TU_Fn	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_LEN(q)	Returns the length in digits of the qualifier q
TU_FLEN(f)	Returns the length in digits of the first field, f, of an **interval** qualifier
TU_ENCODE(p,f,t)	Creates a qualifier from the first field number f with precision p and trailing field number t
TU_DTENCODE(f,t)	Creates a **datetime** qualifier from the first field number f and trailing field number t
TU_IENCODE(p,f,t)	Creates an **interval** qualifier from the first field number f with precision p and trailing field number t

For example, if your program does not provide an **interval** qualifier in the host-variable declaration, you need to use the **interval** qualifier macros to initialize and set the **interval** host variable. In the following example, the **interval** variable gets a **day to second** qualifier. The precision of the largest field in the qualifier, **day**, is set to 2:

```
/* declare a host variable without a qualifier */
EXEC SQL BEGIN DECLARE SECTION;
    interval inv1;
EXEC SQL END DECLARE SECTION;
:

/* set the interval qualifier for the host variable */
inv1.in_qual = TU_IENCODE(2, TU_DAY, TU_SECOND);
:

/* assign values to the host variable */
incvasc ("5 2:10:02", &inv1);
```

Fetching and Inserting DATETIME and INTERVAL Values

When an application fetches or inserts a DATETIME or INTERVAL value, Informix ESQL/C must ensure that the qualifier field of the host variable is valid:

- When an application fetches a DATETIME value into a **datetime** host variable or inserts a DATETIME value from a **datetime** host variable, it must ensure that the **dt_qual** field of the **dtime_t** structure is valid.
- When an application fetches an INTERVAL value into an **interval** host variable or inserts an INTERVAL value from an **interval** host variable, it must ensure that the **in_qual** field of the **intrvl_t** structure is valid.

Fetching and Inserting into datetime Host Variables

When an application uses a **datetime** host variable to fetch or insert a DATETIME value, Informix ESQL/C must find a valid qualifier in the **datetime** host variable. Informix ESQL/C takes one of the following actions, based on the value of the **dt_qual** field in the **dtime_t** structure that is associated with the host variable:

- When the **dt_qual** field contains a valid qualifier, Informix ESQL/C extends the column value to match the **dt_qual** qualifier.

   Extending is the operation of adding or dropping fields of a DATETIME value to make it match a given qualifier. You can explicitly extend DATETIME values with the SQL EXTEND function and the Informix ESQL/C **The dtextend() function** function.

- When the **dt_qual** field does *not* contain a valid qualifier, Informix ESQL/C takes different actions for a fetch and an insert:

   – For a fetch, Informix ESQL/C uses the DATETIME column value and its qualifier to initialize the **datetime** host variable.

      Zero (0) is an invalid qualifier. Therefore, if you set the **dt_qual** field to zero, you can ensure that Informix ESQL/C uses the qualifier of the DATETIME column.

   – For an insert, Informix ESQL/C cannot perform the insert or update operation.

      Informix ESQL/C sets the SQLSTATE status variable to an error-class code (and SQLCODE to a negative value) and the update or insert operation on the DATETIME column fails.

Fetching and Inserting into interval Host Variables

When an application uses an **interval** host variable to fetch or insert an INTERVAL value, Informix ESQL/C must find a valid qualifier in the **interval** host variable. Informix ESQL/C takes one of the following actions, based on the value of the **in_qual** field of the **intrvl_t** structure that is associated with the host variable:

- When the **in_qual** field contains a valid qualifier, Informix ESQL/C checks it for compatibility with the qualifier from the INTERVAL column value.

   The two qualifiers are compatible if they belong to the same interval class: either **year to month** or **day to fraction**. If the qualifiers are incompatible, Informix ESQL/C sets the SQLSTATE status variable to an error-class code (and SQLCODE is set to a negative value) and the select, update, or insert operation fails.

   If the qualifiers are compatible but not the same, Informix ESQL/C extends the column value to match the **in_qual** qualifier. *Extending* is the operation of adding or dropping fields within one of the interval classes of an INTERVAL value to make it match a given qualifier. You can explicitly extend INTERVAL values with the Informix ESQL/C **The invextend() function** function.

- When the **in_qual** field does *not* contain a valid qualifier, Informix ESQL/C takes different actions for a fetch and an insert:
  - For a fetch, if the **in_qual** field contains zero or is not a valid qualifier, Informix ESQL/C uses the INTERVAL column value and its qualifier to initialize the **interval** host variable.
  - For an insert, if the **in_qual** field is not compatible with the INTERVAL column or if it does not contain a valid value, Informix ESQL/C cannot perform the insert or update operation.

    Informix ESQL/C sets the SQLSTATE status variable to an error-class code (and SQLCODE is set to a negative value) and the update or insert operation on the INTERVAL column fails.

Implicit Data Conversion

You can fetch a DATETIME or INTERVAL column value into a character (**char**, **string**, or **fixchar**) host variable. Informix ESQL/C converts the DATETIME or INTERVAL column value to a character string before it stores it in the character host variable. This character string conforms to the ANSI SQL standards for DATETIME and INTERVAL values. If the host variable is too short, Informix ESQL/C sets **sqlca.sqlwarn.sqlwarn1** to W, fills the host variable with asterisk (*) characters, and sets any indicator variable to the length of the untruncated character string.

You can also insert a DATETIME or INTERVAL column value from a character (**char**, **string**, **fixchar**, or **varchar**) host variable. Informix ESQL/C uses the data type and qualifiers of the column value to convert the character value to a DATETIME or INTERVAL value. It expects the character string to contain a DATETIME or INTERVAL value that conforms to ANSI SQL standards.

If the conversion fails, Informix ESQL/C sets the SQLSTATE status variable to an error-class code (and SQLCODE status variable to a negative value) and the update or insert operation fails.

For more information, see "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

Important: IBM Informix products do not support automatic data conversion from DATETIME and INTERVAL column values to numeric (**double**, **int**, and so on) host variables. Nor do IBM Informix products support automatic data conversion from numeric (**double**, **int**, and so on) or **date** host variables to DATETIME and INTERVAL column values.

ANSI SQL standards for DATETIME and INTERVAL values

The ANSI SQL standards specify qualifiers and formats for character representations of DATETIME and INTERVAL values. The standard qualifier for a DATETIME value is YEAR TO SECOND, and the standard format is as follows:

```
YYYY-MM-DD HH:MM:SS
```

The standards for an INTERVAL value specify the following two *classes* of intervals:

- The YEAR TO MONTH class has the format: `YYYY-MM`

  A subset of this format is also valid: for example, just a month interval.
- The DAY TO FRACTION class has the format: `DD HH:MM:SS.F`

  Any subset of contiguous fields is also valid: for example, MINUTE TO FRACTION.

Data conversion for datetime and interval values

You can use the Informix ESQL/C library functions **dtcvasc()**, **dtcvfmtasc()**, **dttoasc()**, and **dttofmtasc()** to explicitly convert between DATETIME column values and character strings. To explicitly convert between INTERVAL column values and character strings, you can use the Informix ESQL/C library functions **incvasc()**, **incvfmtasc()**, **intoasc()**, and **intofmtasc()**.

For example, you can perform conversions between the DATETIME and DATE data types with Informix ESQL/C library functions and intermediate strings.

To convert a DATETIME value to a DATE value:

1. Use the **dtextend()** function to adjust the DATETIME qualifier to *year to day*.
2. Apply the **dttoasc()** function to create a character string in the form yyyy‑mm‑dd.
3. Use The **rdefmtdate()** function with a pattern argument of *yyyy-mm-dd* to convert the string to a DATE value.

To convert a DATE value to a DATETIME value:

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

Support of Non-ANSI DATETIME Formats

ESQL/C supports conversions from a data-time string in a non-ANSI format to the DATETIME data type. This makes it easier to upgrade from Asian Language Support (ALS) client/server products to Global Language Support (GLS) client/server products.

The USE_DTENV environment variable

To support compatibility with earlier versions, ESQL/C uses the USE_DTENV environment variable to activate support for non-ANSI date-time formats.

When the USE_DTENV environment variable is enabled, the following order or precedence is used:

1. DBTIME
2. GL_DATETIME
3. CLIENT_LOCALE
4. LC_TIME
5. LANG (if LC_TIME is not set)
6. ANSI format

When enabled, the USE_DTENV environment variable is passed from the ESQL/C program to the database server. Enabling it for the database server only will have no effect. You must set it for the ESQL/C client program, which then passes it to the database server.

If the database server does not support non-ANSI date-time formats, you should not set the USE_DTENV for the ESQL/C client program.

You must set this environment variable to display localized DATETIME values correctly in a database that uses a non-default locale, and for which the GL_DATETIME environment variable has a non-default setting.

DATETIME and INTERVAL Library Functions

You must use the following IBM Informix ESQL/C library functions for the **datetime** and **interval** data types to perform all operations on those types of values. The following C functions are available in Informix ESQL/C to handle **datetime** and **interval** host variables.

Function Name	Description	See
dtaddinv()	Adds an interval value to a **datetime** value	B-45
dtcurrent()	Gets the current date and time	B-46
dtcvasc()	Converts an ANSI-compliant character string to a **datetime** value	B-47
dtcvfmtasc()	Converts a character string with a specified format to a datetime value	B-49
dtextend()	Changes the qualifier of a datetime value	B-52
dtsub()	Subtracts one datetime value from another	B-53
dtsubinv()	Subtracts an interval value from a **datetime** value	B-55
dttoasc()	Converts a datetime value to an ANSI-compliant character string	B-56
dttofmtasc()	Converts a datetime value to a character string with a specified format	B-58
incvasc()	Converts an ANSI-compliant character string to an **interval** value	B-156
incvfmtasc()	Converts a character string with a specified format to an interval value	B-158
intoasc()	Converts an interval value to an ANSI-compliant character string	B-160
intofmtasc()	Converts an interval value to a character string with a specified format	B-160
invdivdbl()	Divides an interval value by a numeric value	B-161
invdivinv()	Divides an interval value by another **interval** value	B-163
invextend()	Extends an interval value to a different **interval** qualifier	B-165
invmuldbl()	Multiplies an interval value by a numeric value	B-167

For more information about operations on the SQL DATETIME and INTERVAL data types, see the *IBM Informix Guide to SQL: Reference*.

Chapter 7. Working with simple large objects

A simple large object is a large object that is stored in a blobspace on disk and is not recoverable. Simple large objects include the TEXT and BYTE data types. The TEXT data type stores any kind of text data. The BYTE data type can store any kind of binary data in an undifferentiated byte stream.

These topics describe the following information about simple large objects:
- Choosing whether to use a simple large object or a smart large object in your IBM Informix ESQL/C application
- Programming with simple large objects, including how to declare host variables and how to use the locator structure
- Locating simple large objects in memory
- Locating simple large objects in files, both open files and named files
- Locating simple large objects at a user-defined location
- Reading and writing simple large objects to optical disc

The end of this chapter presents an annotated example program called **dispcat_pic**. The **dispcat_pic** sample program demonstrates how to read and display the **cat_descr** and **cat_picture** simple-large-object columns from the **catalog** table of the **stores7** demonstration database.

For information about the data types available in an IBM Informix ESQL/C program, see Chapter 3, "Informix ESQL/C data types," on page 3-1. For information about the TEXT and BYTE data types, as well as other SQL data types, see the *IBM Informix Guide to SQL: Reference*.

Choosing a large-object data type

If you use IBM Informix as your database server, you can choose between using simple large objects or smart large objects.

IBM Informix supports simple large objects primarily for compatibility with earlier versions of IBM Informix applications. When you write new applications that need to access large objects, use smart large objects to hold character (CLOB) and binary (BLOB) data.

The following table summarizes the advantages that smart large objects present over simple large objects:

Large-Object Feature	Simple Large Objects	Smart Large Objects
Maximum size of data	2 gigabytes	4 terabytes
Data accessibility	No random access to data	Random access to data
Reading the large object	The database server reads a simple large object on an *all or nothing* basis.	Library functions provide access that is similar to accessing an operating-system file. You can access specified portions of the smart large object.
Writing the large object	The database server updates a simple large object on an *all or nothing* basis.	The database server can rewrite only a portion of a smart large object.
Data logging	Data logging is always on.	Data logging can be turned on and off.

For more information about how to use smart large objects, see Chapter 8, "Working with smart large objects," on page 8-1. The remainder of this chapter discusses how to use simple large objects.

Programming with Simple Large Objects

Informix ESQL/C supports SQL simple large objects and the data types TEXT and BYTE with the **loc_t** data type.

> **Tip:** You cannot use literal values in an INSERT or UPDATE statement to put simple-large-object data into a TEXT or BYTE column. To insert values into a simple large object, you can use the LOAD statement from DB-Access or **loc_t** host variables from an Informix ESQL/C client application.

Because of the potentially huge size of simple-large-object data, the Informix ESQL/C program does not store the data directly in a **loc_t** host variable. Instead, the **loc_t** structure is a *locator structure*. It does not contain the actual data; it contains information about the size and location of the simple-large-object data. You choose whether to store the data in memory, an operating-system file, or even user-defined locations.

To use simple-large-object variables in an Informix ESQL/C program, take the following actions:
- Declare a host variable with the **loc_t** data type
- Access the fields of the **loc_t** locator structure

Declaring a Host Variable for a Simple Large Object

Use the **loc_t** data type to declare host variables for database values of type TEXT or BYTE. You declare a host variable for a simple-large-object column with the data type **loc_t**, as shown in the following example:

```
EXEC SQL include locator;
   :

EXEC SQL BEGIN DECLARE SECTION;
   loc_t text_lob;
   loc_t byte_lob;
EXEC SQL END DECLARE SECTION;
```

A locator variable with a TEXT data type has the **loc_type** field of the locator structure set to SQLTEXT. For a BYTE variable, **loc_type** is SQLBYTE. For more information about the fields of the locator structure, see "The Fields of the Locator Structure" on page 7-5.

Tip: The **sqltypes.h** header file defines both SQLTEXT and SQLBYTE. Therefore, make sure you include **sqltypes.h** before you use these constants.

From an Informix ESQL/C program, you can both select and insert simple-large-object data into **loc_t** host variables. You can also select only portions of a simple-large-object variable with subscripts on the simple-large-object column name. These subscripts can be coded into the statement as shown in the following example:

```
EXEC SQL declare catcurs cursor for
   select catalog_num, cat_descr[1,10]
   from catalog
   where manu_code = 'HSK';
EXEC SQL open catcurs;
while (1)
   {
   EXEC SQL fetch catcurs into :cat_num, :cat_descr;

   .
   .
   .

   }
```

Subscripts can also be passed as input parameters as the following code fragment shows:

```
EXEC SQL prepare slct_id from
   'select catalog_num, cat_descr[?,?] from catalog \
   where catalog_num = ?'
EXEC SQL execute slct_id into :cat_num, :cat_descr
   using :n, :x, :cat_num;
```

Accessing the Locator Structure

In an Informix ESQL/C program, you use a *locator structure* to access simple-large-object values. The locator structure is the host variable for TEXT and BYTE columns when they are stored in or retrieved from the database. This structure describes the location of a simple-large-object value for the following two database operations:

- When the program inserts the simple large object into the database, the locator structure identifies the *source* of the simple-large-object data to insert.

   It is recommended that you initialize the data structure before using it, as in the following example

   ```
   byfill(&blob1, sizeof(loc_t), 0);
   where blob1 is declared as --
   EXEC SQL BEGIN DECLARE SECTION;
   loc_t blob1;
   EXEC SQL END DECLARE SECTION;
   ```

   This ensures that all variables of the data structure have been initialized and will avoid inconsistencies

- When the program selects the simple large object from the database, the locator structure identifies the *destination* of the simple-large-object data.

The **locator.h** header file defines the locator structure, called **loc_t**. Figure 7-1 shows the definition of the **loc_t** locator structure from the **locator.h** file.

```
typedef struct tag_loc_t
    {
    int2 loc_loctype;          /* USER: type of locator - see below      */
    union                      /* variant on 'loc'                       */
        {
        struct                 /* case LOCMEMORY                         */
            {
            int4  lc_bufsize;      /* USER: buffer size                 */
            char *lc_buffer;       /* USER: memory buffer to use        */
            char *lc_currdata_p;   /* INTERNAL: current memory buffer   */
            mint   lc_mflags;      /* USER/INTERNAL: memory flags          */
                                   /*                  (see below)      */
            } lc_mem;

        struct                 /* cases LOCFNAME & LOCFILE               */
            {
            char *lc_fname;      /* USER: file name                    */
            mint   lc_mode;      /* USER: perm. bits used if creating    */
            mint   lc_fd;        /* USER: os file descriptior            */
            int4  lc_position;   /* INTERNAL: seek position            */
            } lc_file;
        } lc_union;

    int4  loc_indicator;       /* USER/SYSTEM: indicator                 */
    int4  loc_type;            /* SYSTEM: type of blob                   */
    int4  loc_size;            /* USER/SYSTEM: num bytes in blob or -1    */
    mint  loc_status;          /* SYSTEM: status return of locator ops   */
    char *loc_user_env;        /* USER: for the user's PRIVATE use       */
    int4  loc_xfercount;       /* INTERNAL/SYSTEM: Transfer count        */
            /* USER: open function              */
    mint (*loc_open)(struct tag_loc_t *loc, mint flag, mint bsize);
;           /* USER: close function             */
    mint (*loc_close)(struct tag_loc_t *loc)
;            /* USER: read function                */
    mint (*loc_read)(struct tag_loc_t *loc, char *buffer, mint buflen)
;           /* USER: write function             */
    mint (*loc_write)(struct tag_loc_t *loc, char *buffer, mint buflen)
            /* USER/INTERNAL: see flag definitions below */
    mint   loc_oflags;
    } loc_t;
```

Figure 7-1. Declaration of loc_t in the locator.h Header File

In Figure 7-1 on page 7-5, the following comments in the **locator.h** file indicate how the fields are used in the locator structure.

Comment	Description
USER	The Informix ESQL/C program sets the field, and the Informix ESQL/C libraries inspect the field.
SYSTEM	The Informix ESQL/C libraries set the field, and the Informix ESQL/C program inspects the field.
INTERNAL	The field is a work area for the Informix ESQL/C libraries, and the Informix ESQL/C program does *not* need to examine the field.

Informix ESQL/C does *not* automatically include the **locator.h** header file in an Informix ESQL/C program. You must include the **locator.h** header file in any Informix ESQL/C program that defines simple-large-object variables.

```
EXEC SQL include locator;
```

The Fields of the Locator Structure

The locator structure has the following parts:

- The **loc_loctype** field identifies the location of the simple-large-object data. It also indicates the variant type of the **lc_union** structure.

For more information about **loc_loctype**, see "Locations for Simple-Large-Object Data" on page 7-6.

- The **lc_union** structure is a **union** (overlapping variant structures) structure.

  The variant in use depends on where Informix ESQL/C can expect to find the simple large object at runtime. For more information about this structure, see "Locating Simple Large Objects in Memory" on page 7-7 and "Locating Simple Large Objects in Files" on page 7-13.

- Several fields are common to all types of simple-large-object variables.

Table 7-1 lists the fields in the locator structure common to all simple-large-object locations.

Table 7-1. Fields in Locator Structure Common to all Simple-Large-Object Data Locations

Field	Data Type	Description
loc_indicator	4-byte integer	A value of **-1** in the **loc_indicator** field indicates a null simple-large-object value. The Informix ESQL/C program can set the field to indicate insertion of a null value; Informix ESQL/C libraries set it on a select or fetch.
		For consistent behavior on various platforms, it is advised to set the value of the indicator to 0 or -1. If indicator is not set you can experience inconsistent behavior. The value set in the indicator field takes the higher precedence when set.
		You can also use the **loc_indicator** field to indicate an error when your program selects into memory. If the simple large object to be retrieved does not fit in the space provided, the **loc_indicator** field contains the actual size of the simple large object.
loc_size	4-byte integer	Contains the size of the simple-large-object data in bytes. This field indicates the amount of simple-large-object data that the Informix ESQL/C libraries read or write. The Informix ESQL/C program sets **loc_size** when it inserts a simple large object in the database; the Informix ESQL/C libraries set **loc_size** after it selects or fetches a simple large object.
loc_status	mint	Indicates the status of the last locator operation. The Informix ESQL/C libraries set **loc_status** to zero when a locator operation is successful and to a negative value when an error occurs. The SQLCODE variable also contains this status value.
loc_type	4-byte integer	Specifies whether the data type of the variable is TEXT (SQLTEXT) or BYTE (SQLBYTES). The **sqltypes.h** header file defines SQLTEXT and SQLBYTES.

Locations for Simple-Large-Object Data

Before your Informix ESQL/C program accesses a simple-large-object column, it must determine where the simple-large-object data is located. To specify whether the simple large object is located in memory or in a file, specify the contents of the **loc_loctype** field of the locator structure. Table 7-2 shows the possible locations for simple-large-object data.

Table 7-2. Possible Locations for Simple-Large-Object Data

Value of loc_loctype Field	Location of Simple-Large-Object DataS	See
LOCMEMORY	In memory	7-7
LOCFILE	In an open file	7-14
LOCFNAME	In a named file	7-18
LOCUSER	At a user-defined location	7-22

Set **loc_loctype** after you declare the locator variable and before this declared variable receives a simple-large-object value.

The **locator.h** header file defines the LOCMEMORY, LOCFILE, LOCFNAME, and LOCUSER location constants. In your Informix ESQL/C program, use these constant names rather than their constant values when you assign values to **loc_loctype**.

In a client-server environment, Informix ESQL/C locates the simple large object on the client computer (the computer on which the application runs).

Locating Simple Large Objects in Memory

To have IBM Informix ESQL/C locate the TEXT or BYTE data in primary memory, set the **loc_loctype** field of the locator structure to LOCMEMORY as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    loc_t my_simple_lo;
EXEC SQL END DECLARE SECTION;
    .
    .
    .
my_simole_lo.loc_loctype = LOCMEMORY;
```

When you use memory as a simple-large-object location, a locator structure uses the **lc_mem** structure of the **lc_union** structure. Table 7-3 summarizes the **lc_union.lc_mem** fields.

Table 7-3. Fields in lc_union.lc_mem Structure Used for Simple Large Objects Located in Memory

Field	Data Type	Description
lc_bufsize	4-byte integer	The size, in bytes, of the buffer to which the lc_buffer field points. For more information, see "Allocating the Memory Buffer" on page 7-8.
lc_buffer	char *	The address of the buffer to hold the simple large-object value. Your Informix ESQL/C program must allocate the space for this buffer and store its address here in **lc_buffer**. For more information, see "Allocating the Memory Buffer" on page 7-8.
lc_currdata_p	char *	The address of the system buffer. *This is an internal field and must not be modified by the* Informix ESQL/C *program.*
lc_mflags	mint	The flags to use when you allocate memory.

The **locator.h** file provides the following macro shortcuts to use when you access fields in **lc_union.lc_mem**:

```
#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
```

Tip: It is recommended that you use these shortcut names when you access the locator structure. The shortcut names improve code readability and reduce coding errors. This publication uses these shortcut names when it refers to the **lc_bufsize**, **lc_buffer**, **lc_currdata_p**, and **lc_mflags** fields of the **lc_union.lc_mem** structure.

The **demo** directory contains the following two sample Informix ESQL/C programs that demonstrate how to handle simple-large-object data located in memory:

- The **getcd_me.ec** program selects a simple large object into memory.
- The **updcd_me.ec** program inserts a simple large object from memory.

These programs assume the **stores7** database as the default database for the simple-large-object data. The user can specify another database (on the default database server) as a command-line argument.

```
getcd_me mystores
```

The **getcd_me** and **updcd_me** programs are briefly explained in 7-9 and 7-11.

Allocating the Memory Buffer

When your program selects simple-large-object data into memory, Informix ESQL/C uses a *memory buffer*. Before your program fetches TEXT or BYTE data, you must set the **loc_bufsize** (**lc_union.lc_mem.lc_bufsize**) field as follows to indicate how Informix ESQL/C allocates this memory buffer:

- If you set the **loc_bufsize** to **-1**, Informix ESQL/C allocates the memory buffer to hold the simple-large-object data.
- If you set the **loc_bufsize** to a value that is *not* **-1**, Informix ESQL/C assumes that the program handles memory-buffer allocation and deallocation.

Warning: When you locate simple large objects in memory, you must always set loc_mflags (**lc_union.lc_mem.lc_mflags**) and **loc_oflags** to 0 (zero) initially.

A memory buffer that the ESQL/C libraries allocate

When you set **loc_bufsize** to **-1**, Informix ESQL/C allocates the memory buffer on a fetch or select. Informix ESQL/C uses the **malloc()** system call to allocate the memory buffer to hold a single simple-large-object value. (If it cannot allocate the buffer, Informix ESQL/C sets the **loc_status** field to **-465** to indicate an error.) When the select (or the *first* fetch) completes, Informix ESQL/C sets **loc_buffer** to the address of the buffer and both **loc_bufsize** and **loc_size** to the size of the fetched simple large object to update the locator structure.

To fetch subsequent simple-large-objects whose data is of larger or smaller size, set **loc_mflags** to the LOC_ALLOC constant (that **locator.h** defines) to request that Informix ESQL/C reallocate a new memory buffer. Leave **loc_bufsize** to the size of the currently allocated buffer.

Warning: If you do not set **loc_mflags** to LOC_ALLOC after the initial fetch, Informix ESQL/C does not release the memory it has allocated for the **loc_buffer** buffer. Instead, it allocates a new buffer for subsequent

fetches. This situation can cause your program size to grow for each fetch unless you explicitly free the memory allocated to each **loc_buffer** buffer. If your application runs on a Windows operating system and uses the mutli-threaded library then use the **SqlFreeMem()** Informix ESQL/C function to free it. Otherwise use the **free()** system call.

When you set **loc_mflags** to LOC_ALLOC, Informix ESQL/C handles memory allocation as follows:

- If the size of the simple-large-object data increases, Informix ESQL/C frees the existing buffer and allocates the necessary memory.

  If this reallocation occurs, Informix ESQL/C alters the memory address at which it stores simple-large-object data. Therefore, if you reference the address in your programs, your program logic must account for the address change. Informix ESQL/C also updates the **loc_bufsize** and **loc_size** field to the size of the fetched simple large object.

- If the size of the data decreases, Informix ESQL/C does not need to reallocate the buffer.

  After the fetch, the **loc_size** field indicates the size of the fetched simple large object while the **loc_bufsize** field still contains the size of the allocated buffer.

Informix ESQL/C frees the allocated memory when it fetches the next simple-large-object value. Therefore, Informix ESQL/C does not explicitly free the last simple-large-object value fetched until your program disconnects from the database server.

For an example in which **loc_bufsize** is set to **-1**, see "Selecting a simple large object into memory" on page 7-9.

A memory buffer that the program allocates

If you wish to handle your own memory allocation for simple large objects, use the **malloc()** system call to allocate the memory and then set the following fields in the locator structure:

- Before a select or fetch of a TEXT or BYTE column, set the **loc_buffer** field to the address of the allocated memory buffer, and set the **loc_bufsize** field to the size of the memory buffer.

- Before an insert of a TEXT or BYTE column, set the same fields as for a select or fetch. In addition, set **loc_size** to the size of the data to be inserted in the database.

If the fetched data does not fit in the allocated buffer, the Informix ESQL/C libraries set **loc_status** (and SQLCODE) to a negative value (**-451**) and put the actual size of the data in **loc_indicator**. If the fetched data does fit, Informix ESQL/C sets **loc_size** to the size of the fetched data.

Important: When you allocate your own memory buffer, also free the memory when you are finished selecting or inserting simple large objects. Informix ESQL/C does not free this memory because it has no way to determine when you are finished with the memory. Because you have allocated the memory with **malloc()**, you can use the **free()** system call to free the memory.

Selecting a simple large object into memory

The **getcd_me** sample program from the **demo** directory shows how to select a simple large object from the database into memory. Figure 7-2 shows a code

excerpt that selects the **cat_descr** TEXT column of the **catalog** table into memory and then displays it.

```
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 */
cat_descr.loc_mflags = 0;               /* set loc_mflags to 0 */
EXEC SQL select catalog_num, cat_descr  /* look up catalog number */
   into :cat_num, :cat_descr from catalog
   where catalog_num = :cat_num;
if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100)      /* if not found */
   {
   printf("\nCatalog number %ld not found in catalog table\n",
      cat_num);
   if(!more_to_do())                            /* More to do? */
       break;                                   /* no, terminate loop */
    else
       continue;                      /* yes */
   }
if(ret < 0)
   {
   printf("\nSelect for catalog number %ld failed\n", cat_num);
   EXEC SQL disconnect current;
   printf("GETCD_ME Sample Program over.\n\n");
   exit(1);
   }
prdesc();                                   /* if found, print cat_descr */
```

Figure 7-2. Code Excerpt from the getcd_me Sample Program

The program sets the **cat_descr** locator structure fields as follows:

- The **loc_loctype** field is set to LOCMEMORY so that Informix ESQL/C returns the **cat_descr** text in a memory buffer.
- The **loc_bufsize** field is set to **-1** to have Informix ESQL/C allocate the memory for the buffer. For more information, see "A memory buffer that the ESQL/C libraries allocate" on page 7-8.
- The **loc_oflags** field is set to 0 because the program does not use a file for the simple large object.
- You must always set the **loc_mflags** field to 0 when you locate a simple large object in memory.

After the SELECT or FETCH statement, the locator structure contains the following information:

- The **loc_buffer** field contains the address of the memory buffer.
- The **loc_bufsize** field contains the size of the **loc_buffer** buffer. This is the total amount of memory allocated for simple-large-object storage.
- The **loc_size** field contains the number of bytes of simple-large-object data in **loc_buffer**.
- The **loc_indicator** field contains **-1** if the selected simple-large-object value is null.
- The **loc_status** field contains the status of the operation: 0 for success and a negative value if an error has occurred. For information about possible errors, see "Allocating the Memory Buffer" on page 7-8.

The program excerpt in Figure 7-2 on page 7-10 calls **prdesc()** to display the text that the SELECT statement returned. For a description of the **prdesc()** function, see "Guide to the prdesc.c File" on page 7-41. If this program were to select a second simple large object, it would need to set the **loc_mflags** to the LOC_ALLOC constant before the second SELECT statement to prevent memory leaks.

The excerpt also displays the **cat_descr** column for a catalog number that the user enters. Figure 7-3 shows the user input and the output from the **cat_descr** column of the **stores7** demonstration database.

```
GETCD_ME Sample ESQL Program running.

Connected to stores7

This program requires you to enter a catalog number from the catalog
table. For example: '10001'. It then displays the content of the
cat_descr column for that catalog row. The cat_descr value is stored
in memory.

Enter a catalog number: 10004
Description for 10004:

Jackie Robinson signature glove. Highest professional quality, used by National League.

**** More? (y/n) ...
```

Figure 7-3. Sample Output from the getcd_me Sample Program

Inserting a Simple Large Object from Memory

The **updcd_me** sample program from the **demo** directory shows how to insert a simple large object from memory into the database. The program updates the **cat_descr** TEXT column of the **catalog** table from a memory buffer that contains text that the user enters. Figure 7-4 shows sample output as the user updates the **cat_descr** column of the **stores7** database.

```
Enter catalog number: 10004
Description for 10004:

Jackie Robinson signature ball. Highest professional quality, used by National League.

Update this description? (y/n) ...y

Enter description (max 255 chars):and press RETURN
Jackie Robinson home run ball, signed, 1955.

*** Update complete.
**** More?(y/n).... n
```

Figure 7-4. Sample Output from the updcd_me Sample Program

Figure 7-5 shows a code excerpt that illustrates how the **updcd_me** program uses the locator structure to update the **cat_descr** column from the text that is stored in memory.

```
/* Update? */
   ans[0] = ' ';
   while((ans[0] = LCASE(ans[0])) != 'y' && ans[0] != 'n')
      {
      printf("\nUpdate this description? (y/n) ...");
      getans(ans, 1);
      }
   if(ans[0] == 'y')                      /* if yes */
      {
      printf("Enter description (max of %d chars) and press RETURN\n",
         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);
      /* Update */
      EXEC SQL update catalog
         set cat_descr =:cat_descr
         where catalog_num = :cat_num;
 .
 .
 .
      }
```

Figure 7-5. Code Excerpt from the updcd_me Sample Program

The program sets the **cat_descr** locator structure fields as follows:

- The **loc_loctype** field is set to LOCMEMORY so that Informix ESQL/C reads the **cat_descr** text from a memory buffer.
- The **loc_buffer** field is set to **ans**, the address of the memory buffer that holds the simple-large-object value to be inserted.
- The **loc_bufsize** field is set to BUFFSZ, the size of the allocated **ans** memory buffer.
- The **loc_size** field is set to **strlen(ans)** + 1, the number of bytes in the memory buffer that currently holds the new simple-large-object value.

If you insert a null simple-large-object value, your program also needs to set the **loc_indicator** field to **-1**.

Figure 7-6 shows a code excerpt that illustrates how to use a locator structure in an INSERT statement.

```
char photo_buf[BUFFSZ];

EXEC SQL BEGIN DECLARE SECTION;
   char name[20];
   loc_t photo;
EXEC SQL END DECLARE SECTION;

photo.loc_loctype = LOCMEMORY; /* Photo resides in memory */
photo.loc_buffer = photo_buf; /* pointer to where it is */
photo.loc_size = BUFFSZ - 1;                      /* length of image*/

EXEC SQL insert into employee (name, badge_pic)
     values (:name, :photo);
```

Figure 7-6. Sample INSERT Operation from Primary Memory

After the UPDATE or INSERT statement, Informix ESQL/C updates the **loc_size** field with the number of bytes read from the memory buffer and sent to the database server. It also sets the **loc_status** field to indicate the status of the operation: 0 for success and a negative value if an error has occurred. For information about possible errors, see "Allocating the Memory Buffer" on page 7-8.

Locating Simple Large Objects in Files

You can locate simple-large-object data in the following types of files:

- An *open file* is one that has already been opened before the program accesses the simple-large-object data. The program provides a file descriptor as the location of the simple-large-object data.

- A *named file* is one that your program has not yet opened. The program provides a file name as the location of the simple-large-object data.

When you use a file as a simple-large-object location, a locator structure uses the **lc_file** structure for the **lc_union** structure. Table 7-4 summarizes the **lc_union.lc_file** fields.

Table 7-4. Fields in lc_union.lc_file Structure Used for Simple Large Objects Located in Files

Field	Data Type	Description
lc_fname	char *	The address of the path name string that contains the file for the simple-large-object data. The program sets this field when it uses named files for simple-large-object locations.
lc_mode	int	The permission bits to use to create a new file. This value is the third argument passed to the system open() function. For valid values of **lc_mode**, see your system documentation.
lc_fd	int	The file descriptor of the file that contains the simple-large-object data. The program sets this field when it uses open files.
lc_position	4-byte integer	The current seek position in the opened file. *This is an internal field and must not be modified by the ESQL/C program.*

The **locator.h** file provides the following macro shortcuts to use when you access simple large objects stored in files:

```
#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
```

Tip: It is recommended that you use these shortcut names when you access the locator structure. The shortcut names improve code readability and reduce coding errors. This publication uses these shortcut names when it refers to the **lc_fname**, **lc_fd**, and **lc_position** fields of the **lc_union.lc_file** structure.

File-Open Mode Flags

When you use files for simple-large-object data, also set the **loc_oflags** field of the locator structure. The **loc_oflags** field is of type **integer** and it contains the host-system file-open mode flags. These flags determine how the file is to be accessed once it is opened:

- LOC_RONLY is a mask for read-only mode. Use this value when you insert a simple large object into a file.

- LOC_WONLY is a mask for write-only mode. Use this value when you select a simple large object into a file and you want each selected simple large object to write over any existing data.
- LOC_APPEND is a mask for write mode. Use this value when you select a simple large object into a file and you want to append the value to the end of the file.

Error Returns in loc_status

One of these flags is passed to the **loc_open( )** function when Informix ESQL/C opens the file. Informix ESQL/C reads the data and writes it to the current location (which the **loc_position** field indicates) in the file. If Informix ESQL/C is unable to read or write to a file, it sets the **loc_status** field of the locator structure to **-463** or **-464**. If Informix ESQL/C is unable to close a file, it sets **loc_status** to **-462**. Informix ESQL/C updates the SQLCODE variable with this same value.

Locating Simple Large Objects in Open Files

To have Informix ESQL/C locate the TEXT or BYTE data in an open file, set the **loc_loctype** field of the locator structure to LOCFILE.

```
EXEC SQL BEGIN DECLARE SECTION;
    loc_t my_simple_lo;
EXEC SQL END DECLARE SECTION;
    .
    .
    .
my_simple_lo.loc_loctype = LOCFILE;
```

To use an open file as a simple-large-object location, your Informix ESQL/C program must open the desired file *before* it accesses the simple-large-object data. It must then store its file descriptor in the **loc_fd** field of the locator structure to specify this file as the simple-large-object location. The **loc_oflags** field should also contain a file-open mode flag to tell Informix ESQL/C how to access the file when it opens it. For a list of file-open mode flags, see "File-Open Mode Flags" on page 7-13.

The **demo** directory contains the following two sample Informix ESQL/C programs that demonstrate how to handle simple-large-object data located in an open file:

- The **getcd_of.ec** program selects a simple large object into an open file.
- The **updcd_of.ec** program inserts a simple large object from an open file.

These programs assume the **stores7** database as the default database for the simple-large-object data. The user can specify another database (on the default database server) as a command-line argument:

```
getcd_of mystores
```

Each of these programs is briefly explained in the following sections.

Selecting a Simple Large Object into an Open File

The **getcd_of** sample program from the **demo** directory shows how to select a simple large object from the database into an open file. Figure 7-7 shows a code excerpt that selects the **cat_descr** column into a file that the user specifies.

```
EXEC SQL BEGIN DECLARE SECTION;
    char db_name[30];
    mlong cat_num;
    loc_t cat_descr;
EXEC SQL END DECLARE SECTION;
    .
    .
    .

if((fd = open(descfl, O_WRONLY)) < 0)
    {
    printf("\nCan't open file: %s, errno: %d\n", descfl, errno);
    EXEC SQL disconnect current;
    printf("GETCD_OF Sample Program over.\n\n"):
        exit(1);
    }
/*
 * Prepare locator structure for select of cat_descr
 */
cat_descr.loc_loctype = LOCFILE;         /* set loctype for open file */
cat_descr.loc_fd = fd;                   /* load the file descriptor */
cat_descr.loc_oflags = LOC_APPEND;       /* set loc_oflags to append */
EXEC SQL select catalog_num, cat_descr   /* verify catalog number */
    into :cat_num, :cat_descr from catalog
    where catalog_num = :cat_num;
if(exp_chk2("SELECT", WARNNOTIFY) != 100)    /* if not found */
    printf("\nCatalog number %ld not found in catalog table\n",
        cat_num);
else
    {
    if(ret < 0)
        {
    .
    .
    .

        exit(1);
        }
    }
```

Figure 7-7. Code Excerpt from the getcd_of Sample Program

To prepare the locator structure for the SELECT statement, the **getcd_of** program
sets the **cat_descr** locator structure fields as follows:

- The **loc_loctype** field is set to LOCFILE to tell Informix ESQL/C to place the
  text for the **cat_descr** column in the open file.
- The **loc_fd** field is set to the **fd** file descriptor to identify the open file.
- The **loc_oflags** field is set to LOC_APPEND to specify that the data is to be
  appended to any data that already exists in the file.

To access the file descriptor (**loc_fd**) field of the locator structure, the **getcd_of**
program uses the name **cat_descr.loc_fd**. However, the actual name of this field in
the locator structure is as follows:

```
cat_descr.lc_union.lc_file.lc_fd
```

The shortcut name of **loc_fd** is defined as a macro in the **locator.h** file.

After Informix ESQL/C writes data to an open file, it sets the following fields of
the locator structure:

- The **loc_size** field contains the number of bytes written to the open file.
- The **loc_indicator** field contains -1 if the selected simple-large-object value is
  null.
- The **loc_status** field contains the status of the operation: 0 for success and a
  negative value if an error has occurred. For possible causes of the error, see
  "Error Returns in loc_status" on page 7-14.

Inserting a Simple Large Object from an Open File

The **updcd_of** sample program from the **demo** directory shows how to insert a simple large object from an open file into the database. The program updates the **cat_descr** TEXT column of the **catalog** table from an open file that contains a series of records; each consists of a catalog number and the text to update the corresponding **cat_descr** column. The program assumes that this input file has the following format:

```
\10001\
Dark brown leather first baseman's mitt.  Specify right-handed or
left-handed.

\10002\
Babe Ruth signature glove. Black leather. Infield/outfield style.
Specify right- or left-handed.
⋮
```

Figure 7-8 shows a code excerpt that illustrates how to use the locator structure to update the **cat_descr** column of the **catalog** table from an open file.

```
EXEC SQL BEGIN DECLARE SECTION;
   mlong cat_num;
   loc_t cat_descr;
EXEC SQL END DECLARE SECTION;
.
.
.

if ((fd = open(descfl, O_RDONLY)) < 0) /* open input file */
   {
.
.
.

   }
while(getcat_num(fd, line, sizeof(line)))    /* get cat_num line from file */
   {
.
.
.

   printf("\nReading catalog number %ld from file...\n", cat_num);
   flpos = lseek(fd, 0L, 1);
   length = getdesc_len(fd);
   flpos = lseek(fd, flpos, 0);

   /* lookup cat_num in catalog table */
   EXEC SQL select catalog_num
      into :cat_num from catalog
      where catalog_num = :cat_num;
   if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100) /* if not found */
      {
      printf("\nCatalog number %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_RONLY;      /* set file-open mode (read) */
   cat_descr.loc_size = length;           /* set size of simple large obj */

   /* update cat_descr column of catalog table */
   EXEC SQL update catalog set cat_descr = :cat_descr
      where catalog_num = :cat_num;
   if(exp_chk2("UPDATE", WARNNOTIFY) < 0)
      {
      EXEC SQL disconnect current;
      printf("UPDCD_OF Sample Program over.\n\n");
      exit(1);
      }
   printf("Update complete.\n");
   }
```

Figure 7-8. Code Excerpt from the updcd_of Sample Program

The **updcd_of** program opens the input file (**descfl**) that the user specified in
response to a prompt, calls the **getcat_num()** function to read a catalog number
from the file, and then calls the **getdesc_len()** function to determine the length of
the text for the update to the **cat_descr** column. The program performs a SELECT
statement to verify that the catalog number exists in the **catalog** table.

If this number exists, the **updcd_of** program prepares the locator structure as
follows to update **cat_descr** from the text in the open file:

- The **loc_loctype** field is set to LOCFILE to tell Informix ESQL/C that the
  cat_descr column is to be updated from an open file.
- The **loc_fd** field is set to **fd**, the file descriptor for the open-input file.
- The **loc_oflags** field is set to LOC_RONLY, the file-open mode flag for read-only
  mode.

- The **loc_size** field is set to **length**, the length of the incoming text for **cat_descr**.

If you insert a null simple-large-object value, your program also needs to set the **loc_indicator** field to **-1**.

The **updcd_of** program is then able to perform the database update. After Informix ESQL/C reads data from the open file and sends it to the database server, Informix ESQL/C updates the **loc_size** field with the number of bytes read from the open file and sent to the database server. Informix ESQL/C also sets the **loc_status** field to indicate the status of the operation: 0 for success and a negative value if an error has occurred. For possible causes of the error, see "Error Returns in loc_status" on page 7-14.

Locating Simple Large Objects in Named Files

To have Informix ESQL/C locate the TEXT or BYTE data in a named file, set the **loc_loctype** field of the locator structure to LOCFNAME, as shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
    loc_t my_simple_lo;
EXEC SQL END DECLARE SECTION;
    .
    .
    .

my_simple_lo.loc_loctype = LOCFNAME;
```

To use a named file as a simple-large-object location, your Informix ESQL/C program must specify a pointer to the file name in the **loc_fname** field of the locator structure. You must also set the **loc_oflags** field with a file-open mode flag to tell Informix ESQL/C how to access the file when it opens it. For a list of file-open mode flags, see "File-Open Mode Flags" on page 7-13.

To open a named file, Informix ESQL/C opens the file named in the **loc_fname** field with the mode flags that the **loc_oflags** field specifies. If this file does not exist, Informix ESQL/C creates it. Informix ESQL/C then puts the file descriptor of the open file in the **loc_fd** field and proceeds as if your program opened the file. If Informix ESQL/C cannot open this file, it sets the **loc_status** field (and SQLCODE) to **-461**. When the transfer is complete, Informix ESQL/C closes the file, which releases the file descriptor in the **loc_fd** field.

The **demo** directory contains the following two sample Informix ESQL/C programs that demonstrate how to handle simple-large-object data located in a named file:

- The **getcd_nf.ec** program selects a simple large object into a named file.
- The **updcd_nf.ec** program inserts a simple large object from a named file.

These programs assume the **stores7** database as the default database for the simple-large-object data. The user can specify another database (on the default database server) as a command-line argument as follows:

```
getcd_of mystores
```

Each of these programs is briefly explained in the following sections.

Selecting a Simple Large Object into a Named File

The **getcd_nf** sample program from the **demo** directory shows how to select a simple large object from the database into a named file. The following code excerpt prompts the user to enter a catalog number for the **catalog** table and the name of

the file to which the program writes the contents of the **cat_descr** column for that row. The program stores the name of the file in the **descfl** array. It then executes a SELECT statement to read the **cat_descr** TEXT column from the **catalog** table and write it to a file that the user specifies in response to a prompt.

Figure 7-9 shows a code excerpt from the **getcd_nf** sample program.

```
EXEC SQL BEGIN DECLARE SECTION;
    char db_name[30];
    mlong cat_num;
    loc_t cat_descr;
EXEC SQL END DECLARE SECTION;
   .
   .
   .
      printf("\nEnter a catalog number: ");      /* prompt for catalog number */
      getans(ans, 6);
      if(rstol(ans, &cat_num))                  /* cat_num string too long */
         {
         printf("\tCannot convert catalog number '%s' to integer\n", ans);
         continue;
         }
      while(1)
         {
         printf("Enter the name of the file to receive the description: ");
         if(!getans(ans, 15))
             continue;
         break;
         }
      strcpy(descfl, ans);
      break;
      }

   /*
    *  Prepare locator structure for select of cat_descr
    */
   cat_descr.loc_loctype = LOCFNAME;            /* set loctype for in memory */
   cat_descr.loc_fname = descfl;                /* load the addr of file name */
   cat_descr.loc_oflags = LOC_APPEND;           /* set loc_oflags to append */
   EXEC SQL select catalog_num, cat_descr              /* verify catalog number */
     into :cat_num, :cat_descr from catalog
     where catalog_num = :cat_num;
   if(exp_chk2("SELECT", WARNNOTIFY)  != 0 )   /* if error, display and quit */
     printf("\nSelect for catalog number %ld failed\n", cat_num);

   EXEC SQL disconnect current;
   printf("\nGETCD_NF Sample Program over.\n\n");
}
```

Figure 7-9. Code Excerpt from the getcd_nf Sample Program

The program sets the **cat_descr** locator structure fields as follows:
- The **loc_loctype** field contains LOCFNAME to tell Informix ESQL/C to place the text for the **cat_descr** column in a named file.
- The **loc_fname** field is the address of the **descfl** array to tell Informix ESQL/C to write the contents of the **cat_descr** column to the file named in **descfl**.
- The **loc_oflags** field, the file-open mode flags, is set to LOC_APPEND to tell Informix ESQL/C to append selected data to the existing file.

The **getcd_nf** program then executes the SELECT statement to retrieve the row. After Informix ESQL/C writes data to the named file, it sets the following fields of the locator structure:
- The **loc_size** field contains the number of bytes written to the file. If the Informix ESQL/C program fetches a null (or empty) simple-large-object column into a named file that already exists, it truncates the file.

- The **loc_indicator** field contains **-1** if the selected simple-large-object value is null.
- The **loc_status** field contains the status of the operation: **0** for success and a negative value if an error has occurred. For possible causes of the error, see "Error Returns in loc_status" on page 7-14.

Inserting a Simple Large Object from a Named File

The **updcd_nf** sample program from the **demo** directory shows how to insert a simple large object from a named file into the database. The program updates the **cat_descr** TEXT column from a named input file. The program assumes this input file has the following format:

```
Babe Ruth signature glove. Black leather. Infield/outfield
   style. Specify right- or left-handed.
```

Figure 7-10 shows a code excerpt that updates the **cat_descr** column in the **catalog** table from text in a named file.

```
EXEC SQL BEGIN DECLARE SECTION;
   mlong cat_num;
   loc_t cat_descr;
EXEC SQL END DECLARE SECTION;
 .
 .
 .
   cat_descr.loc_loctype = LOCMEMORY;                /* set loctype for in memory */
   cat_descr.loc_bufsize = -1;                       /* let server get memory */
   EXEC SQL select catalog_num, cat_descr            /* verify catalog number */
       into :cat_num, :cat_descr from catalog
      where catalog_num = :cat_num;

   /* if error,display and quit */
   if ((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100)
      {
      printf("\nCatalog number %ld not found in catalog table\n",
         cat_num);
      EXEC SQL disconnect current;
      printf("UPDCD_NF Sample Program over.\n\n");
      exit(1);
      }
   if(ret<0)
      {
      EXEC SQL disconnect current;
      printf("UPDCD_NF Sample Program over.\n\n");
      exit(1);
      }
   prdesc();                                         /* print current cat_descr */

   /* Update? */
   ans[0] = ' ';
   while((ans[0] = LCASE(ans[0]))  != 'y' && ans[0]  != 'n')
      {
      printf("Update this description? (y/n) ...");
      scanf("%1s", ans);
      }
   if(ans[0] == 'y')
      {
      cat_descr.loc_loctype = LOCFNAME;      /* set type to named file */
      cat_descr.loc_fname = descfl;          /* supply file name */
      cat_descr.loc_oflags = LOC_RONLY;      /* set file-open mode (read) */
      cat_descr.loc_size = -1;               /* set size to size of file */
      EXEC SQL update catalog
         set cat_descr = :cat_descr          /* update cat_descr column */
         where catalog_num = :cat_num;
      if(exp_chk2("UPDATE", WARNNOTIFY) < 0)   /* check status */
         {
         EXEC SQL disconnect current;
         printf("UPDCD_NF Sample Program over.\n\n");
         exit(1);
         }
      printf("Update complete.\n");
      }
```

Figure 7-10. Code Excerpt from the updcd_nf Sample Program

The **updcd_nf** program in Figure 7-10 on page 7-21 first performs a SELECT statement on the **catalog** table for a catalog number that the user enters in response to a prompt. The SELECT statement returns the **catalog_num** and **cat_descr** columns. The **prdesc()** function (7-41) displays the current content of **cat_descr**.

The program then asks whether the user wants to update this description. If the user answers yes (ans[0] = = 'y'), the **updcd_nf** program prepares the locator structure as follows to update the **cat_descr** column from text in a file that the user has specified:

• The **cat_descr.loc_loctype** field is set to LOCFNAME to indicate that the source of the update text is a named file.

- The **cat_descr.loc_fname** field is set to **descfl**, the name of the file that contains the simple-large-object data.
- The **cat_descr.loc_oflags** field is set to LOC_RONLY to tell Informix ESQL/C to open the file in read-only mode.
- The **cat_descr.loc_size** field is set to **-1** to tell Informix ESQL/C to transfer the simple large object all at once, not to transfer it in smaller pieces, one piece at a time. You can also set the **loc_oflags** field to the LOC_USEALL mask to perform this operation.

If you insert a null simple-large-object value, your program also needs to set the **loc_indicator** field to **-1**.

After Informix ESQL/C reads data from the named file and sends it to the database server, Informix ESQL/C updates the **loc_size** field with the number of bytes read from the named file and sent to the database server. Informix ESQL/C also sets the **loc_status** field to indicate the status of the operation: 0 for success and a negative value if an error has occurred. For possible causes of the error, see "Error Returns in loc_status" on page 7-14.

Using User-Defined Simple-Large-Object Locations

You can create your own versions of the **loc_open()**, **loc_read()**, **loc_write()**, and **loc_close()** functions to define your own location for simple-large-object data. A typical use for user-defined location functions is when the data needs to be translated in some manner before the application can use it. For example, if the data is compressed, the application must uncompress it before this data can be sent to the database. The application might even have a number of different translation functions that you can choose at runtime; it simply sets the appropriate function pointer to the desired translation function.

To have Informix ESQL/C use your own C functions to define the TEXT or BYTE data location, set the **loc_loctype** field of the locator structure to LOCUSER as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    ifx_loc_t my_simple_lo;
EXEC SQL END DECLARE SECTION;
:

my_simple_lo.loc_loctype = LOCUSER;
```

With a user-defined simple-large-object location, a locator structure uses the fields that Table 7-5 summarizes.

Table 7-5. Fields in the Locator Structure Used to Create User-Defined Location Functions

Field	Data Type	Description
loc_open	mint (*)()	A pointer to a user-defined open function that returns an integer value. For more information, see "The User-Defined Open Function" on page 7-24.
loc_read	mint (*)()	A pointer to a user-defined read function that returns an integer value. For more information, see "The User-Defined Read Function" on page 7-25.
loc_write	mint (*)()	A pointer to a user-defined write function that returns an integer value. For more information, see "The User-Defined Write Function" on page 7-26.
loc_close	mint (*)()	A pointer to a user-defined close function that returns an integer value. For more information, see "The User-Defined Close Function" on page 7-27.
loc_user_env	char *	The address of the buffer to hold data that a user-defined location function needs. For example, you can set loc_user_env to the address of a common work area.
loc_xfercount	4-byte integer	The number of bytes that the last transfer operation for the simple large object transferred.

With a user-defined simple-large-object location, a locator structure can use either the **lc_mem** structure or the **lc_file** structure of the **lc_union** structure. Table 7-3 on page 7-7 and Table 7-4 on page 7-13 summarize fields of the **lc_union.lc_mem** structure and **lc_union.lc_file** structure.

Selecting a Simple Large Object into a User-Defined Location

When your program *selects* a simple-large-object value, the Informix ESQL/C libraries must receive the data from the database server and transfer it to the Informix ESQL/C program. To do this, Informix ESQL/C performs the following steps:

1. Before the transfer, Informix ESQL/C calls the user-defined open function to initialize the user-defined location. The *oflags* argument of this open function is set to LOC_WONLY.
2. Informix ESQL/C receives the simple-large-object value from the database server and puts it into a program buffer.
3. Informix ESQL/C calls the user-defined write function to transfer the simple-large-object data from the program buffer to the user-defined location.

   Informix ESQL/C repeats steps 2 and 3 as many times as needed to transfer the entire simple-large-object value from the database server to the user-defined location.
4. After the transfer, Informix ESQL/C performs the clean-up operations that the user-defined close function specifies.

To select a simple large object into a user-defined location, set **loc_loctype** to LOCUSER and set the **loc_open**, **loc_write**, and **loc_close** fields so they contain the addresses of appropriate user-defined open, write, and close functions.

Inserting a Simple Large Object into a User-Defined Location

When your program *inserts* a simple-large-object value, the Informix ESQL/C libraries must transfer the data from the Informix ESQL/C program to the database server. To do this, Informix ESQL/C performs the following steps:

1. Before the transfer, Informix ESQL/C calls the user-defined open function to initialize the user-defined location. The *oflags* argument of this open function is set to LOC_RONLY.
2. Informix ESQL/C calls the user-defined read function to transfer the simple-large-object data from the user-defined location to the program buffer.
3. Informix ESQL/C sends the value in the program buffer to the database server.

   Informix ESQL/C repeats steps 2 and 3 as many times as needed to transfer the entire simple-large-object value from the user-defined location to the database server.
4. After the transfer, Informix ESQL/C performs the clean-up operations specified in the user-defined close function.

To insert a simple large object that is stored in a user-defined location, set **loc_loctype** to LOCUSER and set the **loc_open**, **loc_read**, and **loc_close** fields so that they contain the addresses of appropriate user-defined open, read, and close functions. If the simple large object to be inserted is null, set the **loc_indicator** field to **-1**.

Set the **loc_size** field to the length of the simple-large-object data that you insert. A **loc_size** value of **-1** tells Informix ESQL/C to send the entire user-defined simple-large-object data in a single operation. If the program sets **loc_size** to **-1**, the database server reads in data until the read function returns an end-of-file (EOF) signal. When the count is not equal to the number of bytes requested, the database server assumes an EOF signal.

Creating the User-Defined Simple-Large-Object Functions

Informix ESQL/C provides four transfer functions that you can redefine to handle a user-defined simple-large-object location. The **loc_open**, **loc_read**, **loc_write**, and **loc_close** fields contain pointers to these user-defined location functions. Each of the functions receives the address of the **ifx_loc_t** structure as its first (or only) parameter. You can use the **loc_user_env** field to hold data that a user-defined location function needs. In addition, the **loc_xfercount** and all the fields of the **lc_union** substructure are available for these functions.

The User-Defined Open Function

To define how to prepare the user-defined location for a transfer operation (read or write), you create a C function called a *user-defined open function*. Before you begin a transfer of simple-large-object data to or from the database server, Informix ESQL/C calls the open function supplied in the **loc_open** field of the locator structure.

This user-defined open function must receive the following two arguments:

- The address of the locator structure, **ifx_loc_t** **loc_struc*, where *loc_struc* is the name of a locator structure that your user-defined open function declares
- The open-mode flags, **int** *oflags*, where *oflags* is a variable that contains the open-mode flag

  This flag contains LOC_RONLY if Informix ESQL/C calls the open function to send the simple large object to the database, or LOC_WONLY if Informix ESQL/C calls the function to receive data from the database.

The user-defined open function must return the success code for the open operations as follows:

0   The initialization was successful.

-1   The initialization failed. This return code generates a **loc_status**
(and SQLCODE) error of **-452**.

Figure 7-11 shows a skeleton function of a user-defined open function.

```
open_simple_lo(adloc, oflags)
ifx_loc_t *adloc;
int oflags;
{
   adloc->loc_status = 0;
   adloc->loc_xfercount = 0L;
   if (0 == (oflags & adloc->loc_oflags))
      return(-1);
   if (oflags & LOC_RONLY)
      /*** prepare for store to db ***/
   else
      /*** prepare for fetch to program ***/
   return(0);
}
```

Figure 7-11. A Sample User-Defined Open Function

The User-Defined Read Function

To define how to read the user-defined location, you create a C function called a
user-defined read function. When Informix ESQL/C sends data to the database
server, it reads this data from a character buffer. To transfer the data from a
user-defined location to the buffer, Informix ESQL/C calls the user-defined read
function. Your Informix ESQL/C program must supply the address of your
user-defined read function in the **loc_read** field of the locator structure.

This user-defined read function must receive the following three arguments:
- The address of the locator structure, **ifx_loc_t** *loc_struc*, where *loc_struc* is a
  locator structure that your user-defined read function uses
- The address of the buffer to send data to the database server, **char** *buffer*, where
  buffer is the buffer that your program allocates
- The number of bytes to be read from the user-defined location, **int** *nread*, where
  nread is a variable that contains the number of bytes

This function must transfer the data from the user-defined location to the character
buffer that *buffer* indicates. Informix ESQL/C might call the function more than
once to read a single simple-large-object value from the user-defined location. Each
call receives the address and length of a segment of data. Keep track of the current
seek position of the user-defined location in your user-defined read function. You
might want to use the **loc_position** or **loc_currdata_p** fields for this purpose. You
can also use the **loc_xfercount** field to keep track of the amount of data that was
read.

The user-defined read function must return the success code for the read operation
as follows:

>0   The read operation was successful. The return value indicates the
number of bytes actually read from the locator structure.

-1   The read operation failed. This return code generates a **loc_status**
(and SQLCODE) error of **-454**.

Figure 7-12 shows a skeleton function of a user-defined read function.

```
read_simple_lo(adloc, bufp, ntoread)
ifx_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);
}
```

Figure 7-12. A Sample User-Defined Read Function

The User-Defined Write Function

To define how to write to the user-defined location, you create a C function called a *user-defined write function*. When Informix ESQL/C receives data from the database server, it stores this data in a character buffer. To transfer the data from the buffer to a user-defined location, Informix ESQL/C calls the user-defined write function. Your Informix ESQL/C program must supply the address of your user-defined write function in the **loc_write** field of the locator structure.

This user-defined write function must receive the following three arguments:
- The address of the locator structure, **ifx_loc_t** *loc_struc*, where *loc_struc* is a locator structure that your user-defined write function uses
- The address of the buffer to receive the data from the database server, **char** **buffer*, where *buffer* is the buffer that your program allocates
- The number of bytes to be written to the user-defined location, **int** *nwrite*, where *nwrite* is a variable that contains the number of bytes

The user-defined write function must transfer the data from the character buffer that *buffer* indicates to the user-defined location. Informix ESQL/C might call the function more than once to write a single simple-large-object value to the user-defined location. Each call receives the address and length of a segment of data. Keep track of the current seek position of the user-defined location in your user-defined write function. You might want to use the **loc_position** or **loc_currdata_p** field for this purpose. You can also use the **loc_xfercount** field to keep track of the amount of data that was written.

The user-defined write function must return the success code for the write operation as follows:

>0   The write operation was successful. The return value indicates the number of bytes actually written to the user-defined location

-1   The write operation failed. This return code generates a **loc_status** (and SQLCODE) error of **-455**.

Figure 7-13 shows a skeleton function of a user-defined write function.

```
write_simple_lo(adloc, bufp, ntowrite)
ifx_loc_t *adloc;
char *bufp;
int ntowrite;
{
    int xtoxfer;

   ntoxfer = ntowrite;
   if (adloc->loc_size != -1)
      ntoxfer = min(ntowrite,
         (adloc->loc_size) - (adloc->loc_xfercount));

   /*** transfer "ntowrite" bytes from *bufp ***/

   adloc->loc_xfercount += ntoxfer;
   return(ntoxfer);
}
```

Figure 7-13. A Sample User-Defined Write Function

The User-Defined Close Function

To define how to perform clean-up tasks for the user-defined location, you create a
C function called a *user-defined close function*. When a transfer to or from the
database server is complete, Informix ESQL/C calls the close function that the
loc_close field of the locator structure supplies. Cleanup tasks include closing files
or deallocating memory that the user-defined location uses.

This function must receive one argument: the address of the locator structure,
ifx_loc_t **loc_struc,* where *loc_struc* is a locator structure that your user-defined
close function uses. The user-defined close function must return the success code
for the close operation as follows:

0   The cleanup was successful.

-1   The cleanup failed. This return code generates a **loc_status** (and
                SQLCODE) error of **-453**.

Figure 7-14 shows a skeleton function of a user-defined close function.

```
close_simple_lo (adloc)
ifx_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);
}
```

Figure 7-14. A Sample User-Defined Close Function

Reading and Writing Simple Large Objects to an Optical Disc (UNIX)

In a table, columns of type simple-large-object do not include the simple-large-object data in the table itself. Instead, the simple-large-object column contains a 56-byte *simple-large-object descriptor* that includes a forward pointer (rowid) to the location where the first segment of simple-large-object data is stored. The descriptor can point to a dbspace blobpage, a blobspace blobpage, or a platter in an optical storage subsystem. For details, see your *IBM Informix Administrator's Guide* and the *IBM Informix Optical Subsystem Guide*.

When a simple large object is stored on a write-once-read-many (WORM) optical-storage subsystem, you can have a single physical simple large object reside in more than one table to conserve storage space on the WORM optical disc. The LOC_DESCRIPTOR flag enables you to migrate a simple-large-object descriptor, rather than the simple large object itself, from one table to another.

When you read or write a simple-large-object column that is stored on a WORM optical disc, you can manipulate only the simple-large-object descriptor if you set the **loc_oflags** field of the locator structure to LOC_DESCRIPTOR.

Important: Only use LOC_DESCRIPTOR with simple large objects that are stored on WORM optical media.

Figure 7-15 on page 7-29 shows a code fragment that 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() SQL function expression to retrieve the simple-large-object descriptor, rather than to retrieve the simple large object itself, for the **cat_picture** column. The program then sets the **loc_oflags** field of the **cat_picture** locator structure to LOC_DESCRIPTOR to signal that the simple-large-object descriptor, rather than the simple large object, 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 simple large objects.

```
#include <stdio.h>
EXEC SQL include locator;

char errmsg[400];

EXEC SQL BEGIN DECLARE SECTION;
   mlong cat_num;
   int2 stock_num;
   char manu_code[4];
   ifx_loc_t cat_descr;
   ifx_loc_t cat_picture;
EXEC SQL END DECLARE SECTION;

main(argc, argv)
mint argc;
char *argv[];
{
   EXEC SQL BEGIN DECLARE SECTION;
      char db_name[250];
   EXEC SQL END DECLARE SECTION;

   if (argc > 2)                            /* correct no. of args? */
   {
      printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
         argv[0]);
      exit(1);
   }
   strcpy(db_name, "stores7");
   if(argc == 2)
      strcpy(db_name, argv[1]);
   EXEC SQL connect to :db_name;
   sprintf(db_msg, "CONNECT TO %s",db_name);
   err_chk(db_msg);
```

Figure 7-15. Code Fragment to Retrieve the Simple-Large-Object Descriptor (Part 1 of 3)

```
    EXEC SQL 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) and
 * cat_picture (BYTE that is the simple-large-object descriptor).
 */
cat_descr.loc_loctype = LOCMEMORY;         /* set loctype for in memory */
cat_picture.loc_loctype = LOCMEMORY;       /* set loctype for in memory */
while(1)
       {
       /*
        * Let server get buffers and set loc_buffer (buffer for
        * simple-large-object 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 descriptor
        * for cat_picture.  For cat_descr, the actual simple LO is returned.
        */
       EXEC SQL fetch catcurs into :stock_num, :manu_code, :cat_descr,
          :cat_picture;
       if(err_chk("FETCH") == SQLNOTFOUND)                /* end of data */
          break;
       /*
        * Set LOC_DESCRIPTOR in loc_oflags to indicate simple-large-object
        * descriptor is being inserted rather than simple-large-object data.
        */
       cat_picture.loc_oflags |= LOC_DESCRIPTOR;
       /*
        * Insert
        */
       EXEC SQL insert into pictures values (:stock_num, :manu_code,
          :cat_descr, :cat_picture);
       if(err_chk("INSERT") < 0)
          printf("Insert failed for stock_num %d, manu_code %s", stock_num,
             manu_code);
       }
```

Figure 7-15. Code Fragment to Retrieve the Simple-Large-Object Descriptor (Part 2 of 3)

```
    /* Clean up db resources */
    EXEC SQL close catcurs;
    EXEC SQL free catcurs;

    /* Deallocate memory buffers */
    free(cat_descr.loc_buffer);
    free(cat_picture.loc_buffer);

    EXEC SQL disconnect current;
}


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

int err_chk(stmt)
char *stmt;
{
    char buffer[512];

    if(sqlca.sqlcode < 0)
        {
        fprintf(stderr, "Error: %s\n", stmt);
        rgetlmsg(sqlca.sqlcode, buffer, sizeof(buffer));
        fprintf(stderr, "SQL %d: ", sqlca.sqlcode);
        fprintf(stderr, buffer sqlca.sqlerrm);
        if (sqlca.sqlerrd[1] != 0)
            {
            rgetlmsg(sqlca.sqlerrd[1], buffer, sizeof(buffer));
            fprintf(stderr, "ISAM %d: ", sqlca.sqlerrd[1]);
            fprintf(stderr, buffer, sqlca.sqlerrm);
            }
        exit(1);
        }
    return(sqlca.sqlcode);
}
```

Figure 7-15. Code Fragment to Retrieve the Simple-Large-Object Descriptor (Part 3 of 3)

You can also use the SQL DESCR() function to achieve the same result without a **loc_oflags** value of LOC_DESCRIPTOR. The SQL statement shown in Figure 7-16 accomplishes the same task as the locator structure in the preceding example.

```
EXEC SQL 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;
```

Figure 7-16. Using DESCR() to Access a Simple- Large-Object Descriptor

The dispcat_pic Program

The **dispcat_pic** program, annotated on the following pages, uses the Informix ESQL/C **ifx_loc_t** locator structure to retrieve two simple-large-object columns. The program retrieves the **cat_descr** TEXT simple-large-object column and the **cat_picture** BYTE column from the **catalog** table of the **stores7** demonstration database. For information about how to create the demonstration database, see "Demonstration Databases" on page xiv of the Introduction.

The **dispcat_pic** program allows you to select a database from the command line in case you created the **stores7** database under a different name. If no database name is given, **dispcat_pic** opens the **stores7** database. For example, the following command runs the **dispcat_pic** executable and specifies the **mystores** database:

```
dispcat_pic mystores
```

The program prompts the user for a **catalog_num** value and performs a SELECT statement 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 **.gif** file.

If the SELECT statement succeeds, the program displays the **catalog_num**, **cat_descr**, and **description** columns. Since these columns store text, they can be displayed on any Informix ESQL/C platform. The program also allows the user to enter another **catalog_num** value or terminate the program.

To prepare to run the dispcat_pic program:
1. Load the simple-large-object images into the **catalog** table with the **blobload** utility.
2. Compile the **dispcat_pic.ec** file into an executable program.

Loading the Simple-Large-Object Images

When the **catalog** table is created as part of the **stores7** demonstration database, the **cat_picture** column for all rows is set to null. The Informix ESQL/C demonstration directory provides five graphic images. Use the **blobload** utility to load simple-large-object images into the **cat_picture** column of the **catalog** table.

To display these simple-large-object images from the **dispcat_pic** program, you must load the images to the **catalog** table.

Choosing the Image Files

The five **cat_picture** images are provided in the Graphics Interchange Format files, which have the **.gif** file extension.

Informix ESQL/C provides the images in **.gif** files to provide them in a standard format that can be displayed on all platforms or translated into other formats with filter programs that other vendors supply. The right column of Table 7-6 shows the names of the **.gif** files for the simple-large-object images.

Table 7-6. Image Files for Simple-Large-Object Demo

Image	Graphics Interchange Format (.gif Files)
Baseball glove	cn_10001.gif
Bicycle crankset	cn_10027.gif
Bicycle helmet	cn_10031.gif
Golf balls	cn_10046.gif
Running shoe	cn_10049.gif

The numeric portion of the image file name is the **catalog_num** value for the row of the **catalog** table to which the image is to be updated. For example, **cn_10027.gif** should be updated to the **cat_picture** column of the row where 10027 is the value of **catalog_num**.

Using the blobload Utility

The **blobload** utility is an Informix ESQL/C program that is provided as part of the Informix ESQL/C demonstration files. It uses a command-line syntax to load a byte image into a specified table and column of a database.

To load the simple-large-object images with blobload:

1. Compile the **blobload.ec** program with the following command:

   ```
   esql -o blobload blobload.ec
   ```

2. Enter **blobload** on the UNIX command line without any arguments, as follows:

   ```
   blobload
   ```

   Figure 7-17 shows the output of this command 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.
```

Figure 7-17. Sample Output from the blobload Utility

3. Run the **blobload** program to load each image to its proper **cat_picture** column.

   The **-u** option of **blobload** updates a specified column with a simple-large-object image. To identify which column to update, you must also use the **-f**, **-d**, **-t**, **-b**, and **-k** options of **blobload**.

You must run the **blobload** program once for each image file that you want to update. For example, the following command loads the contents of the **cn_10027.gif** file 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.gif -d stores7 -t catalog -b cat_picture -k
   catalog_num 10027
```

Use the same command to update each of the four remaining image files, substituting the file name (**-f** option) and corresponding **catalog_num** value (**-k** option) of the image file that you want to load.

Guide to the dispcat_pic.ec File

```
======================================================================
 1. /*
 2.    * dispcat_pic.ec *
 3. The following program prompts the user for a catalog number,
 4. selects the cat_picture column, if it is not null, from the
 5. catalog table of the demonstration database and saves the
 6. image into a .gif file.
```

```
 7.  */
 8.  #include <stdio.h>
 9.  #include <ctype.h>
10.  EXEC SQL include sqltypes;
11.  EXEC SQL include locator;
12.  #define WARNNOTIFY      1
13.  #define NOWARNNOTIFY    0
14.  #define LCASE(c) (isupper(c) ? tolower(c) : (c))
15.  #define BUFFSZ 256
16.  extern errno;
17.  EXEC SQL BEGIN DECLARE SECTION;
18.      mlong cat_num;
19.      ifx_loc_t cat_descr;
20.      ifx_loc_t cat_picture;
21.  EXEC SQL END DECLARE SECTION;
22.  char cpfl[18];  /* file to which the .gif will be copied */
```

==

Lines 8 to 11

The **#include <stdio.h>** statement includes the **stdio.h** header file from the
/usr/include directory on UNIX and from the **include** subdirectory for Microsoft
Visual C++ on Windows. The **stdio.h** file enables **dispcat_pic** to use the standard C
I/O library. The program also includes the Informix ESQL/C header files
sqltypes.h and **locator.h** (lines 10 and 11). The **locator.h** file contains the definition
of the locator structure and the constants that you need to work with this
structure.

Lines 12 to 16

Use the WARNNOTIFY and NOWARNNOTIFY constants (lines 12 and 13) with
the **exp_chk2()** exception-handling function. Calls to **exp_chk2()** specify one of
these constants as the second argument to indicate whether or not to display
SQLSTATE and SQLCODE information for warnings (WARNNOTIFY or
NOWARNNOTIFY). See lines 171 to 177 for more information about the
exp_chk2() function.

The program uses BUFFSZ (line 15) to specify the size of arrays that store input
from the user. Line 16 defines **errno**, an external integer where system calls store
an error number.

Lines 17 to 21

These lines define *global host variables* needed for the program. The **cat_num**
variable holds the **catalog_num** column value of the **catalog** table. Lines 19 and 20
specify the locator structure as the data type for host variables that receive data for
the **cat_descr** and **cat_picture** simple-large-object columns of the **catalog** table. The
locator structure is the host variable for a simple-large-object column that is
retrieved from or stored to the database. The locator structure has a **ifx_loc_t**
typedef. The program uses the locator structure to specify simple-large-object size
and location.

Line 22

Line 22 defines a single *global C variable*. The **cpfl** character array stores the name
of a file. This named file is the location for the simple-large-object **.gif** image of
cat_picture that the database server writes.

==
```
23.  main(argc, argv)
24.  mint argc;
```

```
25.   char *argv[];
26.   {
27.         char ans[BUFFSZ];
28.         int4 ret, exp_chk2();
29.         char db_msg[ BUFFSZ + 1 ];
30.      EXEC SQL BEGIN DECLARE SECTION;
31.           char db_name[20];
32.           char description[16];
33.      EXEC SQL END DECLARE SECTION;
```

==

Lines 23 to 26

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, to specify the name of the database
to access. If **argv[1]** is not present, the program opens the **stores7** database.

Lines 27 to 29

Lines 27 to 29 define the C variables that are *local* in scope to the **main()** function.
The **ans[BUFFSZ]** array is the buffer that receives input from the user, namely the
catalog number for the associated **cat_picture** column. Line 28 defines a 4-byte
integer (**ret**) for the value that **exp_chk2()** returns and declares **exp_chk2()** as a
function that returns a **long**. The **db_msg[BUFFSZ + 1]** character array holds the
form of the CONNECT statement used to open the database. If an error occurs
while the CONNECT executes, the string in **db_msg** is passed into the **exp_chk2()**
function to identify the cause of the error.

Lines 30 to 33

Lines 30 to 33 define the Informix ESQL/C host variables that are *local* to the
main() function. A host variable receives data that is fetched from a table and
supplies data that is written to a table. The **db_name[20]** character array is a host
variable that stores the database name if the user specifies one on the command
line. The **description** variable holds the value that the user entered, which is to be
stored in the column of the **stock** table.

==

```
34.         printf("DISPCAT_PIC Sample ESQL Program running.\n\n");
35.      if (argc > 2)            /* correct no. of args? */
36.          {
37.          printf("\nUsage: %s [database]\nIncorrect no. of
    argument(s)\n",
38.                argv[0]);
39.          printf("DISPCAT_PIC Sample Program over.\n\n");
40.          exit(1);
41.          }
42.      strcpy(db_name, "stores7");
43.      if(argc == 2)
44.          strcpy(db_name, argv[1]);
45.      EXEC SQL connect to :db_name;
46.      sprintf(db_msg,"CONNECT TO %s",db_name);
47.      if(exp_chk2(db_msg, NOWARNNOTIFY) < 0)
48.          {
49.          printf("DISPCAT_PIC Sample Program over.\n\n");
50.          exit(1);
51.          }
```

```
52.      if(sqlca.sqlwarn.sqlwarn3 != 'W')
53.          {
54.          printf("\nThis program does not work with Informix SE. ");
55.          EXEC SQL disconnect current;
56.          printf("\nDISPCAT_PIC Sample Program over.\n\n");
57.          exit(1);
58.          }
59.      printf("Connected to %s\n", db_name);
60.      ++argv;
```

==

Lines 34 to 51

These lines interpret the command-line arguments and open the database. Line 35
checks 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 42 assigns the default database name of **stores7** to the **db_name**
host variable. The program opens this database if the user does not enter a
command-line argument.

The program then tests 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 44 uses the **strcpy()** function to copy
the name of the database from the **argv[1]** command line into the **db_name** host
variable. The program then executes the CONNECT statement (line 45) to establish
a connection to the default database server and open the specified database (in
db_name).

The program reproduces the CONNECT statement in the **db_msg[]** array (line 46).
It does so for the sake of the **exp_chk2()** call on line 47, which takes as its
argument the name of a statement. Line 47 calls the **exp_chk2()** function to check
on the outcome. This call to **exp_chk2()** specifies the NOWARNNOTIFY argument
to prevent the display of warnings that CONNECT generates.

Lines 52 to 60

After CONNECT successfully opens the database, it stores information about the
database server in the **sqlca.sqlwarn** array. Because the **dispcat_pic** program
handles simple-large-object data types that are not supported on IBM Informix SE,
line 52 checks the type of database server. If the **sqlwarn3** element of
sqlca.sqlwarn is set to W, the database server is any server other than IBM Informix
SE. Otherwise, the program notifies the user that it cannot continue and exits. The
program has established the validity of the database server and now displays the
name of the database that is opened (line 59).

==
```
61.   while(1)
62.       {
63.       printf("\nEnter catalog number: "); /* prompt for cat.
                                              * number */
64.       if(!getans(ans, 6))
65.           continue;
66.       printf("\n");
67.       if(rstol(ans, &cat_num))     /* cat_num string to long */
68.           {
69.           printf("** Cannot convert catalog number '%s' to long
      integer\n",
                   ans);
70.           EXEC SQL disconnect current;
```

```
71.            printf("\nDISPCAT_PIC Sample Program over.\n\n");
72.            exit(1);
73.            }
74.        ret=sprintf(cpfl, "pic_%s.gif", ans);
75.        /*
76.         *  Prepare locator structure for select of cat_descr
77.         */
78.        cat_descr.loc_loctype = LOCMEMORY;  /* set for 'in memory' */
79.        cat_descr.loc_bufsize = -1;         /* let db get buffer */
80.        cat_descr.loc_mflags = 0;   /* clear memory-deallocation
                                         * feature */
81.        cat_descr.loc_oflags = 0;           /* clear loc_oflags */
82.        /*
83.         *  Prepare locator structure for select of cat_picture
84.         */
85.        cat_picture.loc_loctype = LOCFNAME;   /* type = named file */
86.        cat_picture.loc_fname = cpfl;       /* supply file name */
87.        cat_picture.loc_oflags = LOC_WONLY; /* file-open mode = write
                                                */
88.        cat_picture.loc_size = -1;  /* size = size of file */
```

==

Lines 61 to 74

The **while(1)** on line 61 begins the main processing loop in **dispcat_pic**. Line 63 prompts the user to enter a catalog number for the **cat_picture** column that the user wants to see. Line 64 calls **getans()** to receive the catalog number that the user inputs. The arguments for **getans()** are the address in which the input is stored, **ans[]**, and the maximum length of the input that is expected, including the null terminator. If the input is unacceptable, **getans()** returns 0 and line 65 returns control to the **while** at the top of the loop in line 61, which causes the prompt for the catalog number to be redisplayed. For a more detailed explanation of **getans()**, see "Guide to the inpfuncs.c File" on page 7-42.Line 67 calls the Informix ESQL/C library function **rstol()** to convert the character 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 fails and lines 69 to 72 display a message to the user, close the connection, and exit. Line 74 creates the name of the **.gif** file to which the program writes the simple-large-object image. The file name consists of the constant **pic_**, the catalog number that the user entered, and the extension **.gif**. The file is created in the directory from which the program is run.

Lines 75 to 81

These lines define the simple-large-object location for the TEXT **cat_descr** column of the **catalog** table, as follows:

- Line 78 sets **loc_loctype** in the **cat_descr** locator structure to LOCMEMORY to tell Informix ESQL/C to select the data for **cat_descr** into memory.

- Line 79 sets **loc_bufsize** to **-1** so that Informix ESQL/C allocates a memory buffer to receive the data for **cat_descr**.

- Line 80 sets **loc_mflags** to 0 to disable the memory-deallocation feature (see Line 149) of Informix ESQL/C.

If the select is successful, Informix ESQL/C returns the address of the allocated buffer in **loc_buffer**. Line 81 sets the **loc_oflags** file-open mode flags to 0 because the program retrieves the simple-large-object information into memory rather than a file.

Lines 82 to 88

These lines prepare the locator structure to retrieve the BYTE column **cat_picture** of the **catalog** table. Line 85 moves LOCFNAME to **loc_loctype** to tell Informix ESQL/C to locate the data for **cat_descr** in a named file. Line 86 moves the address of the **cpfl** file name into **loc_fname**. Line 87 moves the LOC_WONLY value into the **loc_oflags** file-open mode flags to tell Informix ESQL/C to open the file in write-only mode. Finally, line 88 sets **loc_size** to **-1** to tell Informix ESQL/C to send the BYTE data in a single transfer rather than break the value into smaller pieces and use multiple transfers.

```
======================================================================
89.      /* Look up catalog number */
90.      EXEC SQL select description, catalog_num, cat_descr, cat_picture
91.          into :description, :cat_num, :cat_descr, :cat_picture
92.          from stock, catalog
93.          where catalog_num = :cat_num and
94.          catalog.stock_num = stock.stock_num and
95.          catalog.manu_code = stock.manu_code;
96.      if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100) /* if not
                                                    * found */
97.          {
98.          printf("** Catalog number %ld not found in ", cat_num);
99.          printf("catalog table.\n");
100.          printf("\t OR item not found in stock table.\n");
101.          if(!more_to_do())
102.              break;
103.          continue;
104.          }
105.      if (ret < 0)
106.          {
107.          EXEC SQL disconnect current;
108.          printf("\nDISPCAT_PIC Sample Program over.\n\n");
109.          exit(1);
110.          }
111.      if(cat_picture.loc_indicator == -1)
112.          printf("\tNo picture available for catalog number %ld\n\n",
113.              cat_num);
114.      else
115.          {
116.          printf("Stock Item for %ld: %s\n", cat_num, description);
117.          printf("\nThe cat_picture column has been written to the
    file:
118.              %s\n",    cpfl);
119.          printf("Use an image display tool or a Web browser ");
120.          printf("to open %s for viewing.\n\n", cpfl);
121.          }
122.      prdesc();       /* display catalog.cat_descr */

======================================================================
```

Lines 89 to 95

These lines define a SELECT statement 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 that the user entered. The INTO clause of the SELECT statement identifies the host variables that contain the selected values. The two **ifx_loc_t** host variables, **cat_descr** and **cat_picture**, are listed in this clause for the TEXT and BYTE values.

Lines 96 to 104

The **exp_chk2()** function checks whether the SELECT statement was able to find the **stock_num** and **manu_code** for the selected row in the **catalog** table and in the **stock** table. The **catalog** table should not contain a row that does not have a corresponding row in the **stock** table. Lines 98 to 103 handle a NOT FOUND condition. If the **exp_chk2()** function returns 100, the row was not found; lines 98 to 100 display a message to that effect. The **more_to_do()** function (line 101) asks whether the user wants to continue. If the user answers n for no, a **break** terminates the main processing loop and control transfers to line 131 to close the database before the program terminates.

Lines 105 to 110

If a runtime error occurs during the select, the program closes the current connection, notifies the user, and exits with a status of 1.

Lines 111 to 113

If **cat_picture.loc_indicator** contains **-1** (line 111), the **cat_picture** column contains a null and the program informs the user (line 112). Execution then continues to line 113 to display the other returned column values.

Lines 114 to 122

These lines display the other columns that the SELECT statement returned. Line 116 displays the catalog number that is being processed and the **description** column from the **stock** table. Line 122 calls **prdesc()** to display the **cat_descr** column. For a detailed description of **prdesc()**, see "Guide to the prdesc.c File" on page 7-41.

```
========================================================================
123.    if(!more_to_do())    /* More to do? */
124.      break;        /* no, terminate loop */
125.       /* If user chooses to display more catalog rows, enable the
126.        * memory-deallocation feature so that ESQL/C deallocates old
127.        * cat_desc buffer before it allocates a new one.
128.        */
129.      cat_descr.loc_mflags = 0;   /* clear memory-deallocation feature
                                       */
130.      }
131.      EXEC SQL disconnect current;
132.      printf("\nDISPCAT_PIC Sample Program over.\n\n");
133.    } /* end main */
134.    /* prdesc() prints cat_desc for a row in the catalog table */
135.    #include "prdesc.c"

========================================================================
```

Lines 123 to 130

The **more_to_do()** function then asks whether the user wants to enter more catalog numbers. If not, **more_to_do()** 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 130 terminates the main processing loop, which began with the **while(1)** on line 61. If the user wants to enter another catalog number, control returns to line 61.

Line 131 to 133

When a **break** statement (line 124) terminates the main processing loop that the **while(1)** on line 61 began, control transfers to line 131, which closes the database

and the connection to the default database server. The closing brace on line 133 terminates the **main()** function on line 23 and terminates the program.

Lines 134 and 135

Several of the Informix ESQL/C simple-large-object demonstration programs call the **prdesc()** function. To avoid having the function in each program, the function is put in its own source file. Each program that calls **prdesc()** includes the **prdesc.c** source file. Since **prdesc()** does not contain any Informix ESQL/C statements, the program can include it with the C **#include** preprocessor statement (instead of the Informix ESQL/C **include** directive). For a description of this function, see "Guide to the prdesc.c File" on page 7-41.

```
=======================================================================
136. /*
137.  * The inpfuncs.c file contains the following functions used in this
138.  * program:
139.  *    more_to_do() - asks the user to enter 'y' or 'n' to indicate
140.  *                   whether to run the main program loop again.
141.  *
142.  *    getans(ans, len) - accepts user input, up to 'len' number of
143.  *                   characters and puts it in 'ans'
144.  */
145. #include "inpfuncs.c"
146. /*
147.  * The exp_chk.ec file contains the exception handling functions to
148.  * check the SQLSTATE status variable to see if an error has
         occurred
149.  * following an SQL statement. If a warning or an error has
150.  * occurred, exp_chk2() executes the GET DIAGNOSTICS statement and
151.  * displays the detail for each exception that is returned.
152. */
153. EXEC SQL include exp_chk.ec;
=======================================================================
```

Lines 136 and 145

Several of the Informix ESQL/C demonstration programs also call the **more_to_do()** and **getans()** functions. These functions are also broken out into a separate C source file and included in the appropriate demonstration program. Neither of these functions contain Informix ESQL/C, so the program can use the C **#include** preprocessor statement to include the files. For a description of these functions, see "Guide to the inpfuncs.c File" on page 7-42.

Line 146 to 153

The **exp_chk2()** function examines the SQLSTATE status variable to determine the outcome of an SQL statement. Because many demonstration programs use exception checking, the **exp_chk2()** function and its supporting functions are broken out into a separate **exp_chk.ec** source file. The **dispcat_pic** program must use the Informix ESQL/C **include** directive to include this file because the exception-handling functions use Informix ESQL/C statements. For a description of the **exp_chk.ec** source file, see "Guide to the exp_chk.ec File" on page 11-32.

Tip: In a production environment, functions such as **prdesc()**, **more_to_do()**, **getans()**, and **exp_chk2()** would be put into C libraries and included on the command line of the *Informix ESQL/C* program at compile time.

Guide to the prdesc.c File

The **prdesc.c** file contains the **prdesc()** function. This function sets the pointer **p** to the address that is provided in the **loc_buffer** field of the locator structure to access the simple large object. The function then reads the text from the buffer 80 bytes at a time up to the size specified in **loc_size**. This function is used in several of the simple-large-object demonstration programs so it is in a separate file and included in the appropriate source files.

```
========================================================================
 1. /* prdesc() prints cat_desc for a row in the catalog table */
 2. prdesc()
 3. {
 4.      int4 size;
 5.      char shdesc[81], *p;
 6.      size = cat_descr.loc_size;    /* get size of data */
 7.      printf("Description for %ld:\n", cat_num);
 8.      p = cat_descr.loc_buffer;    /* set p to buffer addr */
 9.    /* print buffer 80 characters at a time */
10.      while(size >= 80)
11.         {
12.         ldchar(p, 80, shdesc);      /* mv from buffer to shdesc */
13.         printf("\n%80s", shdesc);   /* display it  */
14.         size -= 80;                 /* decrement length */
15.         p += 80;                    /* bump p by 80 */
16.         }
17.       strncpy(shdesc, p, size);
18.      shdesc[size] = '\0';
19.      printf("%-s\n", shdesc);       /* display last segment */
20. }
========================================================================
```

Lines 1 to 20

Lines 2 to 20 make up the **prdesc()** function, which displays the **cat_descr** column of the **catalog** table. Line 4 defines **size**, a long integer that **prdesc()** initializes with the value in **cat_descr.loc_size.** Line 5 defines **shdesc[81]**, an array into which **prdesc()** temporarily moves 80-byte chunks of the **cat_descr** text for output. Line 5 also defines ***p**, a pointer that marks the current position in the buffer as it is being displayed.

In **loc_size**, the database server returns the size of the buffer that it allocates for a simple large object. Line 6 moves **cat_descr.loc_size** to **size.** Line 7 displays the string "Description for:" as a header for the **cat_descr** text. Line 8 sets the **p** pointer to the buffer address that the database server returned in **cat_descr.loc_size.**

Line 10 begins the loop that displays the **cat_descr** text to the user. The **while()** repeats the loop until **size** is less than 80. Line 11 begins the body of the loop. The Informix ESQL/C **ldchar()** library function copies 80 bytes from the current position in the buffer, which **p** addresses, to **shdesc[]** and removes any trailing blanks. Line 13 prints the contents of **shdesc[]**. Line 14 subtracts 80 from **size** to account for the portion of the buffer that was just printed. Line 15, 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 fewer than 80 characters are left to be displayed (**size** < 80). Line 17 copies the

remainder of the buffer into **shdesc[]** for the length of **size**. Line 18 appends a null to **shdesc[size]** to mark the end of the array and line 19 displays **shdesc[]**.

Guide to the inpfuncs.c File

The **inpfuncs.c** file contains the following two functions:

- The **getans()** function
- The **more_to_do()** function

Because these functions are used in several Informix ESQL/C demonstration programs, they are in a separate file and included in the appropriate demonstration source files.

```
========================================================================
1.  /* The inpfuncs.c file contains functions useful in character-based
2.     input for a C program.
3.  */
4.  #include <ctype.h>
5.  #ifndef LCASE
6.  #define LCASE(c) (isupper(c) ? tolower(c) : (c))
7.  #endif
8.  /*
9.     Accepts user input, up to 'len' number of characters and returns
10.    it in 'ans'
11. */
12. #define BUFSIZE 512
13. getans(ans, len)
14. char *ans;
15. mint len;
16. {
17.     char buf[BUFSIZE + 1];
18.     mint c, n = 0;
19.     while((c = getchar()) != ';' && n < BUFSIZE)
20.         buf[n++] = c;
21.     buf[n] = '\0';
22.     if(n > 1 && n >= len)
23.         {
24.         printf("Input exceeds maximum length");
25.         return 0;
26.         }
27.     if(len <= 1)
28.         *ans = buf[0];
29.     else
30.         strncpy(ans, buf, len);
31.     return 1;
32. }
========================================================================
```

Lines 1 to 7

Line 4 includes the UNIX **ctype.h** header file. This header file provides the definitions of the **islower()** and **tolower()** macros used in the definition of the LCASE() macro (defined on line 6). The program only defines the LCASE macro if it has not yet been defined in the program.

Lines 8 to 32

The BUFSIZE constant (line 12) defines the size of the character buffer used in the **getans()** function. Lines 13 to 32 constitute the **getans()** function. The **getans()**

function uses the **getchar()** standard library function to accept input from the user. Lines 14 and 15 define the arguments for **getans()**, the address of the buffer (**ans**) where it copies the input, and the maximum number of characters (**len**) that the calling function expects. Line 17 defines **buf[]**, an input buffer array. The **int** variable **c** (line 18) receives the character that **getchar()** returned. The second integer defined on line 18, **n**, is used to subscript the **buf[]** input buffer.

Line 19 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 20 moves the input character **c** into the current position in **buf[]**. Line 21 places a null terminator at the end of the input, **buf[n]**.

Lines 22 to 26 check whether the number of characters received, **n**, is less than the number of characters expected, **len**. If not, line 24 displays a message to the user and line 25 returns 0 to the calling function to indicate that an error occurred. Line 27 checks whether one or more characters were entered. If the expected number of characters, **len**, is less than or equal to 1, line 28 moves only a single character to the address that the **ans** calling function gives. If only one character is expected, **getans()** does not append a null terminator to the input. If the length of the input is greater than 1, line 30 copies the user's input to the address that the calling function (**ans**) supplies. Line 31 returns 1 to the calling function to indicate successful completion.

```
=========================================================================
33. /*
34.  *  Ask user if there is more to do
35.  */
36. more_to_do()
37. {
38.    char ans;
39.        do
40.           {
41.           printf("\n**** More? (y/n) ...");
42.           getans(&ans, 1);
43.           } while((ans = LCASE(ans)) != 'y' && ans != 'n');
44.        return (ans == 'n') ? 0 : 1;
45. }

=========================================================================
```

Lines 33 to 45

The **more_to_do ()** function displays "More? (y/n)..." to ask whether the user wants to continue program execution. The **more_to_do()** function does not have any input arguments. Line 38 defines a one-character field, **ans**, to receive the user's response. The condition expressed on line 43 causes the question to be redisplayed until the user answers y(yes) or n(no). The LCASE macro converts the user's answer to lowercase letters for the comparison. Line 42 calls **getans()** to accept the user's input. Once the user answers yes or no, control passes to line 44, which returns 1 for yes and 0 for no to the calling function.

Chapter 8. Working with smart large objects

The information in these topics apply only if you are using IBM Informix as your database server.

A smart large object is a data type that stores large, non-relational data objects such as images, sound clips, documents, graphics, maps and other large objects, and allows you to perform read, write, and seek operations on those objects. Smart large objects consist of the CLOB (character large object) and BLOB (binary large object) data types. The CLOB data type stores large objects of text data. The BLOB data type stores large objects of binary data in an undifferentiated byte stream. A smart large object is stored in a particular type of database space called an *sbspace.* For information about creating and administering sbspaces, see your *IBM Informix Administrator's Guide.*

These topics describe the following information about programming with smart large objects:

- Data structures for smart large objects
- Creating a smart large object

- Accessing a smart large object
- Obtaining the status of a smart large object
- Altering a smart-large-object column
- Reading and writing smart large objects on an optical disc
- The Informix ESQL/C API for smart large objects

The end of this chapter presents an example program called **create_clob**. The **create_clob** sample program demonstrates how to create a new smart large object from an Informix ESQL/C program, insert data into a CLOB column of the **stores7** database, and then select the smart-large-object data back from this column.

For more information about the CLOB and BLOB data types, as well as other SQL data types, see the *IBM Informix Guide to SQL: Reference*.

Data structures for Smart Large Objects

Informix ESQL/C supports the SQL data types CLOB and BLOB with the **ifx_lo_t** data type. Because of the potentially huge size of smart-large-object data, the Informix ESQL/C program does not store the data directly in a host variable. Instead, the client application accesses the data as a file-like structure. To use smart-large-object variables in an Informix ESQL/C program, take the following actions:

- Declare a host variable with the **ifx_lo_t** data type.

  For more information, see "Declaring a Host Variable" on page 8-2.
- Access the smart large object with a combination of the following three data structures:
  - The LO-specification structure, **ifx_lo_create_spec_t**

    For more information, see "The LO-specification structure" on page 8-3 and "Obtain storage characteristics" on page 8-6.
  - The LO-pointer structure, **ifx_lo_t**

    For more information, see "Deallocating the LO-Specification Structure" on page 8-9.
  - An integer LO file descriptor

    For more information, see "Opening a Smart Large Object" on page 8-13.

Important: The structures that ESQL/C uses to access smart large objects begin with the LO prefix. This prefix is an acronym for large object. Currently, the database server uses **large object** to refer to both smart large objects and simple large objects. However, use of this prefix in the ESQL/C structures that access smart large objects is retained for legacy purposes.

Declaring a Host Variable

Declare Informix ESQL/C host variables for database columns of type CLOB or BLOB as a **fixed binary** host variable with the **ifx_lo_t** structure (called an **ifx_lo_t** data type) as follows:

```
EXEC SQL include locator;
   .
   .
   .
EXEC SQL BEGIN DECLARE SECTION;
   fixed binary 'clob' ifx_lo_t clob_loptr;
   fixed binary 'blob' ifx_lo_t blob_loptr;
EXEC SQL END DECLARE SECTION;
```

.
.
.

```
EXEC SQL select blobcol into :blob_loptr from tab1;
```

The **ifx_lo_t** data type stores an *LO-pointer structure*. For more information, see "The ifx_lo_t Structure" on page 8-10.

Tip:: For more information about the **fixed binary** Informix ESQL/C data type, see "Accessing a Fixed-Length Opaque Type" on page 10-12.

To access smart large objects, you must include the **locator.h** header file in your Informix ESQL/C program. This header file contains definitions of data structures and constants that your program needs to work with smart large objects.

The LO-specification structure

Before you create a new smart large object, you must allocate an LO-specification structure with the **ifx_lo_def_create_spec()** function. The **ifx_lo_def_create_spec()** function performs the following tasks:

1. It allocates a new LO-specification structure, whose pointer you provide as an argument.
2. It initializes all fields of the LO-specification structure: disk-storage information and create-time flags to the appropriate null values.

For more information about the **ifx_lo_def_create_spec()** function, see B-112.

The ifx_lo_create_spec_t structure

The LO-specification structure, **ifx_lo_create_spec_t**, stores the storage characteristics for a smart large object in an Informix ESQL/C program. The **locator.h** header file defines the LO-specification structure, so you must include the **locator.h** file in your Informix ESQL/C programs that access this structure.

Important: The LO-specification structure, **ifx_lo_create_spec_t**, is an opaque structure to Informix ESQL/C programs. Do not access its internal structure directly. The internal structure of **ifx_lo_create_spec_t** may change in future releases. Therefore, to create portable code, always use the Informix ESQL/C access functions for this structure to obtain and store values in the LO-specification structure.

For a list of these access functions, see Table 8-1 on page 8-4 and Table 8-2 on page 8-5.

The LO-specification structure stores the following storage characteristics for a smart large object:
- Disk-storage information
- Create-time flags

Disk-storage information: The LO-specification structure stores disk-storage information, which helps the database sever determine how to store the smart large object most efficiently on disk. The following table shows the disk-storage information along with the corresponding Informix ESQL/C access functions.

Table 8-1. Disk-storage information in the LO-specification structure

Disk-storage information	Description	ESQL/C accessor functions
Estimated number of bytes	An estimate of the final size, in bytes, of the smart large object. The database Server uses this value to determine the extents in which to store the smart large object. This value provides optimization information. If the value is grossly incorrect, it does not cause incorrect behavior. However, it does mean that the database server might not necessarily choose optimal extent sizes for the smart large object.	**ifx_lo_specget_estbytes()** **ifx_lo_specset_estbytes()**
Maximum number of bytes	The maximum size, in bytes, for the smart large object. The database server does not allow the smart large object to grow beyond this size.	**ifx_lo_specget_maxbytes()** **ifx_lo_specset_maxbytes()**
Allocation extent size	The allocation extent size is specified in kilobytes. Optimally, the allocation extent is the single extent in a chunk that holds all the data for the smart large object. The database server performs storage allocations for smart large objects in increments of the allocation extent size. It tries to allocate an allocation extent as a single extent in a chunk. However, if no single extent is large enough, the database server must use multiple extents as necessary to satisfy the request.	**ifx_lo_sepcget_extsz(),** **ifx_lo_specset_extsz()**
Name of the sbspace	The name of the sbspace that contains the smart large object. The sbspace name can be at most 18 characters long. This name must be null terminated.	**ifx_lo_specget_sbspace(),** **ifx_lo_specset_sbspace()**

For most applications, it is recommended that you use the values for the disk-storage information that the database server determines. For more information about each of the accessor functions, see its description in Appendix B, "ESQL/C function library," on page B-1.

Create-time flags: The LO-specification structure stores create-time flags, which tell the database server what options to assign to the smart large object. The following table shows the create-time flags along with the corresponding Informix ESQL/C access functions.

Table 8-2. Create-time flags in the LO-specification structure

Type of indicator	Create-time flag	Description
Logging	LO_LOG	Tells the database server to log changes to the smart large object in the system log file.
		Consider carefully whether to use the LO_LOG flag value. The database server incurs considerable overhead to log smart large objects. You must also ensure that the system log file is large enough to hold the value of the smart large object. For more information, see your *IBM Informix Administrator's Guide*.
	LO_NOLOG	Tells the database server to turn off logging for all operations that involve the associated smart large object.
Last access-time	LO_KEEP_LASTACCESS_TIME	Tells the database server to save the last access time for the smart large object. This access time is the time of the last read or write operation.
		Consider carefully whether to use the LO_KEEP_LASTACCESS_TIME flag value. The database server incurs considerable overhead to maintain last access times for smart large objects.
	LO_NOKEEP_LASTACCESS_TIME	Tells the database server not to maintain the last access time for the smart large object.

The **locator.h** header file defines the LO_LOG, LO_NOLOG, LO_KEEP_LASTACCESS_TIME, and LO_NOKEEP_LASTACCESS_TIME create-time constants. The two groups of create-time flags, logging indicators and the last access-time indicators, are stored in the LO-specification structure as a single flag value. To set a flag from each group, use the C-language OR operator to mask the two flag values together. However, masking mutually exclusive flags results in an error.

The **ifx_lo_specset_flags()** function sets the create-time flags to a new value. The **ifx_lo_specget_flags()** function retrieves the current value of the create-time flag.

If you do not specify a value for one of the flag groups, the database server uses the inheritance hierarchy to determine this information. For more information about the inheritance hierarchy, see "Obtain storage characteristics" on page 8-6.

For more information about the create-time flags, see the descriptions of the **ifx_lo_specset_flags()** and **ifx_lo_specget_flags()** functions in Appendix B, "ESQL/C function library," on page B-1.

ESQL/C functions that use the LO-specification structure

The following table shows the Informix ESQL/C library functions that access the LO-specification structure.

ESQL/C Library Function	Purpose	See
ifx_lo_col_info()	Updates the LO-specification structure with the column-level storage characteristics	B-107
ifx_lo_create()	Reads an LO-specification structure to obtain storage characteristics for a new smart large object that it creates	B-111

ESQL/C Library Function	Purpose	See
ifx_lo_def_create_spec()	Allocates and initializes an LO-specification structure	B-112
ifx_lo_spec_free()	Frees the resources of the LO-specification structure	B-122
ifx_lo_specget_estbytes()	Gets the estimated number of bytes from the LO-specification structure	B-123
ifx_lo_specget_extsz()	Gets the allocation extent size from the LO-specification structure	B-124
ifx_lo_specget_flags()	Gets the create-time flags from the LO-specification structure	B-125
ifx_lo_specget_maxbytes()	Gets the maximum number of bytes from the LO-specification structure	B-126
ifx_lo_specget_sbspace()	Gets the name of the sbspace from the LO-specification structure	B-127
ifx_lo_specset_estbytes()	Sets the estimated number of bytes from the LO-specification structure	B-129
ifx_lo_specset_extsz()	Sets the allocation extent size in the LO-specification structure	B-130
ifx_lo_specset_flags()	Sets the create-time flags in the LO-specification structure	B-131
ifx_lo_specset_maxbytes()	Sets the maximum number of bytes in the LO-specification structure	B-132
ifx_lo_specset_sbspace()	Sets the name of the sbspace in the LO-specification structure	B-132
ifx_lo_stat_cspec()	Returns the storage characteristics into the LO-specification structure for a specified smart large object	B-135

Obtain storage characteristics

Once you have allocated an LO-specification structure with the
ifx_lo_def_create_spec() function, you must ensure that this structure contains the
appropriate storage characteristics when you create a smart large object.

IBM Informix uses an inheritance hierarchy to obtain storage characteristics. The
following figure shows the inheritance hierarchy for smart-large-object storage
characteristics.

```
┌─────────────────────────────────────────────┐
│   Database server storage characteristics    │
│   (system defaults and the ONCONFIG file)    │
└─────────────────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────┐
│        sbspace storage characteristics       │
│     (assigned when the sbspace is created)   │
└─────────────────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────┐
│       Column-level storage characteristics   │
│   (assigned with the CREATE TABLE statement) │
└─────────────────────────────────────────────┘
                      │
                      ▼
┌─────────────────────────────────────────────┐
│       User-defined storage characteristics   │
│     (assigned from within an ESQL/C program) │
└─────────────────────────────────────────────┘
```
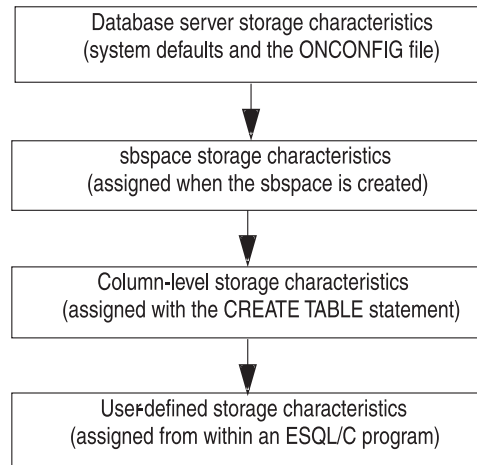
Figure 8-1. Inheritance hierarchy for storage characteristics

The system-specified storage characteristics: IBM Informix uses one of the
following sets of storage characteristics as the system-specified storage
characteristics:

* If the sbspace in which the smart large object is stored has specified a value for
  a particular storage characteristic, the database server uses the sbspace value as
  the system-specified storage characteristic.

  The database administrator (DBA) defines storage characteristics for an sbspace
  with the **onspaces** utility.

* If the sbspace in which the smart large object is stored has not specified a value
  for a particular storage characteristic, the database server uses the system default
  as the system-specified storage characteristic.

  The database server defines the system defaults for storage characteristics
  internally. However, you can specify a default sbspace name with the
  SBSPACENAME configuration parameter of the ONCONFIG file. Also, an
  application call to **ifx_lo_col_info()** or **ifx_lo_specset_sbspace()** can supply the
  target sbspace in the LO-specification structure.

Warning: An error occurs if you do not specify the sbspacename configuration
parameter and the lo-specification structure does not contain the name
of the target sbspace.

It is recommended that you use the values for the system-specified disk-storage
information. Most applications do not need to change these system-specified
storage characteristics. For more information about database server and sbspace
storage characteristics, see the description of the **onspaces** utility in your *IBM
Informix Administrator's Guide*.

To use the system-specified storage characteristics for a new smart large object,
follow these steps:

1. Use the **ifx_lo_def_create_spec()** function to allocate an LO-specification
   structure and to initialize this structure to null values.

2. Pass this LO-specification structure to the **ifx_lo_create_function** function to
   create the instance of the smart large object.

The **ifx_lo_create()** function creates a smart-large-object instance with the storage
characteristics in the LO-specification structure that it receives as an argument.

Because the previous call to **ifx_lo_def_create_spec()** stored null values in this structure, the database server assigns the system-specified characteristics to the new instance of the smart large object.

The column-level storage characteristics: The database administrator (DBA) assigns column-level storage characteristics with the CREATE TABLE statement. The PUT clause of CREATE TABLE specifies storage characteristics for a particular smart-large-object (CLOB or BLOB) column. (For more information, see the description of the CREATE TABLE statement in the *IBM Informix Guide to SQL: Syntax*.) The **syscolattribs** system catalog table stores column-level storage characteristics.

The **ifx_lo_col_info()** function obtains column-level storage characteristics for a smart-large-object column. To use the column-level storage characteristics for a new smart-large-object instance, follow these steps:

1. Use the **ifx_lo_def_create_spec()** function to allocate an LO-specification structure and initialize this structure to null values.
2. Pass this LO-specification structure to the **ifx_lo_col_info()** function and specify the desired column and table name as arguments.

   The function stores the column-level storage characteristics into the specified LO-specification structure.
3. Pass this same LO-specification structure to the **ifx_lo_create()** function to create the instance of the smart large object.

When the **ifx_lo_create()** function receives the LO-specification structure as an argument, this structure contains the column-level storage characteristics that the previous call to **ifx_lo_col_info()** stored. Therefore, the database server assigns these column-level characteristics to the new instance of the smart large object.

When you use the column-level storage characteristics, you do not usually need to provide the name of the sbspace for the smart large object. The sbspace name is specified in the PUT clause of the CREATE TABLE statement or by the SBSPACENAME parameter in the ONCONFIG file.

The User-Defined Storage Characteristics: An Informix ESQL/C application program can define a unique set of storage characteristics for a new smart large object, as follows:

- For smart large objects that are to be stored in a column, you can override some storage characteristics for the column when it creates an instance of a smart large object.

   If the application does not override some or all of these characteristics, the smart large object uses the column-level storage characteristics. For information about how to obtain column-level storage characteristics, see "The column-level storage characteristics" on page 8-8.
- You can specify a wider set of characteristics for each smart large object since the smart large object is not constrained by table column properties.

   If the application programmer does not override some or all of these characteristics, the smart large object inherits the system-specified storage characteristics. For more information, see "The system-specified storage characteristics" on page 8-7.

To specify user-defined storage characteristics, use the appropriate Informix ESQL/C accessor functions for the LO-specification structure. For more information about these accessor functions, see "The LO-specification structure" on page 8-3.

Deallocating the LO-Specification Structure

Once you are finished with an LO-specification structure, deallocate the resources assigned to it with the **ifx_lo_spec_free()** function. When the resources are freed, they can be reallocated to other structures that your program needs. For more information about the **ifx_lo_spec_free()** function, see B-122.

The LO-Pointer Structure

To open a smart large object for read and write operations, an IBM Informix ESQL/C program must have an *LO-pointer structure* for the smart large object. This structure contains the disk address and a unique hexadecimal identifier for a smart large object. For more information about the unique identifier, see the description of the **ifx_lo_copy_to_file()** function in B-108.

To create an LO-pointer structure for a new smart large object, use the **ifx_lo_copy_to_file()** function. The **ifx_lo_copy_to_file()** function performs the following tasks:

1. It initializes an LO-pointer structure, whose pointer you provide as an argument, for the new smart large object.

   This new smart large object has the storage characteristics that the LO-specification structure you provide specifies. For more information about the LO-specification structure, see "The ifx_lo_create_spec_t structure" on page 8-3.

2. It opens the new smart large object in the specified access mode and returns an LO file descriptor that is needed for subsequent operations on the smart large object.

   For more information about an LO file descriptor, see "Duration of an Open on a Smart Large Object" on page 8-17.

You must call **ifx_lo_def_create_spec()** before you call the **ifx_lo_create()** function to create a new smart large object. For more information about the **ifx_lo_def_create_spec()** function, see B-112.

Storing a Smart Large Object

An IBM Informix ESQL/C program accesses a smart large object through an LO-pointer structure. The Informix ESQL/C library functions in the table from 8-11 accept an LO-pointer structure as an argument. Through the LO-pointer structure, these functions allow you to create and manipulate a smart large object without binding it to a database row.

An INSERT or UPDATE statement does not perform the actual input of the smart-large-object data. It does, however, provide a means for the application program to identify which smart-large-object data to associate with the column. A CLOB or BLOB column in a database table stores the LO-pointer structure for a smart large object. Therefore, when you store a CLOB or BLOB column, you provide an LO-pointer structure for the column in an **ifx_lo_t** host variable to the INSERT or UPDATE statement. For this reason, you declare host variables for CLOB and BLOB values as LO-pointer structures. (For more information, see "Declaring a Host Variable" on page 8-2.)

Figure 8-2 shows how the Informix ESQL/C client application transfers the data of a smart large object to the database server.

ESQL/C Client Application

1. Use ESQL/C API functions to create an LO-pointer structure.
2. Initialize the smart-large-object data through LO-pointer structure.

3. Execute SQL INSERT or UPDATE statement to assign the LO-pointer structure to a CLOB or BLOB column.
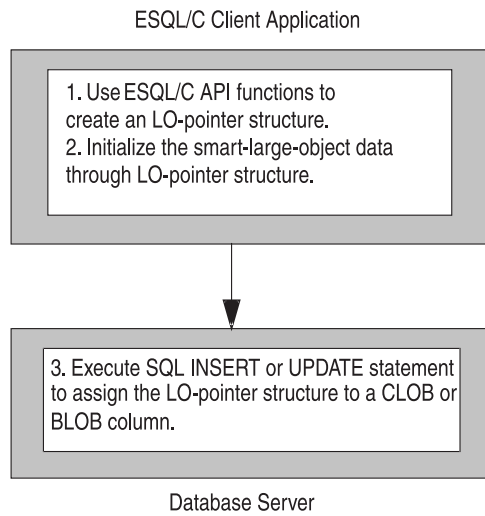
Database Server

Figure 8-2. Transferring Smart- Large-Object Data from Client Application to Database Server

The smart large object that an LO-pointer structure identifies exists as long as its LO-pointer structure exists. When you store an LO-pointer structure in the database, the database server can ensure that the smart large objects are deallocated when appropriate.

When you retrieve a row and then update a smart large object which that row contains, the database server exclusively locks the row for the time that it updates the smart large object. Moreover, long updates for smart large objects (whether or not logging is enabled and whether or not they are associated with a table row) create the potential for a long transaction condition if the smart large object takes a long time to update or create.

For an example of code that stores a new smart large object into a database column, see "The create_clob.ec program" on page C-2. For information about how to select a smart large object from the database, see "Selecting a Smart Large Object" on page 8-12.

The ifx_lo_t Structure

The LO-pointer structure, **ifx_lo_t**, serves as a reference to a smart large object. It provides security-related information and holds information about the actual disk location of the smart large object. The **locator.h** header file defines the LO-pointer structure so you must include the **locator.h** file in your IBM Informix ESQL/C programs that access this structure.

Important: The LO-pointer structure, **ifx_lo_t**, is an opaque structure to Informix ESQL/C programs. That is, you should not access its internal structure directly. The internal structure of **ifx_lo_t** may change. Therefore, to create portable code, use the appropriate Informix ESQL/C library function (see 8-11) to use this structure.

The LO-pointer structure, not the CLOB or BLOB data itself, is stored in a CLOB or BLOB column in the database. Therefore, SQL statements such as INSERT and SELECT accept an LO-pointer structure as the column value for a smart-large-object column. You declare an Informix ESQL/C host variable to hold

the value of a smart large object as an **ifx_lo_t** structure. For more information about how to declare a host variable for a smart large object, see "Declaring a Host Variable" on page 8-2.

ESQL/C functions that use the LO-pointer structure

The following table shows the IBM Informix ESQL/C library functions that access the LO-pointer structure and how they access it.

ESQL/C library function	Purpose	See
ifx_lo_copy_to_file()	Copies the smart large object that the LO-pointer structure identifies to an operating-system file.	B-108
ifx_lo_create()	Initializes an LO-pointer structure for a new smart large object that it creates and returns an LO file descriptor for this smart large object.	B-111
ifx_lo_filename()	Returns the name of the file where the **ifx_lo_copy_to_file()** function would store the smart large object that the LO-pointer structure identifies.	B-113
ifx_lo_from_buffer()	Copies a specified number of bytes from a user-defined buffer into the smart large object that the LO-pointer structure references.	B-114
ifx_lo_release()	Tells the database server to release the resources associated with the temporary smart large object that the LO-pointer structure references.	B-114
ifx_lo_to_buffer()	Copies a specified number of bytes from the smart large object referenced by the LO-pointer structure into a user-defined buffer.	B-141

The LO file descriptor

The LO file descriptor is an integer value that identifies an open smart large object. An LO file descriptor is very similar to the file descriptors for operating-system files. It serves as an I/O handle to the data of the smart large object in the server. LO file descriptors start with a seek position of zero (0). Use the LO file descriptor in one of the Informix ESQL/C library functions that accepts LO file descriptors.

ESQL/C library functions that use an LO file descriptor

The following table shows the IBM Informix ESQL/C library functions that access the LO file descriptor.

ESQL/C library function	Purpose	See
ifx_lo_close()	Closes the smart large object that the LO file descriptor identifies and deallocates the LO file descriptor	B-107
ifx_lo_copy_to_lo()	Copies an operating-system file to an open smart large object that the LO file descriptor identifies	B-110
ifx_lo_create()	Creates and opens a new smart large object and returns an LO file descriptor	B-111
ifx_lo_open()	Opens a smart large object and returns an LO file descriptor	B-116
ifx_lo_read()	Reads data from the open smart large object that the LO file descriptor identifies	B-118

ESQL/C library function	Purpose	See
ifx_lo_readwithseek()	Seeks a specified file position in the open smart large object that the LO file descriptor identifies and then reads data from this position	B-119
ifx_lo_seek()	Moves the file position in the open smart large object that the LO file descriptor identifies	B-121
ifx_lo_stat()	Obtains status information for the open smart large object that the LO file descriptor identifies	B-133
ifx_lo_tell()	Determines the current file position in the open smart large object that the LO file descriptor identifies	B-140
ifx_lo_truncate()	Truncates at a specified offset the open smart large object that the LO file descriptor identifies	B-142
ifx_lo_write()	Writes data to the open smart large object that the LO file descriptor identifies	B-143
ifx_lo_writewithseek()	Seeks a specified file position in the open smart large object that the LO file descriptor identifies and then writes data to this position	B-144

Creating a Smart Large Object

Perform the following steps to create a smart large object:

1. Allocate an LO-specification structure with the **ifx_lo_def_create_spec()** function.
2. Ensure that the LO-specification structure contains the desired storage characteristics for the new smart large object.
3. Create an LO-pointer structure for the new smart large object and open the smart large object with the **ifx_lo_create()** function.
4. Write the data for the new smart large object to the open smart large object with the **ifx_lo_write()** or **ifx_lo_writewithseek()** function. For more information, see "Writing Data to a Smart Large Object" on page 8-17.
5. Save the new smart large object in a column of the database.
6. Deallocate the LO-specification structure with the **ifx_lo_spec_free()** function.

For more information about how to create a smart large object, see "The LO-specification structure" on page 8-3.

Accessing a Smart Large Object

To access a smart large object, take the following steps:

1. Select the smart large object from the database into an **ifx_lo_t** host variable with the SELECT statement.
2. Open the smart large object with the **ifx_lo_open()** function.
3. Perform the appropriate read or write operations to update the data of the smart large object.
4. Close the smart large object with the **ifx_lo_close()** function.

Selecting a Smart Large Object

A SELECT statement does not perform the actual output for the smart-large-object data. It does, however, establish a means for the application program to identify a

smart large object so that it can then issue Informix ESQL/C library functions to open, read, write, or perform other operations on the smart large object.

A CLOB or BLOB column in a database table contains the LO-pointer structure for a smart large object. Therefore, when you select a CLOB or BLOB column into an **ifx_lo_t** host variable, the SELECT statement returns an LO-pointer structure. For this reason, you declare host variables for CLOB and BLOB values as LO-pointer structures. For more information, see "Declaring a Host Variable" on page 8-2.

Figure 8-3 shows how the database server transfers the data of a smart large object to the Informix ESQL/C client application.
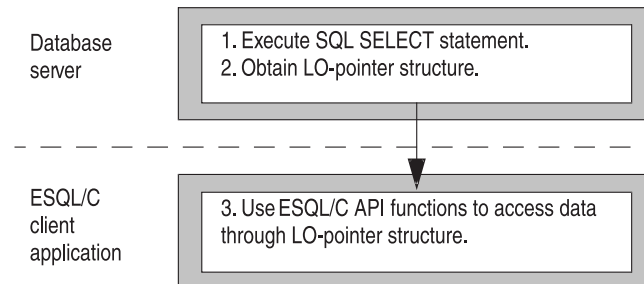
Figure 8-3. Transferring Smart- Large-Object Data from Database Server to Client Application

For an example of code that selects a smart large object from a database column, see "The create_clob.ec program" on page C-2. For information about how to store a smart large object in the database, see "Storing a Smart Large Object" on page 8-9.

Opening a Smart Large Object

When you open a smart large object, you obtain an LO file descriptor for the smart large object. Through the LO file descriptor you can access the data of a smart large object as if it were in an operating-system file.

Access Modes

When you open a smart large object, you specify the access mode for the data. The access mode determines which read and write operations are valid on the open smart large object. You specify an access mode with one of the access-mode constants that the **locator.h** file defines.

Table 8-3 shows the access modes and their corresponding defined constants that the **ifx_lo_open()** and **ifx_lo_create()** functions support.

Table 8-3. Access-Mode Flags for Smart Large Objects

Access Mode	Purpose	Access-Mode Constant
Read-only mode	Only read operations are valid on the data.	LO_RDONLY
Dirty-read mode	For **ifx_open()** only, allows you to read uncommitted data pages for the smart large object. You cannot write to a smart large object after you set the mode to LO_DIRTY_READ. When you set this flag, you reset the current transaction isolation mode to dirty read for the smart large object. Do not base updates on data that you obtain from a smart large object in dirty-read mode.	LO_DIRTY_READ
Write-only mode	Only write operations are valid on the data.	LO_WRONLY
Append mode	Intended for use in conjunction with LO_WRONLY or LO_RDWR. Sets the location pointer to the end of the object immediately prior to each write. Appends any data you write to the end of the smart large object. If LO_APPEND is used alone, the object is opened for reading only.	LO_APPEND
Read/write mode	Both read and write operations are valid on the data.	LO_RDWR
Buffered access	Use standard database server buffer pool.	LO_BUFFER
Lightweight I/O	Use private buffers from the session pool of the database server.	LO_NOBUFFER
Lock all	Specify that locking will occur for an entire smart large object.	LO_LOCKALL
Lock byte range	Specify that locking will occur for a range of bytes, which will be specified through the **ifx_lo_lock()** function when the lock is placed.	LO_LOCKRANGE

> **Tip:** These access-mode flags for a smart large object are patterned after the UNIX System V access modes.

Setting Dirty Read Access Mode: To set dirty read isolation mode for a smart large object, set it for the transaction with the SET ISOLATION statement, or set the LO_DIRTY_READ access mode when you open the smart large object. Setting the LO_DIRTY_READ access mode when you open the smart large object affects the read mode only for the smart large object and not for the entire transaction. In other words, if your transaction is executing in committed-read mode, you can use the LO_DIRTY_READ access mode to open the smart large object in dirty-read mode, without changing the isolation mode for the transaction.

For more information about dirty read isolation mode, see the SET ISOLATION statement in the *IBM Informix Guide to SQL: Syntax*.

Using LO_APPEND: When you open a smart large object with LO_APPEND only, the smart large object is opened as read-only. Seek operations move the file pointer but write operations to the smart large object fail and the file pointer is not moved from its position just before the write. Read operations occur from where the file pointer is positioned and then the file pointer is moved.

You can mask the LO_APPEND flag with another access mode. In any of these OR combinations, the seek operation remains unaffected. The following table shows the effect on the read and write operations that each of the OR combinations has.

OR Operation	Read Operations	Write Operations
LO_RDONLY | LO_APPEND	Occur at the file position and then move the file position to the end of the data that was read	Fail and do not move the file position.
LO_WRONLY | LO_APPEND	Fail and do not move the file position	Move the file position to the end of the smart large object and then write the data; file position is at the end of the data after the write.
LO_RDWR | LO_APPEND	Occur at the file position and then move the file position to the end of the data that was read	Move the file position to the end of the smart large object and then write the data; file position is at the end of the data after the write.

Lightweight I/O: When the database server accesses smart large objects, it uses buffers from the buffer pool for buffered access. Unbuffered access is called *lightweight I/O*. Lightweight I/O uses private buffers instead of the buffer pool to hold smart large objects. These private buffers are allocated out of the database server session pool.

Lightweight I/O allows you to bypass the overhead of the least-recently-used (LRU) queues that the database server uses to manage the buffer pool. For more information about LRUs, see your *IBM Informix Performance Guide*.

You can specify lightweight I/O by setting the flags parameter to LO_NOBUFFER when you create a smart large object with the **ifx_lo_create()** function or when you open a particular smart large object with the **ifx_lo_open()** function. To specify buffered access, which is the default, use the LO_BUFFER flag.

Important: Keep the following issues in mind when you use lightweight I/O:
- Close smart large objects with ifx_lo_close() when you are finished with them to free memory allocated to the private buffers.
- All opens that use lightweight I/O for a particular smart large object share the same private buffers. Consequently, one operation can cause the pages in the buffer to be flushed while other operations expect the object to be present in the buffer.

The database server imposes the following restrictions on switching from lightweight I/O to buffered I/O:
- You can use the **ifx_lo_alter()** function to switch a smart large object from lightweight I/O (LO_NOBUFFER) to buffered I/O (LO_BUFFER) if the smart large object is *not* open. However, **ifx_lo_alter()** generates an error if you try to change a smart large object that uses buffered I/O to one that uses lightweight I/O.
- Unless you first use **ifx_lo_alter()** to change the access mode to buffered access (LO_BUFFER), you can only open a smart large object that was created with lightweight I/O with the LO_NOBUFFER access-mode flag. If an open specifies LO_BUFFER, the database server ignores the flag.
- You can open a smart large object that was created with buffered access (LO_BUFFER) with the LO_NOBUFFER flag only if you open the object in read-only mode. If you attempt to write to the object, the database server returns an error. To write to the smart large object, you must close it then reopen it with the LO_BUFFER flag and an access flag that allows write operations.

You can use the database server utility **onspaces** to specify lightweight I/O for all smart large objects in an sbspace. For more information about the **onspaces** utility, see your *IBM Informix Administrator's Guide*.

Smart-Large-Object Locks

When you open a smart large object the database server locks either the entire smart large object or a range of bytes that you specify to prevent simultaneous access to smart-large-object data, Locks on smart large objects are different than row locks. If you retrieve a smart large object from a row, the database server might hold a row lock as well as a smart-large-object lock. The database server locks smart-large-object data because many columns can contain the same smart-large-object data.You use the access-mode flags, LO_RDONLY, LO_DIRTY_READ, LO_APPEND, LO_WRONLY, LO_RDWR, and LO_TRUNC to specify the lock *mode* of a smart large object. You pass these flags to the **ifx_lo_open()** and **ifx_lo_create()** functions. When you specify LO_RDONLY, the database server places a share lock on the smart large object. When you specify LO_DIRTY_READ, the database server does not place a lock on the smart large object. If you specify any other access-mode flag, the database server obtains an update lock, which it promotes to an exclusive lock on first write or other update operation.

Share and update locks (read-only mode, or write mode before an update operation occurs) are held until your program takes one of the following actions:
- Closes the smart large object
- Commits the transaction or rolls it back

Exclusive locks are held until the end of a transaction even if you close the smart large object.

Important: You lose the lock at the end of a transaction, even if the smart large object remains open. When the database server detects that a smart large object has no active lock, it automatically obtains a new lock when the first access occurs to the smart large object. The lock it obtains is based on the original open mode of the smart large object.

Range of a Lock: When you place a lock on a smart large object you can lock either the entire smart large object or you can lock a *byte range*. A byte range lock allows you to lock only the range of bytes that you will affect within the smart large object.

Two access-mode flags, LO_LoCKALL and LO_LOCKRANGE, enable you to designate the default type of lock that will be used for the smart large object. You can set them with **ifx_lo_specset_flags()** and retrieve them with **ifx_specget_flags()**. The LO_LOCKALL flag specifies that the entire smart large object will be locked; the LO_LOCKRANGE flag specifies that you will use byte-range locks for the smart large object. For more information, see "The ifx_lo_specget_flags() function" on page B-125 and "The ifx_lo_specset_flags() function" on page B-131.

You can use the **ifx_lo_alter()** function to change the default range from one type to the other. You can also override the default range by setting either the LO_LOCKALL or the LO_LOCKRANGE flag in the access-mode flags for **ifx_lo_open()**. For more information, see "Opening a Smart Large Object" on page 8-13 and "The ifx_lo_open() function" on page B-116.

The **ifx_lo_lock()** function allows you to lock a range of bytes that you want to access for a smart large object and the **ifx_lo_unlock()** function allows you to unlock the bytes when you are finished. For more information, see "The ifx_lo_lock() function" on page B-115 and "The ifx_lo_unlock() function" on page B-142.

Duration of an Open on a Smart Large Object

Once you open a smart large object with the **ifx_lo_create()** function or the **ifx_lo_open()** function, it remains open until one of the following events occurs:
- The **ifx_lo_close()** function closes the smart large object.
- The session ends.

Warning: The end of the current transaction does not close a smart large object. It does, however, release any lock on a smart large object.

Have your applications close smart large objects as soon as they finish with them. Leaving smart large objects open unnecessarily consumes system memory. Leaving a sufficient number of smart large objects open can eventually produce an out-of-memory condition.

Deleting a Smart Large Object

A smart large object is not deleted until *both* of the following conditions are met:
- The current transaction commits.
- The smart large object is closed, if the application opened the smart large object.

Modifying a Smart Large Object

You modify the data of the smart large object with the following steps:
1. Read and write the data in the open smart large object until the data is ready to save.
2. Store the LO-pointer for the smart large object in the database with the UPDATE or INSERT statement.

For information about how to save the smart large object, see "Storing a Smart Large Object" on page 8-9.

Reading Data From a Smart Large Object

The **ifx_lo_read()** and **ifx_lo_readwithseek()** Informix ESQL/C library functions read data from an open smart large object. They both read a specified number of bytes from the open smart large object into the user-defined character buffer. The **ifx_lo_read()** function begins the read operation at the current file position. You can specify the starting file position of the read with the **ifx_lo_seek()** function, and you can obtain the current file position with the **ifx_lo_tell()** function. The **ifx_lo_readwithseek()** function performs the seek and read operations with a single function call.

The **ifx_lo_read()** and **ifx_lo_readwithseek()** functions require a valid LO file descriptor to identify the smart large object to be read. You obtain an LO file descriptor with the **ifx_lo_open()** or **ifx_lo_create()** function. For more information, see the descriptions of the **ifx_lo_read()** function in B-118 and the **ifx_lo_readwithseek()** function in B-119.

Writing Data to a Smart Large Object

The **ifx_lo_write()** and **ifx_lo_writewithseek()** Informix ESQL/C library functions write data to an open smart large object. They both write a specified number of

bytes from a user-defined character buffer to the open smart large object. The **ifx_lo_write()** function begins the write operation at the current file position. You can specify the starting file position of the write with the **ifx_lo_seek()** function, and you can obtain the current file position with the **ifx_lo_tell()** function. The **ifx_lo_writewithseek()** function performs the seek and write operations with a single function call.

The **ifx_lo_write()** and **ifx_lo_writewithseek()** functions require a valid LO file descriptor to identify the smart large object to write. You obtain an LO file descriptor with the **ifx_lo_open()** or **ifx_lo_create()** function. For more information, see the descriptions of the **ifx_lo_write()** function in B-143 and the **ifx_lo_writewithseek()** function in B-144.

Closing a Smart Large Object

Once you have finished the read and write operations on the smart large object, deallocate the resources assigned to it with the **ifx_lo_close()** function. When the resources are freed, they can be reallocated to other structures that your program needs. In addition, the LO file descriptor can be reallocated to other smart large objects. For more information about the **ifx_lo_close()** function, see B-107.

Obtaining the Status of a Smart Large Object

To obtain status information for a smart large object, take the following steps:

1. Obtain a valid LO-pointer structure to the smart large object for which you want status.
2. Allocate and fill an LO-status structure with the **ifx_lo_stat()** function
3. Use the appropriate IBM Informix ESQL/C accessor function to obtain the status information you need.
4. Deallocate the LO-status structure.

Obtaining a Valid LO-Pointer Structure

You can obtain status information for any smart large object for which you have a valid LO-pointer structure. You can perform either of the following steps to obtain an LO-pointer structure:

- Select a CLOB or BLOB column from a database table.

  For more information, see "Selecting a Smart Large Object" on page 8-12.

- Create a new smart large object.

  For more information, see "The LO-Pointer Structure" on page 8-9.

Allocating and Accessing an LO-Status Structure

The LO-status structure stores status information for a smart large object. This section describes how to allocate and access an LO-status structure.

Allocating an LO-Status Structure

The **ifx_lo_stat()** function performs the following tasks:

- It allocates a new LO-status structure, whose pointer you provide as an argument.
- It initializes the LO-status structure with all status information for the smart large object that the LO file descriptor, which you provide, identifies.

For more information about the **ifx_lo_stat()** function, see B-133.

Accessing the LO-Status Structure

The LO-status structure, **ifx_lo_stat_t**, stores the status information for a smart large object in an IBM Informix ESQL/C program. The **locator.h** header file defines the LO-status structure so you must include the **locator.h** file in your Informix ESQL/C programs that access this structure.

Important: The LO-status structure, **ifx_lo_stat_t**, is opaque to Informix ESQL/C programs. Do not access its internal structure directly. The internal structure of **ifx_lo_stat_t** may change in future releases. Therefore, to create portable code, always use the Informix ESQL/C accessor functions for this structure to obtain and store values in the LO-status structure.

The following table shows the status information along with the corresponding Informix ESQL/C accessor functions.

Table 8-4. Status Information in the LO-Status Structure

Disk-Storage Information	Description	ESQL/C Accessor Functions
Last access time	The time, in seconds, that the smart large object was last accessed. This value is available only if the LO_KEEP_LASTACCESS_TIME flag is set for this smart large object.	**ifx_lo_stat_atime()**
Storage characteristics	The storage characteristics for the smart large object. These characteristics are stored in an LO-specification structure (see 8-3). Use the Informix ESQL/C accessor functions for an LO-specification structure (see Table 8-1 on page 8-4 and Table 8-2 on page 8-5) to obtain this information.	**ifx_lo_stat_cspec()**
Last change in status	The time, in seconds, of the last status change for the smart large object. A change in status includes updates, changes in ownership, and changes to the number of references.	**ifx_lo_stat_ctime()**
Last modification time (seconds)	The time, in seconds, that the smart large object was last modified.	**ifx_lo_stat_mtime_sec()**
Reference count	A count of the number of references to the smart large object.	**ifx_lo_stat_refcnt()**
Size	The size, in bytes, of the smart large object.	**ifx_lo_stat_size()**

The time values (such as last access time and last change time) might differ slightly from the system time. This difference is due to the algorithm that the database server uses to obtain the time from the operating system.

Deallocating the LO-Status Structure

Once you have finished with an LO-status structure, deallocate the resources assigned to it with the **ifx_lo_stat_free()** function. When the resources are freed, they can be reallocated to other structures that your program needs. For more information about the **ifx_lo_stat_free()** function, see B-137.

Altering a Smart-Large-Object Column

You can use the PUT clause of the ALTER TABLE statement to change the storage location and the storage characteristics of a CLOB or BLOB column. You can change the sbspace where the column is stored and also implement round-robin fragmentation, which causes the smart large objects in the CLOB or BLOB column to be distributed among a series of specified sbspaces. For example, the ALTER TABLE statement in the following example changes the original storage location of the **advert.picture** column from **s9_sbspc** to the sbspaces **s10_sbspc** and **s11_sbspc**. The ALTER TaBLE statement also changes the characteristics of the column:

```
advert          ROW (picture BLOB, caption VARCHAR(255, 65)),
.
.
.

PUT advert IN (s9_sbspc)
   (EXTENT SIZE 100)

ALTER TABLE catalog
   PUT advert IN (s10_sbspc, s11_sbspc)
   (extent size 50, NO KEEP ACCESS TIME);
```

When you change the storage location or storage characteristics of a smart-large-object column, the changes apply only to new instances created for the column. The storage location and storage characteristics of existing smart large objects for the column are not affected.

For a description of the **catalog** table that the preceding example references, see the Appendix C, "Examples for Smart-Large-Object Functions," on page C-1.

For more information about the ALTER TABLE statement, see the *IBM Informix Guide to SQL: Syntax*.

Migrating Simple Large Objects

To migrate simple large objects to smart large objects, cast TEXT data to CLOB data and BYTE data to BLOB data. You can use the cast syntax (`bytecolblobcol`, for example) to migrate a simple large object to a smart large object. The following example migrates the BYTE column **cat_picture** from the **catalog** table in the **stores7** database to the BLOB field **picture** in the **advert** row type in the alternate **catalog** table that is described in Appendix C, "Examples for Smart-Large-Object Functions," on page C-1:

```
update catalog set advert = ROW ((SELECT cat_picture::blob
   FROM stores7:catalog WHERE catalog_num = 10027), pwd
   advert.caption)
   WHERE catalog_num = 10027
```

For a description of the **stores7** table, see *IBM Informix Guide to SQL: Reference*.

You can also use the MODIFY clause of the ALTER TABLE statement to change a TEXT or BYTE column to a CLOB or BLOB column. When you use the MODIFY clause of the ALTER TABLe statement, the database server implicitly casts the old data type to the new data type to create the ClOB or BLOB column.

For example, if you want to change the **cat_descr** column from a TEXT column to a BYTE column in the **catalog** table of the **stores7** database, you can use a construction similar to the following statement:

```
ALTER TABLE catalog modify cat_descr CLOB,
   PUT cat_descr in (sbspc);
```

For more information about the ALTER TABLE statement, see the *IBM Informix Guide to SQL: Syntax*

For more information about casting, see the *IBM Informix Guide to SQL: Syntax* and the *IBM Informix Guide to SQL: Tutorial.*

Reading and Writing Smart Large Objects on an Optical Disc (UNIX)

Within a table, rows that include smart-large-object data do not include the smart-large-object data in the row itself. Instead, the smart-large-object column contains the LO-pointer structure (see "The LO-specification structure" on page 8-3). The LO-pointer structure can point to an sbpage in an sbspace or to a platter in an optical storage subsystem.

However, you can store smart large objects on optical disc *only* if this media is mounted as a UNIX file system and is *write many* (WMRM). The optical disc must contain the sbspaces for the smart large objects. Your application can use the Informix ESQL/C API for smart large objects (which this chapter describes) to access the smart large objects on the mounted optical disc.

The database server does *not* provide support for a write-once-read-many (WORM) optical-storage subsystem as a location for smart large objects. However, it does support access to simple large objects (BYTE and TEXT) on WORM media. For more information, see "Reading and Writing Simple Large Objects to an Optical Disc (UNIX)" on page 7-28.

For details about the optical subsystem, see your *IBM Informix Administrator's Guide* and the *IBM Informix Optical Subsystem Guide.*

The ESQL/C API for Smart Large Objects

The IBM Informix ESQL/C API for smart large objects allows an application program to access a smart large object much like an operating-system file. A smart large object that does not fit into memory does not have to be read into a file and then accessed from a file; it can be accessed one piece at a time. An Informix ESQL/C application program accesses smart large objects through the Informix ESQL/C library functions in the following table. The library functions are located in Appendix B, "ESQL/C function library," on page B-1.

ESQL/C Function	Description	See
ifx_lo_alter()	Alters the storage characteristics of an existing smart large object	B-106
ifx_lo_close()	Closes an open smart large object	B-107
ifx_lo_col_info()	Retrieves column-level storage characteristics in an LO-specification structure	B-107
ifx_lo_copy_to_file()	Copies a smart large object into an operating-system file	B-108
ifx_lo_copy_to_lo()	Copies an operating-system file into an open smart large object	B-110
ifx_lo_create()	Creates an LO-pointer structure for a smart large object	B-111

ESQL/C Function	Description	See
ifx_lo_def_create_spec()	Allocates an LO-specification structure and initializes its fields to null values	B-112
ifx_lo_filename()	Returns the generated file name, given an LO-pointer structure and a file specification	B-113
ifx_lo_from_buffer()	Copies a specified number of bytes from a user-defined buffer into a smart large object	B-114
ifx_lo_open()	Opens an existing smart large object	B-116
ifx_lo_read()	Reads a specified number of bytes from an open smart large object	B-118
ifx_lo_readwithseek()	Seeks to a specified position in an open smart large object and reads a specified number of bytes	B-119
ifx_lo_release()	Releases resources committed to a temporary smart large object	B-120
ifx_lo_seek()	Sets the seek position for the next read or write on an open smart large object	B-121
ifx_lo_spec_free()	Frees the resources allocated to an LO-specification structure	B-122
ifx_lo_specget_estbytes()	Gets the estimated size, in bytes, of the smart large object	B-123
ifx_lo_specget_extsz()	Gets the allocation extent size for the smart large object	B-124
ifx_lo_specget_flags()	Gets the create-time flags for the smart large object	B-125
ifx_lo_specget_maxbytes()	Gets the maximum size for the smart large object	B-126
ifx_lo_specset_sbspace()	Gets the sbspace name for the smart large object	B-127
ifx_lo_specset_estbytes()	Sets the estimated size, in bytes, of the smart large object	B-129
ifx_lo_specset_extsz()	Sets the allocation extent size for the smart large object	B-130
ifx_lo_specset_flags()	Sets the create-time flags for the smart large object	B-131
ifx_lo_specset_maxbytes()	Sets the maximum size for the smart large object	B-132
ifx_lo_specset_sbspace()	Sets the sbspace name for the smart large object	B-132
ifx_lo_stat()	Obtains status information for an open smart large object	B-133
ifx_lo_stat_atime()	Returns the last access time for a smart large object	B-134
ifx_lo_stat_cspec()	Returns the storage characteristics for a smart large object	B-135
ifx_lo_stat_ctime()	Returns the last change-in-status time for the smart large object	B-136
ifx_lo_stat_free()	Frees the resources allocated to an LO-status structure	B-137

ESQL/C Function	Description	See
ifx_lo_stat_mtime_sec()	Returns the last modification time, in seconds, for the smart large object	B-138
ifx_lo_stat_refcnt()	Returns the reference count for the smart large object	B-138
ifx_lo_stat_size()	Returns the size of the smart large object	B-139
ifx_lo_tell()	Returns the current seek position of an open smart large object	B-140
ifx_lo_to_buffer()	Copies a specified number of bytes from a smart large object into a user-defined buffer	B-141
ifx_lo_truncate()	Truncates a smart large object to a specific offset	B-142
ifx_lo_write()	Writes a specified number of bytes to an open smart large object	B-143
ifx_lo_writewithseek()	Seeks to a specified position in an open smart large object and writes a specified number of bytes	B-144

Chapter 9. Working with complex data types

The information in these topics apply only if you are using IBM Informix as your database server.

These topics explain how to use **collection** and **row** data types in an IBM Informix ESQL/C program. These Informix ESQL/C data types access the complex data types, as the following table shows.

Data Type	ESQL/C Host Variable
Collection types: LIST, MULTISET, SET	Typed **collection** host variable
	Untyped **collection** host variable

Data Type	ESQL/C Host Variable
Row types: named and unnamed	Typed **row** host variable
	Untyped **row** host variable

For information about all the data types that are available for use in an Informix ESQL/C program, see Chapter 3, "Informix ESQL/C data types," on page 3-1. For information about SQL complex data types, see the *IBM Informix Guide to SQL: Reference*.

Accessing a collection

IBM Informix supports the following kinds of collections:

- The SET data type stores a collection of elements that are unique values and have no ordered positions.
- The MULTISET data type stores a collection of elements that can be duplicate values and have no ordered positions.
- The LIST data type stores a collection of elements that can be duplicate values and have ordered positions.

Both SQL and Informix ESQL/C enable you to use the SQL *collection derived table* clause to access the elements of a collection as if they were rows in a table. In Informix ESQL/C, the collection derived table takes the form of a *collection variable*. The collection variable is a host variable into which you retrieve the collection. Once you have retrieved the collection into a collection variable, you can perform select, insert, update, and delete operations on it, with restrictions. For more information about how to access a collection with an Informix ESQL/C collection variable, see "Declaring Collection Variables" on page 9-4, and more specifically, "Operating on a Collection Variable" on page 9-10.

Important: When the SQL statement references a collection variable, Informix ESQL/C and *not* the database server, processes the statement.

SQL allows you to perform read-only (SELECT) operations on a collection by implementing the collection derived table as a *virtual table*.

Accessing a Collection Derived Table

When the SELECT statement for a collection does not reference an Informix ESQL/C collection variable, the database server performs the query.

Consider, for example, the following schema:

```
create row type person(name char(255), id int);
create table parents(name char(255), id int,
   children list(person not null));
```

You can select the children's names and IDs from the table **parent** using the following SELECT statement:

```
select name, id from table(select children from parents
   where parents.id = 1001) c_table(name, id);
```

To execute the query, the database server creates a virtual table (**c_table**) from the list **children** in the row of the **parents** table where **parents.id** equals 1001.

Advantage of a Collection Derived Table

The advantage of querying a collection as a virtual table as opposed to querying it through a collection variable is that the virtual table provides more efficient access. By contrast, if you were to use collection variables, you might be required to allocate multiple variables and multiple cursors. For example, consider the following schema:

```
EXEC SQL create row type parent_type(name char(255), id int,
   children list(person not null));
EXEC SQL create grade12_parents(class_id int,
   parents set(parent_type not null));
```

You can query the collection derived table as a virtual table as shown in the following SELECT statement:

```
EXEC SQL select name into :host_var1
   from table((select children from table((select parents
   from grade12_parents where class_id = 1))
   p_table where p_table.id = 1001)) c_table
   where c_table.name like 'Mer%';
```

To perform the same query using collection variables you need to execute the following statements:

```
EXEC SQL client collection hv1;
EXEC SQL client collection hv2;
EXEC SQL int parent_id;

EXEC SQL char host_var1[256];
.
.
.

EXEC SQL allocate collection hv1;
EXEC SQL allocate collection hv2;

EXEC SQL select parents into :hv1 from grade12_parents
   where class_id = 1;
EXEC SQL declare cur1 cursor for select id, children
   from table(:hv1);
EXEC SQL open cur1;
for(;;)
{
   EXEC SQL fetch cur1 into :parent_id, :hv2;
   if(parent_id = 1001)
      break;
}
EXEC SQL declare cur2 cursor for select name from
   table(:hv2));
EXEC SQL open cur2;
for(;;)
{
   EXEC SQL fetch cur2 into :host_var1;
   /* user needs to implement 'like' function */
   if(like_function(host_var1, "Mer%"))
      break;
}
```

Restrictions on a Collection Derived Table

The following restrictions apply to querying a collection derived table that is a virtual table:

- It cannot be the target of INSERT, DELETE, or UPDATE statements.
- It cannot be the underlying table of any cursors or views that can be updated.
- It does not support ordinality. For example, it *does not* support the following statement:

```
select name, order_in_list from table(select children
   from parents where parents.id = 1001)
   with ordinality(order_in_list);
```

- It is an error if the underlying collection expression of the collection derived table evaluates to a null value.
- It cannot reference columns of tables that are referenced in the same FROM clause. For example, it does not support the following statement because the collection derived table **table(parents.children)** refers to the table **parents**, which is referenced in the FROM clause:

```
select count(distinct c_id) from parents,
   table(parents.children) c_table(c_name, c_id)
   where parents.id = 1001
```

- The database server must be able to statically determine the type of the underlying collection expression. For example, the database server cannot support:

```
TABLE(?)
```

- The database server cannot support a reference to a host variable without casting it to a known collection type. For example, rather than specifying `TABLE(:hostvar)`, you must cast the host variable:

```
TABLE(CAST(:hostvar AS type))
TABLE(CAST(? AS type))
```

- It will not preserve the order of rows in the list if the underlying collection is a list.

Declaring Collection Variables

To access the elements of a column that has a collection type (LIST, MULTISET, or SET) as its data type, perform the following steps:

1. Declare a **collection** host variable, either typed or untyped.
2. Allocate memory for the **collection** host variable.
3. Perform any select, insert, update, or delete operations on the **collection** host variable.
4. Save the contents of the **collection** host variable into the collection column.

Use the **collection** data type to declare host variables for columns of collection data types (SET, MULTISET, or LIST). As the following syntax diagram illustrates, you must use the **collection** keyword as the data type for a **collection** host variable.

```
       (1)
►►─────────client──collection───────────────────────────────────────────►
                                 ┌──set────┐  (──element type──not null──)┐
                                 ├─multiset─┤
                                 └─list─────┘


      ┌──,──────────┐
►──────▼─variable name─┴──;──────────────────────────────────────────────►◄
```
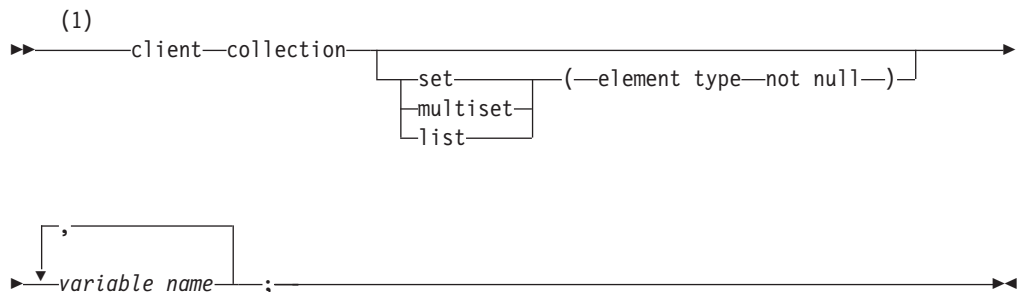
Notes:

1   Informix extension

Element	Purpose	Restrictions	SQL Syntax
element type	Data type of the elements in the **collection** variable	Can be any data type except SERIAL, SERIAL8, BIGSERIAL, TEXT, or BYTE	Data Type segment in the *IBM Informix Guide to SQL: Syntax*
variable name	Name of the Informix ESQL/C variable to declare as a **collection** variable		Name must conform to language-specific rules for variable names.

A **collection** variable can be any SQL collection type: LIST, MULTISET, or SET.

Important: You must specify the **client** keyword when you declare **collection** variables.

Typed and Untyped Collection Variables

Informix ESQL/C supports the following two **collection** variables:

- A *typed* **collection** variable specifies the data type of the elements in the collection and the collection itself.
- An *untyped* **collection** variable specifies neither the collection type nor the element type.

The Typed Collection Variable: A *typed* **collection** variable provides an exact description of the collection. This declaration specifies the data type of the collection (SET, MULTISET, or LIST) and the element type for the **collection** variable.

Figure 9-1 shows declarations for three typed **collection** variables.

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection list(smallint not null)
        list1;
    client collection set(row(
        x char(20),
        y set(integer not null),
        z decimal(10,2)) not null) row_set;
    client collection multiset(set(smallint
            not null)
        not null) collection3;
EXEC SQL END DECLARE SECTION;
```

Figure 9-1. Sample Typed Collection Variables

Typed **collection** variables can contain elements with the following data types:

- Any built-in data type (such as INTEGER, CHAR, BOOLEAN, and FLOAT) *except* BYTE, TEXT, SERIAL, or SERIAL8.
- Collection data types, such as SET and LIST, to create a *nested* collection
- Unnamed row types (named row types are *not* valid)
- Opaque data types

When you specify the element type of the **collection** variable, use the SQL data types, not the Informix ESQL/C data types. For example, as the declaration for the **list1** variable in Figure 9-1 on page 9-5 illustrates, use the SQL SMALLINT data type, not the Informix ESQL/C **short** data type, to declare a LIST variable whose

elements are small integers. Similarly, use the SQL syntax for a CHAR column to declare a SET variable whose elements are character strings, as the following example illustrates:

```
client collection set(char(20) not null) set_var;
```

Important: You must specify the not-null constraint on the element type of a **collection** variable.

A named row type is not valid as the element type of a **collection** variable. However, you can specify an element type of unnamed row type, whose fields match those of the named row type.

For example, suppose your database has the named row type, **myrow**, and the database table, **mytable**, that are defined as follows:

```
CREATE ROW TYPE myrow
(
   a int,
   b float
);
CREATE TABLE mytable
(
   col1 int8,
   col2 set(myrow not null)
);
```

You can define a **collection** variable for the **col2** column of **mytable** as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(row(a int, b float) not null)
      my_collection;
EXEC SQL END DECLARE SECTION;
```

You can declare a typed **collection** variable whose element type is different from that of the collection column as long as the two data types are compatible. If the database server is able to convert between the two element types, it automatically performs this conversion when it returns the fetched collection.

Suppose you create the **tab1** table as follows:

```
CREATE TABLE tab1 (col1 SET(INTEGER NOT NULL))
```

You can declare a typed **collection** variable whose element type matches (**set_int**) or one whose element type is compatible (**set_float**), as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(float not null) set_float;
   client collection set(integer not null) set_int;
EXEC SQL END DECLARE SECTION;

EXEC SQL declare cur1 cursor for select * from tab1;
EXEC SQL open cur1;
EXEC SQL fetch cur1 into:set_float;
EXEC SQL fetch cur1 into :set_int;
```

When it executes the first FETCH statement, the Informix ESQL/C client program automatically converts the integer elements in the column to the float values in the **set_float** host variable. The Informix ESQL/C program only generates a type-mismatch error if you change the host variable after the first fetch. In the preceding code fragment, the second FETCH statement generates a type-mismatch error because the initial fetch has already defined the element type as float.

Use a typed **collection** variable in the following cases:

- When you insert into a derived table (Informix ESQL/C needs to know what the type is)
- When you update an element in a derived table (Informix ESQL/C needs to know what the type is)
- When you want the server to perform a cast. (The Informix ESQL/C client sends the type information to the database server, which attempts to perform the requested cast operation. If it is not possible, the database server returns an error.)

Match the declaration of a typed **collection** variable exactly with the data type of the collection column. You can then use this **collection** variable directly in SQL statements such as INSERT, DELETE, or UPDATE, or in the collection-derived table clause. (For more information about the collection-derived table clause, see "Using the Collection-Derived Table Clause on Collections" on page 9-10.)

Tip: If you do not know the exact data type of the collection column you want to access, use an untyped **collection** variable.

For more information, see "The Untyped Collection Variable" on page 9-7.

In a single declaration line, you can declare several **collection** variables for the *same* typed collection, as the following declaration shows:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(integer not null) mset1, mset2;
EXEC SQL END DECLARE SECTION;
```

You cannot declare **collection** variables for different collection types in a single declaration line.

The Untyped Collection Variable: An *untyped* **collection** variable provides a general description of a collection. This declaration includes only the **collection** keyword and the variable name. The following lines declare three untyped **collection** variables:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection collection1, collection2;
   client collection grades;
EXEC SQL END DECLARE SECTION;
```

The advantage of an untyped **collection** host variable is that it provides more flexibility in collection definition. For an untyped **collection** variable, you do not have to know the definition of the collection column at compile time. Instead, you obtain, at runtime, a description of the collection from a collection column with the SELECT statement.

Tip: If you know the exact data type of the collection column you want to access, use a typed **collection** variable.

For more information, see 9-5.

To obtain the description of a collection column, execute a SELECT statement to retrieve the column into the untyped **collection** variable. The database server returns the column description (the collection type and the element type) with the column data. Informix ESQL/C assigns this definition of the collection column to the untyped **collection** variable.

For example, suppose the **a_coll** host variable is declared as an untyped **collection** variable, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection a_coll;
EXEC SQL END DECLARE SECTION;
```

The following code fragment uses a SELECT statement to initialize the **a_coll** variable with the definition of the **list_col** collection column (which Figure 9-2 on page 9-12 defines) *before* it uses the **collection** variable in an INSERT statement:

```
EXEC SQL allocate collection :a_coll;

/* select LIST column into the untyped collection variable
 * to obtain the data-type information */
EXEC SQL select list_col into :a_coll from tab_list;

/* Insert an element at the end of the LIST in the untyped
 * collection variable */
EXEC SQL insert into table(:a_coll) values (7);
```

For more information about the ALLOCATE COLLECTION statement, see "Managing Memory for Collections" on page 9-9. For more information about how to insert into a collection variable, see "Inserting into a Collection Variable" on page 9-13.

To obtain the description of a collection column, your application must verify that a collection column has data in it *before* it selects the column. If the table has no rows in it, the SELECT statement returns neither column data nor the column description and Informix ESQL/C cannot assign the column description to the untyped **collection** variable.

You can use an untyped **collection** variable to store collections with different column definitions, as long as you select the associated collection column description into the **collection** variable *before* you use the variable in an SQL statement.

Important: You must obtain the definition of a collection column for an untyped **collection** variable *before* you use the variable in an SQL statement. Before the **collection** variable can hold any values, you must use a SELECT statement to obtain a description of the collection data type from a collection column in the database. Therefore, you cannot insert or select values directly into an untyped **collection** variable.

Client Collections

The Informix ESQL/C application declares the **collection** variable name, allocates the memory for it with the ALLOCATE COLLECTION statement, and performs operations on the **collection** data.

To access the elements of a collection variable, specify the variable in the Collection Derived Table clause of a SELECT, INSERT, UPDATE, or DELETE statement. Informix ESQL/C will perform the select, insert, update, or delete operation. Informix ESQL/C does *not* send these statements to the database server when they include a client **collection** variable in the collection-derived table clause.

For example, Informix ESQL/C performs the following INSERT operation on the **a_multiset** collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(integer not null) a_multiset;
EXEC SQL END DECLARE SECTION;
EXEC SQL insert into table(:a_multiset) values (6);
```

When an SQL statement includes a **collection** variable, it has the following syntax restrictions:

- You can only access elements of a client-side collection with the collection-derived table clause and a SELECT, INSERT, UPDATE, or DELETE statement.
- An INSERT statement cannot have a SELECT, an EXECUTE FUNCTION, or an EXECUTE PROCEDURE statement in the VALUES clause.
- You cannot include a WHERE clause
- You cannot include an expression
- You cannot use scroll cursors

For more information about how to insert into a collection variable, see "Inserting into a Collection Variable" on page 9-13. For more information about a collection-derived table, see "Operating on a Collection Variable" on page 9-10.

Managing Memory for Collections

Informix ESQL/C does not automatically allocate or deallocate memory for **collection** variables. You must explicitly manage the memory that is allocated to a **collection** variable.

Use the following SQL statements to manage memory for both typed and untyped **collection** host variables:

- The ALLOCATE COLLECTION statement allocates memory for the specified **collection** variable.

  This **collection** variable can be a typed or untyped collection. The ALLOCATE COLLECTION statement sets **SQLCODE** (**sqlca.sqlcode**) to zero if the memory allocation was successful and a negative error code if the allocation failed.

- The DEALLOCATE COLLECTION statement deallocates memory for a specified **collection** variable.

  Once you free the **collection** variable with the DEALLOCATE COLLECTION statement, you can reuse the **collection** variable.

Important: You must explicitly deallocate memory allocated to a **collection** variable. Use the DEALLOCATE COLLECTION statement to deallocate the memory.

The following code fragment declares the **a_set** host variable as a typed collection, allocates memory for this variable, then deallocates memory for this variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;
.
.
.
EXEC SQL allocate collection :a_set;
.
.
.
EXEC SQL deallocate collection :a_set;
```

The ALLOCATE COLLECTION statement allocates memory for the **collection** variable and the collection data.

Note: When DEALLOCATE COLLECTION fails because a cursor on the collection is still open, an error message will be returned. Prior to this, the error is not trapped.

For syntax information for the ALLOCATE COLLECTION and DEALLOCATE COLLECTION statements, see their descriptions in the *IBM Informix Guide to SQL: Syntax*.

Operating on a Collection Variable

IBM Informix supports access to a collection column as a whole through the SELECT, UPDATE, INSERT, and DELETE statements. For example, the SELECT statement can retrieve all elements of a collection, and the UPDATE statement can update all elements in a collection to a single value. (For more information, see "Operating on a Collection Column" on page 9-28.)

Tip: IBM Informix can only access the *contents* of collection columns directly with the IN predicate in the WHERE clause of a SELECT statement and this IN predicate works only with simple collections (collections whose element types are not complex types).

The SELECT, INSERT, UPDATE, and DELETE statements cannot access elements of a collection column in a table. To access elements in a collection column, an Informix ESQL/C application constructs a subtable, called a *collection-derived table,* in the collection host variable. From collection-derived table, the Informix ESQL/C application to access the elements of the collection variable as rows of a table.

This section discusses the following topics on how to use a collection-derived table in an Informix ESQL/C application to access a collection column:

- Using the collection-derived table clause in SQL statements to access a **collection** host variable
- Initializing a **collection** host variable with a collection column
- Inserting elements into a **collection** host variable
- Selecting elements from a **collection** host variable
- Updating elements in a **collection** host variable
- Specifying element values for a **collection** host variable
- Deleting elements from a **collection** host variable
- Accessing a nested collection with **collection** host variables

Using the Collection-Derived Table Clause on Collections

The collection-derived table clause allows you to specify a **collection** host variable as a table name. This clause has the following syntax:

```
TABLE(:coll_var)
```

In this example, *coll_var* is a **collection** host variable. It can be either a typed or untyped **collection** host variable, but it must be declared and have memory allocated in the Informix ESQL/C application before it appears in a collection-derived table clause.

For more information about the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the *IBM Informix Guide to SQL: Syntax*.

Accessing a Collection Variable: In SQL statements, the Informix ESQL/C application specifies a collection-derived table in place of a table name to perform the following operations on the **collection** host variable:

- You can *insert* an element into the **collection** host variable with the collection-derived table clause after the INTO keyword of an INSERT, or with the PUT statement.

  For more information, see "Inserting into a Collection Variable" on page 9-13.

- You can *select* an element from a **collection** host variable with the collection-derived table clause in the FROM clause of the SELECT statement.

  For more information, see "Selecting from a Collection Variable" on page 9-17.

- You can *update* all or some elements in the **collection** host variable with the collection-derived table clause (instead of a table name) after the UPDATE keyword in an UPDATE statement.

  For more information, see "Updating a Collection Variable" on page 9-20.

- You can *delete* all or some elements from the **collection** host variable with the collection-derived table clause after the FROM keyword in the DELETE statement.

  For more information, see "Deleting Elements from a Collection Variable" on page 9-24.

Tip: If you only need to insert or update a collection column with literal values, you do not need to use a **collection** host variable. Instead, you can explicitly list the literal-collection value in either the INTO clause of the INSERT statement or the SET clause of the UPDATE statement.

For more information, see "Inserting into and Updating a Collection Column" on page 9-28.

Once the **collection** host variable contains valid elements, you update the collection column with the contents of the host variable. For more information, see "Operating on a Collection Column" on page 9-28. For more information about the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the *IBM Informix Guide to SQL: Syntax*.

Distinguishing Between Columns and Collection Variables: When you use the collection-derived table clause with a collection host variable in an SQL statement (such as SELECT, INSERT, or UPDATE), the statement is not sent to the database server for processing. Instead, Informix ESQL/C processes the statement. Consequently, some of the syntax checking that the database server performs is not done on SQL statements that include the collection-derived table clause.

In particular, the Informix ESQL/C preprocessor cannot distinguish between column names and host variables. Therefore, when you use the collection-derived table clause with an UPDATE or INSERT statement, you must use valid host-variable syntax in:

- The SET clause of an UPDATE statement
- The VALUES clause of an INSERT statement

For more information, see "Distinguishing Between Columns and Row Variables" on page 9-37.

Initializing a Collection Variable

You *must* always initialize an *untyped* **collection** variable by selecting a collection column into it. You must execute a SELECT statement, regardless of the operation you wish to perform on the untyped **collection** variable.

> **Important:** Selecting the collection column into the untyped **collection** variable provides Informix ESQL/C with a description of the collection declaration.

For more information, see "The Untyped Collection Variable" on page 9-7.

You can initialize a **collection** variable by selecting a collection column into the **collection** variable, constructing the SELECT statement as follows:

- Specify the name of the collection column in the select list.
- Specify the **collection** host variable in the INTO clause.
- Specify the table or view name (not the collection-derived table clause) in the FROM clause.

You can initialize a *typed* collection variable by executing an INSERT statement that uses the collection derived table syntax. You do not need to initialize a typed collection variable before an INSERT or UPDATE because Informix ESQL/C has a description of the collection variable.

Suppose, for example, that you create the **tab_list** and **tab_set** tables with the statements in Figure 9-2.

```
EXEC SQL create table tab_list
    (list_col list(smallint not null));
EXEC SQL create table tab_set
(
    id_col integer,
    set_col set(integer not null)
);
```

Figure 9-2. Sample Tables with Collection Columns

The following code fragment accesses the **set_col** column with a typed **collection** host variable called **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL select set_col into :a_set from tab_set
    where id_col = 1234;
```

When you use a typed **collection** host variable, the description of the collection column (the collection type *and* the element type) should be compatible with the corresponding description of the typed **collection** host variable. If the data types don't match, the database server will do a cast if it can. The SELECT statement in the preceding code fragment successfully retrieves the **set_col** column because the **a_set** host variable has the same collection type (SET) and element type (INTEGER) as the **set_col** column.

The following SELECT statement *succeeds* because the database server casts **list_col** column to a set in **a_set** host variable and discards any duplicates:

```
/* This SELECT generates an error */
EXEC SQL select list_col into :a_set from tab_list;
```

You can select any type of collection into an untyped **collection** host variable. The following code fragment uses an untyped **collection** host variable to access the **list_col** and **set_col** columns that Figure 9-2 on page 9-12 defines:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection a_collection;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_collection;
EXEC SQL select set_col into :a_collection
   from tab_set
   where id_col = 1234;
⋮

EXEC SQL select list_col into :a_collection
   from tab_list
   where list{6} in (list_col);
```

Both SELECT statements in this code fragment can successfully retrieve collection columns into the **a_collection** host variable.

Once you have initialized the **collection** host variable, you can use the collection-derived table clause to select, update, or delete existing elements in the collection or to insert additional elements into the collection. For more information, see the following sections.

Inserting into a Collection Variable

To insert one or more elements into a **collection** variable, use the INSERT statement with the collection-derived table clause after the INTO keyword. The collection- derived table clause identifies the **collection** variable in which to insert the elements. Associate the INSERT statement and the collection-derived table clause with a cursor to insert more than one element into a collection variable.

Important: You cannot use expressions in the VALUES clause and you cannot use a WHERE clause.

For information about the types of values you can insert into a **collection** variable, see "Specifying Element Values" on page 9-22.

Inserting One Element: The INSERT statement and the collection-derived table clause allow you to insert *one* element into a collection. Informix ESQL/C inserts the values that the VALUES clause specifies into the **collection** variable that the collection-derived table clause specifies.

Tip: When you insert elements into a client-side **collection** variable, you cannot specify a SELECT, an EXECUTE FUNCTION, or an EXECUTE PROCEDURE statement in the VALUES clause of the INSERT.

Inserting into SET and MULTISET Collections: For SET and MULTISET collections, the position of the new element is undefined, because the elements of these collections do not have ordered positions. Suppose the table **readings** has the following declaration:

```
CREATE TABLE readings
(
   dataset_id      INT8,
   time_dataset MULTISET(INT8 NOT NULL)
);
```

To access the **time_dataset** column, the typed Informix ESQL/C host variable **time_vals** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(int8 not null) time_vals;
   ifx_int8_t an_int8;
EXEC SQL END DECLARE SECTION;
```

The following INSERT statement adds a new MULTISET element of 1,423,231 to **time_vals**:

```
EXEC SQL allocate collection :time_vals;
EXEC SQL select time_dataset into :time_vals
   from readings
   where dataset_id = 1356;
ifx_int8cvint(1423231, &an_int8);
EXEC SQL insert into table(:time_vals) values (:an_int8);
```

For more information about the **ifx_int8cvint()** function and the INT8 data type, see Chapter 5, "Working with numeric data types," on page 5-1.

Inserting into LIST Collections: LIST collections have elements that have ordered positions. If the collection is of type LIST, you can use the AT clause of the INSERT statement to specify the position in the list at which you want to add the new element. Suppose the table **rankings** has the following declaration:

```
CREATE TABLE rankings
(
   item_id             INT8,
   item_rankings            LIST(INTEGER NOT NULL)
);
```

To access the **item_rankings** column, the typed Informix ESQL/C host variable **rankings** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection list(integer not null) rankings;
   int an_int;
EXEC SQL END DECLARE SECTION;
```

The following INSERT statement adds a new list element of 9 as the new third element of **rankings**:

```
EXEC SQL allocate collection :rankings;
EXEC SQL select rank_col into :rankings from results;
an_int = 9;
EXEC SQL insert at 3 into table(:rankings) values (:an_int);
```

Suppose that before this insert, **rankings** contained the elements {1,8,4,5,2}. After this insert, this variable contains the elements {1,8,9,4,5,2}.

If you do not specify the AT clause, INSERT adds new elements at the end of a LIST collection. For more information about the AT clause, see the description of the INSERT statement in the *IBM Informix Guide to SQL: Syntax*.

Inserting More Than One Element: An insert cursor that includes an INSERT statement with the collection-derived table clause allows you to insert many elements into a **collection** variable. To insert elements, follow these steps:

1. Create a client **collection** variable in your Informix ESQL/C program.

   For more information, see "Declaring Collection Variables" on page 9-4 and "Managing Memory for Collections" on page 9-9.

2. Declare the insert cursor for the **collection** variable with the DECLARE statement and open the cursor with the OPEN statement.

3. Put the element(s) into the **collection** variable with the PUT statement and the FROM clause.
4. Close the insert cursor with the CLOSE statement, and if you no longer need the cursor, free it with the FREE statement.
5. Once the **collection** variable contains all the elements, you then use the UPDATE statement or the INSERT statement on a table name to save the contents of the **collection** variable in a collection column (SET, MULTISET, or LIST).

   For more information, see "Operating on a Collection Column" on page 9-28.

Tip: Instead of an insert cursor, you can use an INSERT statement to insert elements one at a time into a **collection** variable. However, an insert cursor is more efficient for large insertions.

For more information, see "Inserting One Element" on page 9-13.

The following sections provide information about how to declare an insert cursor for a collection variable, put elements into this cursor, and save the insert cursor into the collection variable. For sample code that inserts several elements into a collection variable, see Figure 9-3 on page 9-17.

Declaring an Insert Cursor for a Collection Variable: An insert cursor allows you to insert one or more elements in the collection. To declare an insert cursor for a **collection** variable, include the collection-derived table clause in the INSERT statement that you associate with the cursor. The insert cursor for a collection variable has the following restrictions:

- It must be a sequential cursor; the DECLARE statement cannot specify the SCROLL keyword.
- It cannot be a hold cursor; the DECLARE statement cannot specify the WITH HOLD cursor characteristic.

If you need to use input parameters, you must prepare the INSERT statement and specify the prepared statement identifier in the DECLARE statement.

You can use input parameters to specify the values in the VALUES clause of the INSERT statement.

The following DECLARE statement declares the **list_curs** insert cursor for the **a_list** variable:

```
EXEC SQL prepare ins_stmt from
   'insert into table values';
EXEC SQL declare list_curs cursor for ins_stmt;
EXEC SQL open list_curs using :a_list;
```

You can then use the PUT statement to specify the values to insert. For a code fragment that includes this statement, see Figure 9-3 on page 9-17.

Important: Whenever you use a question mark (?) in a PREPARE statement for a collection host variable in a collection-derived table, if you execute a DESCRIBE statement you must execute it after an OPEN statement. Until the OPEN statement, Informix ESQL/C does not know what the collection row looks like.

- The name of the collection variable in the collection-derived table clause

  The following DECLARE statement declares the **list_curs2** insert cursor for the **a_list** variable:

```
EXEC SQL prepare ins_stmt2 from
   'insert into table values';
EXEC SQL declare list_curs2 cursor for ins_stmt2;
EXEC SQL open list_curs2 using :a_list;
while (1)
   {
   EXEC SQL put list_curs2 from :an_element;
   .
   .
   .

   }
```

The USING clause of the OPEN statement specifies the name of the **collection** variable. You can then use the PUT statement to specify the values to insert. For information about how to use the PUT statement, see "Putting Elements into the Insert Cursor."

After you declare the insert cursor, you can open it with the OPEN statement. You can insert elements into the **collection** variable once the associated insert cursor is open.

Putting Elements into the Insert Cursor: To put elements, one at a time, into the insert cursor, use the PUT statement and the FROM clause. The PUT statement identifies the insert cursor that is associated with the **collection** variable. The FROM clause identifies the element value to be inserted into the cursor. The data type of any host variable in the FROM clause must be compatible with the element type of the collection.

To indicate that the collection element is to be provided later by the FROM clause of the PUT statement, use an input parameter in the VALUES clause of the INSERT statement. You can use the PUT statement with an insert cursor following either a static DECLARE statement or the PREPARE statement. The following example uses a PUT following a static DECLARE statement.

```
EXEC SQL DECLARE list_curs cursor FOR INSERT INTO table
   (:alist);
EXEC SQL open list_curs;
EXEC SQL PUT list_curs from :asmint;
```

No input parameters can appear in the DECLARE statement.

Figure 9-3 contains a code fragment that demonstrates how to insert elements into the **collection** variable **a_list** and then to update the **list_col** column of the **tab_list** table (which Figure 9-2 on page 9-12 defines) with this new collection.

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection list(smallint not null) a_list;
   int a_smint;
EXEC SQL END DECLARE SECTION;
   :
   :

EXEC SQL allocate collection :a_list;

/* Step 1: declare the insert cursor on the collection variable */
EXEC SQL prepare ins_stmt from
   'insert into table values';
EXEC SQL declare list_curs cursor for ins_stmt;
EXEC SQL open list_curs using :a_list;

/* Step 2: put the LIST elements into the insert cursor */
for (a_smint=0; a_smint<10; a_smint++)
{
   EXEC SQL put list_curs from :a_smint;
};
/* Step 3: save the insert cursor into the collection variable
EXEC SQL close list_curs;

/* Step 4: save the collection variable into the LIST column */
EXEC SQL insert into tab_list values (:a_list);

/* Step 5: clean up */
EXEC SQL deallocate collection :a_list;
EXEC SQL free ins_stmt;
EXEC SQL free list_curs;
```

Figure 9-3. Insertion of Many Elements Into a Collection Host Variable

In Figure 9-3, the first statement that accesses the **a_list** variable is the OPEN
statement. Therefore, at this point in the code, Informix ESQL/C must be able to
determine the data type of the **a_list** variable. Because the **a_list** host variable is a
typed **collection** variable, Informix ESQL/C can determine the data type from the
variable declaration. However, if **a_list** was declared an *untyped* **collection** variable,
you would need a SELECT statement before the DECLARE statement executes to
return the definition of the associated collection column.

Informix ESQL/C automatically saves the contents of the insert cursor into the
collection variable when you put them into the insert cursor with the PUT
statement.

Freeing Cursor Resources: The CLOSE statement explicitly frees resources assigned
to the insert cursor. However, the cursor ID still exists, so you can reopen the
cursor with the OPEN statement. The FREE statement explicitly frees the cursor
ID. To reuse the cursor, you must redeclare it with the DECLARE statement.

The FLUSH statement does not have an effect on an insert cursor that is associated
with a collection variable. For the syntax of the CLOSE statement, see the *IBM
Informix Guide to SQL: Syntax*.

Selecting from a Collection Variable

The SELECT statement with the collection-derived table clause allows you to select
elements from a **collection** variable. The collection-derived table clause identifies
the **collection** variable from which to select the elements. The SELECT statement
on a *client* **collection** variable (one that has the collection-derived table clause) has
the following restrictions:

- The select list of the SELECT cannot contain expressions.
- The select list must be an asterisk (*).
- Column names in the select list must be simple column names.

These columns cannot use the *database@server:table.column* syntax.
- The following SELECT clauses and options are not allowed: GROUP BY, HAVING, INTO TEMP, ORDER BY, WHERE, WITH REOPTIMIZATION.
- The FROM clause has no provisions to do a join.

The SELECT statement and the collection-derived table clause allow you to perform the following operations on a **collection** variable:
- Select *one* element from the collection

  Use the SELECT statement with the collection-derived table clause.
- Select *one* row element from the collection.

  Use the SELECT statement with the collection-derived table clause and a row variable.
- Select *one or more* elements into the collection

  Associate the SELECT statement and the collection-derived table clause with a cursor to declare a select cursor for the **collection** variable.

Selecting One Element: The SELECT statement and the collection-derived table clause allow you to select *one* element into a collection. The INTO clause identifies the variable in which to store the element value that is selected from the **collection** variable. The data type of the host variable in the INTO clause must be compatible with the element type of the collection.

The following code fragment selects only one element from the **set_col** column (see Figure 9-2 on page 9-12) with a typed **collection** host variable called **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
   int an_element, set_size;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL select set_col, cardinality(set_col)
   into :a_set, :set_size from tab_set
   where id_col = 3;
if (set_size == 1)
   EXEC SQL select * into :an_element from table(:a_set);
```

Important: Use this form of the SELECT statement when you are sure that the SELECT returns only one element. Informix ESQL/C returns an error if the SELECT returns more than one element. If you do not know the number of elements in the set or if you know that the set contains more than one element, use a select cursor to access the elements.

For more information about how to use a select cursor, see "Selecting More Than One Element" on page 9-19.

If the element of the collection is itself a complex type (collection or row type), the collection is a *nested collection*. For information about how to use a cursor to select elements from a nested collection, see "Selecting Values from a Nested Collection" on page 9-26. The following section describes how to use a **row** variable to select a row element from a collection.

Selecting One Row Element: You can select an entire row element from a collection into a **row** type host variable. The INTO clause identifies a **row** variable in which to store the row element that is selected from the **collection** variable.

The following code fragment selects one row from the **set_col** column into the row type host variable **a_row**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(row(a integer) not null) a_set;
   row (a integer) a_row;
EXEC SQL END DECLARE SECTION;

EXEC SQL select set_col into :a_set from tab1
   where id_col = 17;
EXEC SQL select * into :a_row from table(:a_set);
```

Selecting More Than One Element: A select cursor that includes a SELECT statement with the collection-derived table clause allows you to select many elements from a **collection** variable. To select elements, follow these steps:

1. Create a client **collection** variable in your Informix ESQL/C program.

   For more information, see "Declaring Collection Variables" on page 9-4 and "Managing Memory for Collections" on page 9-9.

2. Declare the select cursor for the **collection** variable with the DECLARE statement and open this cursor with the OPEN statement.

3. Fetch the element(s) from the **collection** variable with the FETCH statement and the INTO clause.

4. If necessary, perform any updates or deletes on the fetched data and save the modified **collection** variable in the collection column.

   For more information, see "Operating on a Collection Column" on page 9-28.

5. Close the select cursor with the CLOSE statement, and if you no longer need the cursor, free it with the FREE statement.

Declaring a Select Cursor for a Collection Variable: To declare a select cursor for a **collection** variable, include the collection-derived table clause with the SELECT statement that you associate with the cursor. The DECLARE for this select cursor has the following restrictions:

- The select cursor is an update cursor.

  The DECLARE statement cannot include the FOR READ ONLY clause that specifies the read-only cursor mode.

- The select cursor must be a sequential cursor.

  The DECLARE statement cannot specify the SCROLL or WITH HOLD cursor characteristics.

When you declare a select cursor for a **collection** variable, the collection-derived table clause of the SELECT statement *must* contain the name of the **collection** variable. For example, the following DECLARE statement declares a select cursor for the **collection** variable, **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;
.
.
.
EXEC SQL declare set_curs cursor for
   select * from table(:a_set);
```

To select the element(s) from the **collection** variable, use the FETCH statement with the INTO clause. For more information, see "Fetching Elements from the Select Cursor" on page 9-20.

If you want to modify the elements of the **collection** variable, declare the select cursor as an update cursor with the FOR UPDATE keywords. You can then use the WHERE CURRENT OF clause of the DELETE and UPDATE statements to delete or update elements of the collection. For more information, see "Deleting One Element" on page 9-24 and "Updating One Element" on page 9-21.

Fetching Elements from the Select Cursor: To fetch elements, one at a time, from a **collection** variable, use the FETCH statement and the INTO clause. The FETCH statement identifies the select cursor that is associated with the **collection** variable. The INTO clause identifies the host variable for the element value that is fetched from the **collection** variable. The data type of the host variable in the INTO clause must be compatible with the element type of the collection.

Figure 9-4 contains a code fragment that selects all elements from the **set_col** column (see Figure 9-2 on page 9-12) into the typed **collection** host variable called **a_set** then fetches these elements, one at a time, from the **a_set collection** variable.

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
   int an_element, set_size;
EXEC SQL END DECLARE SECTION;
int an_int
:

EXEC SQL allocate collection :a_set;
EXEC SQL select set_col, cardinality(set_col)
   into :a_set from tab_set
   from tab_set where id_col = 3;

/* Step 1: declare the select cursor on the host variable */
EXEC SQL declare set_curs cursor for
   select * from table(:a_set);
EXEC SQL open set_curs;

/* Step 2: fetch the SET elements from the select cursor */
for (an_int=0; an_int<set_size; an_int++)
{
   EXEC SQL fetch set_curs into :an_element;

:

};
EXEC SQL close set_curs;

/* Step 3: update the SET column with the host variable */
EXEC SQL update tab_list SET set_col = :a_set
   where id_col = 3

EXEC SQL deallocate collection :a_set;
EXEC SQL free set_curs;
```

Figure 9-4. Selection of Many Elements From a Collection Host Variable

Updating a Collection Variable

Once you have initialized a **collection** host variable with a collection column (see 9-12), you can use the UPDATE statement with the collection-derived table clause to update the elements in the collection. The collection-derived table clause identifies the **collection** variable whose elements are to be updated. The UPDATE statement and the collection-derived table clause allow you to perform the following operations on a **collection** variable:

• Update all elements in the collection to the same value.

  Use the UPDATE statement (without the WHERE CURRENT OF clause) and specify a derived column name in the SET clause.

- Update a particular element in the collection.

  You must declare an update cursor for the **collection** variable and use UPDATE with the WHERE CURRENT OF clause.

Neither form of the UPDATE statement can include a WHERE clause. For information about the types of values you can specify when you update a **collection** variable, see "Specifying Element Values" on page 9-22.

Updating All Elements: You cannot include a WHERE clause on an UPDATE statement with a collection-derived table clause. Therefore, an UPDATE statement on a **collection** variable sets *all* elements in the collection to the value you specify in the SET clause. No update cursor is required to update all elements of a collection.

For example, the following UPDATE changes all elements in the **a_list** Informix ESQL/C **collection** variable to a value of 16:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection list(smallint not null) a_list;
   int an_int;
EXEC SQL END DECLARE SECTION;
⋮

EXEC SQL update table(:a_list) (list_elmt)
      set list_elmt = 16;
```

In this example, the derived column **list_elmt** provides an alias to identify an element of the collection in the SET clause.

Updating One Element: To update a particular element in a collection, declare an update cursor for the **collection** host variable. An update cursor for a **collection** variable is a select cursor that was declared with the FOR UPDATE keywords. The update cursor allows you to sequentially scroll through the elements of the collection and update the current element with the UPDATE...WHERE CURRENT OF statement.

To update elements, follow these steps:

1. Create a client **collection** variable in your IBM Informix ESQL/C program.

   For more information, see "Declaring Collection Variables" on page 9-4 and "Managing Memory for Collections" on page 9-9.

2. Declare the update cursor for the **collection** variable with the DECLARE statement and the FOR UPDATE clause; open this cursor with the OPEN statement.

   By default, a select cursor on a **collection** variable supports updates. For more information about how to declare a select cursor, see 9-19.

3. Fetch the element(s) from the **collection** variable with the FETCH statement and the INTO clause.

   For more information, see "Selecting More Than One Element" on page 9-19.

4. Update the fetched data with the UPDATE statement and the WHERE CURRENT OF clause.

5. Save the modified **collection** variable in the collection column.

   For more information, see "Operating on a Collection Column" on page 9-28.

6. Close the update cursor with the CLOSE statement, and if you no longer need the cursor, free it with the FREE statement.

The application must position the update cursor on the element to be updated and then use UPDATE...WHERE CURRENT OF to update this value.

The Informix ESQL/C program in Figure 9-5 uses an update cursor to update an element in the **collection** variable, **a_set**, and then to update the **set_col** column of the **tab_set** table (see Figure 9-2 on page 9-12).

```
EXEC SQL BEGIN DECLARE SECTION;
    int an_element;
    client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL select set_col into :a_set from tab_set
    where id_col = 6;

EXEC SQL declare set_curs cursor for
    select * from table(:a_set)
    for update;

EXEC SQL open set_curs;
while (SQLCODE != SQLNOTFOUND)
    {
    EXEC SQL fetch set_curs into :an_element;
    if (an_element = 4)
        {
        EXEC SQL update table(:a_set)(x)
            set x = 10
            where current of set_curs;
        break;
        }
    }

EXEC SQL close set_curs;

EXEC SQL update tab_set set set_col = :a_set
    where id_col = 6;

EXEC SQL deallocate collection :a_set;
EXEC SQL free set_curs;
```

Figure 9-5. Updating One Element in a Collection Host Variable

Specifying Element Values

You can specify any of the following values as elements in a **collection** variable:

- A literal value

  You can also specify literal values directly for a collection column without first using a **collection** variable. For more information, see "Inserting into and Updating a Collection Column" on page 9-28.

- An Informix ESQL/C host variable

  The host variable must contain a value whose data type is compatible with the element type of the collection.

You cannot include complex expressions directly to specify values.

For information about how to insert elements into a **collection** variable, see "Inserting into a Collection Variable" on page 9-13. For information about how to update elements in a **collection** variable, see "Updating a Collection Variable" on page 9-20. The following sections describe the values you can assign to an element in a **collection** variable.

Literal Values as Elements: You can use a literal value to specify an element of a **collection** variable. The literal values must have a data type that is compatible with the element type of the collection. For example, the following INSERT statement inserts a literal integer into a SET(INTEGER NOT NULL) host variable called **a_set**:

```
EXEC SQL insert into table(:a_set) values (6);
```

The following UPDATE statement uses a derived column name (**an_element**) to update all elements of the **a_set** collection variable with the literal value of 19:

```
EXEC SQL update table(:a_set) (an_element)
   set an_element = 19;
```

The following INSERT statement inserts a quoted string into a LIST(CHAR(5)) host variable called **a_set2**:

```
EXEC SQL insert into table(:a_set2) values ('abcde');
```

The following INSERT statement inserts a literal collection into a SET(LIST(INTEGER NOT NULL) host variable called **nested_coll**:

```
EXEC SQL insert into table(:nested_coll)
   values (list{1,2,3});
```

Tip: The syntax of a literal collection for a **collection** variable is different from the syntax of a literal collection for a collection column. A **collection** variable does not need to be a quoted string.

For more information about literal-collection values for a collection column, see "Inserting into and Updating a Collection Column" on page 9-28.

The following UPDATE statement updates the **nested_coll** collection variable with a new literal collection value:

```
EXEC SQL update table(:nested_coll) (a_list)
   set a_list = list{1,2,3};
```

Tip: If you only need to insert or update the collection column with literal values, you do not need to use a **collection** host variable. Instead, you can explicitly list the literal values as a literal collection in either the INTO clause of the INSERT statement or the SET clause of the UPDATE statement.

For more information, see "Inserting into and Updating a Collection Column" on page 9-28.

ESQL/C Host Variables as Elements: You can use an Informix ESQL/C host variable to specify an element of a **collection** variable. The host variable must be declared with a data type that is compatible with the element type of the collection and must contain a value that is also compatible. For example, the following INSERT statement uses a host variable to insert a single value into the same **a_set** variable as in the preceding example:

```
an_int = 6;
EXEC SQL insert into table(:a_set) values (:an_int);
```

To insert multiple values into a **collection** variable, you can use an INSERT statement for each value or you can declare an insert cursor and use the PUT statement. For more information, see "Inserting More Than One Element" on page 9-14.

The following UPDATE statement uses a host variable to update all elements in the **a_set** collection to a value of 4:

```
an_int = 4;
EXEC SQL update table(:a_set) (an_element)
   set an_element = :an_int;
```

To update multiple values into a **collection** variable, you can declare an update cursor and use the WHERE CURRENT OF clause of the UPDATE statement. For more information, see "Updating One Element" on page 9-21.

Deleting Elements from a Collection Variable

Once you have initialized a **collection** host variable with a collection column (see 9-12), you can use the DELETE statement and the collection-derived table clause to delete an element of a **collection** variable. The collection-derived table clause identifies the **collection** variable in which to delete the elements.

The DELETE statement and the collection-derived table clause allow you to perform the following operations on a **collection** variable:

- Delete all elements in the collection.

  Use the DELETE statement (without the WHERE CURRENT OF clause).

- Delete a particular element in the collection.

  You must declare an update cursor for the **collection** variable and use DELETE with the WHERE CURRENT OF clause.

Neither form of the DELETE statement can include a WHERE clause.

Deleting All Elements: You cannot include a WHERE clause on a DELETE statement with a collection-derived table clause. Therefore, a DELETE statement on a **collection** variable deletes *all* elements from the collection. No update cursor is required to delete all elements of a collection.

For example, the following DELETE removes all elements in the **a_list** Informix ESQL/C **collection** variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection list(smallint not null) a_list;
EXEC SQL END DECLARE SECTION;
:

EXEC SQL delete from table(:a_list);
```

Deleting One Element: To delete a particular element in a collection, declare an update cursor for the **collection** host variable. An update cursor for a **collection** variable is a select cursor that was declared with the FOR UPDATE keywords. The update cursor allows you to sequentially scroll through the elements of the collection and delete the current element with the DELETE...WHERE CURRENT OF statement.

To delete particular elements, follow the same steps for how to update particular elements (see "Updating One Element" on page 9-21). In these steps, you replace the use of the UPDATE...WHERE CURRENT OF statement with the DELETE...WHERE CURRENT OF statement.

The application must position the update cursor on the element to be deleted and then use DELETE...WHERE CURRENT OF to delete this value. The following Informix ESQL/C code fragment uses an update cursor and a DELETE statement

with a WHERE CURRENT OF clause to delete the element from the **set_col**
column of **tab_set** (see Figure 9-2 on page 9-12).

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
   int an_int, set_size;
EXEC SQL END DECLARE SECTION;
:

EXEC SQL allocate collection :a_set;
EXEC SQL select set_col, cardinality(set_col)
   into :a_set, :set_size
   from tab_set
   where id_col = 6;

EXEC SQL declare set_curs cursor for
   select * from table(:a_set)
   for update;

EXEC SQL open set_curs;
while (i < set_size)
{
   EXEC SQL fetch set_curs into :an_int;
   if (an_int == 4)
   {
      EXEC SQL delete from table(:a_set)
         where current of set_curs;
      break;
   }
   i++;
}
EXEC SQL close set_curs;
EXEC SQL free set_curs;

EXEC SQL update tab_set set set_col = :a_set
   where id_col = 6;

EXEC SQL deallocate collection :a_set;
```

Suppose that in the row with an **id_col** value of 6, the **set_col** column contains the
values {1,8,4,5,2} before this code fragment executes. After the DELETE...WHERE
CURRENT OF statement, this **collection** variable contains the elements {1,8,5,2}.
The UPDATE statement at the end of this code fragment saves the modified
collection into the **set_col** column of the database. Without this UPDATE statement,
the collection column never has element 4 deleted.

Accessing a Nested Collection

IBM Informix supports nested collections as a column type. A *nested collection* is a
collection column whose element type is another collection. For example, the code
fragment in Figure 9-6 creates the **tab_setlist** table whose column is a nested
collection.

```
EXEC SQL create table tab_setlist
   ( setlist_col set(list(integer not null));
```

Figure 9-6. Sample Column with Nested Collection

The **setlist_col** column is a set, each element of which is a list. This nested
collection resembles a two-dimensional array with a *y*-axis of set elements and an
x-axis of list elements.

Selecting Values from a Nested Collection: To select values from a nested collection, you must declare a **collection** variable and a select cursor for each level of collection. The following code fragment uses the nested **collection** variable, **nested_coll** and the **collection** variable **list_coll** to select the lowest-level elements in the nested-collection column, **setlist_col**.

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(list(integer not null) not null) nested_coll;
   client collection list(integer not null) list_coll;
   int an_element;
EXEC SQL END DECLARE SECTION;
int num_elements = 1;
int an_int;
int keep_fetching = 1;
   .
   .

EXEC SQL allocate collection :nested_coll;
EXEC SQL allocate collection :list_coll;

/* Step 1: declare the select cursor on the SET collection variable */
EXEC SQL declare set_curs2 cursor for
   select * from table(:nested_coll);

/* Step 2: declare the select cursor on the LIST collection variable */
EXEC SQL declare list_curs2 cursor for
   select * from table(:list_coll);

/* Step 3: open the SET cursor */
EXEC SQL open set_curs2;

while (keep_fetching)
   {

/* Step 4: fetch the SET elements into the SET insert cursor */
   EXEC SQL fetch set_curs2 into :list_coll;

/* Open the LIST cursor */
   EXEC SQL open list_curs2;

/* Step 5: put the LIST elements into the LIST insert cursor */
   for (an_int=0; an_int<10; an_int++)
      {
      EXEC SQL fetch list_curs2 into :an_element;
   .
   .


      };
   EXEC SQL close list_curs2;
   num_elements++;

   if (done_fetching(num_elements))
      {
      EXEC SQL close set_curs2;
      keep_fetching = 0;
      }
   };
EXEC SQL free set_curs2;
EXEC SQL free list_curs2;

EXEC SQL deallocate collection :nested_coll;
EXEC SQL deallocate collection :list_coll;:
```

Inserting Values into a Nested Collection: To insert literal values into a **collection** variable for a nested column, you specify the literal collection for the element type. You do not need to specify the constructor keyword for the actual collection type. The following typed **collection** host variable can access the **setlist_col** column of the **tab_setlist** table:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(list(integer not null) not null)
      nested_coll;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection nested_coll;
```

The following code fragment inserts literal values into the **nested_coll collection** variable and then updates the **setlist_col** column (which Figure 9-6 on page 9-25 defines):

```
EXEC SQL insert into table(:nested_coll)
   values (list{1,2,3,4});
EXEC SQL insert into tab_setlist values (:nested_coll);
```

To insert nonliteral values into a nested collection, you must declare a **collection** variable and an insert cursor for each level of collection. For example, the following code fragment uses the nested **collection** variable, **nested_coll**, to insert new elements into the nested-collection column, **setlist_col**.

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(list(integer not null) not null) nested_coll;
   client collection list(integer not null) list_coll;
   int an_element;
EXEC SQL END DECLARE SECTION;
int num_elements = 1;
int keep_adding = 1;
int an_int;
:

EXEC SQL allocate collection :nested_coll;
EXEC SQL allocate collection :list_coll;

/* Step 1: declare the insert cursor on the SET collection variable */
EXEC SQL declare set_curs cursor for
   insert into table(:nested_coll) values;

/* Step 2: declare the insert cursor on the LIST collection variable */
EXEC SQL declare list_curs cursor for
   insert into table(:list_coll) values;

/* Step 3: open the SET cursor */
EXEC SQL open set_curs;

while (keep_adding)
   {

/* Step 4: open the LIST cursor */
    SQL open list_curs;

/* Step 5: put the LIST elements into the LIST insert cursor */
   for (an_int=0; an_int<10; an_int++)
      {
      an_element = an_int * num_elements;
      EXEC SQL put list_curs from :an_element;

:
```

```
            };
     EXEC SQL close list_curs;
     num_elements++;

/* Step 6: put the SET elements into the SET insert cursor */
     EXEC SQL put set_curs from :list_coll;
     if (done_adding(num_elements)
         {
         EXEC SQL close set_curs;
         keep_adding = 0;
         }
     };
EXEC SQL free set_curs;
EXEC SQL free list_curs;

/* Step 7: insert the nested SET column with the host variable */
EXEC SQL insert into tab_setlist values (:nested_coll);

EXEC SQL deallocate collection :nested_coll;
EXEC SQL deallocate collection :list_coll;
```

Operating on a Collection Column

The **collection** variable stores the elements of the collection. However, it has no intrinsic connection with a database column. You must use an INSERT or UPDATE statement to explicitly save the contents of the collection variable into the collection column.

You can use the SELECT, UPDATE, INSERT, and DELETE statements to access a collection column (SET, MULTISET, or LIST), as follows:

- The SELECT statement fetches all elements from a collection column.
- The INSERT statement inserts a new collection into a collection column.

  Use the INSERT statement on a table or view name and specify the **collection** variable in the VALUES clause.

  Figure 9-3 on page 9-17 shows an INSERT statement that saves the contents of a collection variable in a collection column.

- The UPDATE statement updates the entire collection in a collection column with new values.

  Use an UPDATE statement on a table or view name and specify the **collection** variable in the SET clause.

  Figure 9-5 on page 9-22 shows an UPDATE statement that saves the contents of a collection variable in a collection column.

The following sections summarize these SQL statements with collection columns. For more information about how to use these statements with collection columns, see the *IBM Informix Guide to SQL: Tutorial*.

Selecting from a Collection Column

To select *all* elements in a collection column, specify the collection column in the select list of the SELECT statement. If you put a **collection** host variable in the INTO clause of the SELECT statement, you can access these elements from an Informix ESQL/C application. For more information, see "Initializing a Collection Variable" on page 9-12. For an example that uses a collection variable to select and display the elements of a collection, see "The collect.ec Program" on page 14-21.

Inserting into and Updating a Collection Column

The INSERT and UPDATE statements support collection columns as follows:

- To *insert* a collection of elements into an empty collection column, specify the new elements in the VALUES clause of the INSERT statement.
- To *update* the entire collection in a collection column, specify the new elements in the SET clause of the UPDATE statement. The UPDATE statement must also specify a *derived column name* to create an identifier for the element. You then use this derived column name in the SET clause to identify where to assign the new element values.

In the VALUES clause of an INSERT statement or the SET clause of an UPDATE statement, the element values can be in any of the following formats:
- An Informix ESQL/C **collection** host variable
- A literal collection value

To represent literal values for a collection column, you specify a literal-collection value. You create a literal-collection value, introduce the value with the SET, MULTISET, or LIST keyword and provide the field values in a comma-separated list that is enclosed in braces. You surround the entire literal-collection value with quotes (double or single). The following INSERT statement inserts the literal collection of SET {7, 12, 59, 4} into the **set_col** column in the **tab_set** table (that Figure 9-2 on page 9-12 defines):

```
EXEC SQL insert into tab_set values
(
   5, 'set{7, 12, 59, 4}'
);
```

The UPDATE statement in Figure 9-7 overwrites the SET values that the previous INSERT added to the **tab_set** table.

```
EXEC SQL update tab_set
   set set_col = ("list{1,2,3,4}")
   where id_col = 5;
```

Figure 9-7. Updating a Collection Column

Important: If you omit the WHERE clause, the UPDATE statement in Figure 9-7 updates the **set_col** column in all rows of the **tab_set** table.

If any character value appears in this literal-collection value, it too must be enclosed in quotes; this condition creates nested quotes. For example, for column **col1** of type SET(CHAR(5), a literal value can be expressed as follows:

```
'SET{"abcde"}'
```

To specify nested quotes in an SQL statement in an Informix ESQL/C program, you must escape every double quote when it appears in a single-quote string. The following INSERT statement shows how to use escape characters for inner double quotes:

```
EXEC SQL insert into (col1) tab1
   values ('SET{\"abcde\"}');
```

When you embed a double-quoted string inside another double-quoted string, you do not need to escape the inner-most quotation marks, as the following INSERT statement shows:

```
EXEC SQL insert into tabx
   values (1, "set{""row(12345)""}");
```

For more information about the syntax of literal values for **collection** variables, see "Literal Values as Elements" on page 9-23. For more information about the syntax of literal-collection values for collection columns, see the Literal Collection segment in the *IBM Informix Guide to SQL: Syntax*.

If the collection or row type is nested, that is, if it contains another collection or row type as a member, the inner collection or row does not need to be enclosed in quotes. For example, for column **col2** whose data type is LIST(ROW(a INTEGER, b SMALLINT) NOT NULL), you can express the literal value as follows:

```
'LIST{ROW(80, 3)}'
```

Deleting an Entire Collection: To *delete* the entire collection in a collection column you can use the UPDATE statement to set the collection to empty. The UPDATE statement in the following example effectively deletes the entire collection in the **set_col** column of the **tab_set** table for the row in which **id_col** equals 5.

```
EXEC SQL create table tab_set
(
   id_col integer,
   set_col set(integer not null)
);
EXEC SQL update tab_set set set_col = set{}
   where id_col = 5;
```

The same UPDATE statement without the WHERE clause, as shown in the following example, would set the **set_col** column to empty for all rows in the **tab_set** table.

```
EXEC SQL update tab_set set set_col = set{};
```

Accessing Row Types

Informix ESQL/C supports the SQL row types with the Informix ESQL/C **row** type host variable. A *row type* is a complex data type that contains one or more members called fields. Each *field* has a name and a data type associated with it.

IBM Informix supports the following two kinds of row types:

- A *named* row type has a unique name that identifies to a group of fields.

  The named row type is a template for a row definition. You create a named row type with the CREATE ROW TYPE statement. You can then use a named row type as follows:

  – In a column definition of a CREATE TABLE statement to assign the data type for a column in the database

  – In the OF TYPE clause of the CREATE TABLE statement to create a typed table

- An *unnamed* row type uses the ROW constructor to define fields.

  You can use a particular unnamed row type as the data type of one column in the database. You create an unnamed row type with the ROW constructor in the column definition of a CREATE TABLE statement.

For more information about row types, see the CREATE ROW TYPE statement in the *IBM Informix Guide to SQL: Syntax* and the *IBM Informix Guide to SQL: Reference*.

To access a column in a table that has a row type as its data type, perform the following steps:

1. Declare a **row** host variable.

2. Allocate memory for the **row** host variable with the ALLOCATE ROW statement.
3. Perform any select or update operations on the **row** host variable.
4. Save the contents of the **row** host variable in the row-type column.

The following sections describe each of these steps in more detail.

Declaring Row Variables

To declare a **row** host variable, use the following syntax.

```
           (1)
►►──────row─────────────────────────────────────────────────────►
         └'─named row type─'┘    ┌─────────,◄───────┐
                              └(─▼─field name─field type─┘─)─┘


    ┌─────,◄─────┐
►──▼─variable name─┘──;──────────────────────────────────────────►◄
```
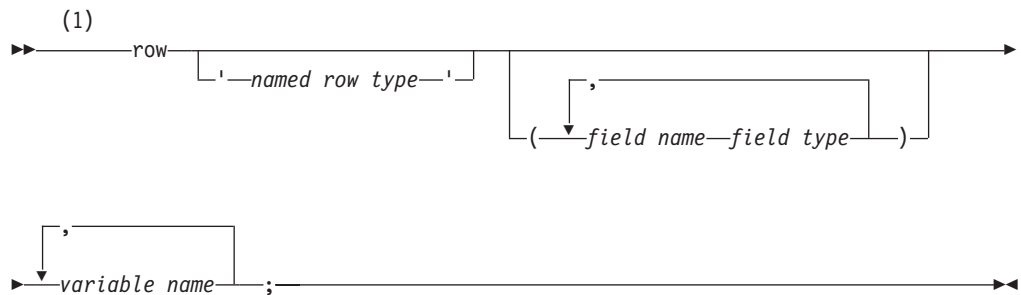
Notes:

1   Informix extension

Element	Purpose	Restrictions	SQL Syntax
field name	Name of a field in the **row** variable	Must match the corresponding field name in any associated row-type column.	Identifier segment in the *IBM Informix Guide to SQL: Syntax*
field type	Data type of the field name field in the **row** variable	Can be any data type except SERIAL, SERIAL8, BIGSERIAL, TEXT, or BYTE.	Data Type segment in the *IBM Informix Guide to SQL: Syntax*
named row type	Name of the named row type to assign to the **row** variable	Named row type must be defined in the database.	Identifier segment in the *IBM Informix Guide to SQL: Syntax*
variable name	Name of the ESQL/C variable to declare as a **row** variable		Name must conform to language-specific rules for variable names.

Typed and Untyped Row Variables

Informix ESQL/C supports the following two **row** variables:

- A typed **row** variable specifies the names and data types of the fields in the row.
- An untyped **row** variable specifies neither the field names nor the field types for the row.

Informix ESQL/C handles **row** variables as client-side **collection** variables.

The Typed Row Variable: A *typed* **row** variable specifies a field list, which contains the name and data type of each field in the row. Figure 9-8 shows declarations for three typed **row** variables.

```
EXEC SQL BEGIN DECLARE SECTION;
  row (circle_vals circle_t, circle_id integer) mycircle;
  row (a char(20),
       b set(integer not null),
       c decimal(10,2)) row2;
  row (x integer,
       y integer,
       length integer,
       width integer) myrect;
EXEC SQL END DECLARE SECTION;
```

Figure 9-8. Sample Typed Row Variables

Typed **row** variables can contain fields with the following data types:
- Any built-in data type (such as INTEGER, CHAR, BOOLEAN, and FLOAT) *except* BYTE, TEXT, SERIAL, or SERIAL8.
- Collection data types, such as SET and LIST
- Row types, named or unnamed
- Opaque data types

When you specify the type of a field in the **row** variable, use SQL data types, not Informix ESQL/C data types. For example, to declare a **row** variable with a field that holds small integers, use the SQL SMALLINT data type, not the Informix ESQL/C **int** data type. Similarly, to declare a field whose values are character strings, use the SQL syntax for a CHAR column, not the C syntax for **char** variables. For example, the following declaration of the **row_var** host variable contains a field of small integers and a character field:

```
row (
   smint_fld smallint,
   char_fld char(20)
   ) row_var;
```

Use a typed **row** variable when you know the exact data type of the row-type column that you store in the **row** variable. Match the declaration of a typed **row** variable exactly with the data type of the row-type column. You can use this **row** variable directly in SQL statements such as INSERT, DELETE, or UPDATE. You can also use it in the collection-derived table clause.

You can declare several **row** variables in a single declaration line. However, all variables must have the *same* field types, as the following declaration shows:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x integer, y integer) typed_row1, typed_row2;
EXEC SQL END DECLARE SECTION;
```

If you do not know the exact data type of the row-type column you want to access, use an untyped **row** variable.

The Untyped Row Variable: The definition of an *untyped* **row** variable specifies only the **row** keyword and a name. The following lines declare three untyped **row** variables:

```
EXEC SQL BEGIN DECLARE SECTION;
   row row1, row2;
   row rectangle1;
EXEC SQL END DECLARE SECTION;
```

The advantage of an untyped **row** host variable is that it provides more flexibility in row definition. For an untyped **row** variable, you do not have to know the

definition of the row-type column at compile time. Instead, you obtain, at runtime, a description of the row from a row-type column.

To obtain this description at runtime, execute a SELECT statement that retrieves the row-type column into the untyped **row** variable. When the database server executes the SELECT statement, it returns the data-type information for the row-type column (the types of the fields in the row) to the client application.

For example, suppose the **a_row** host variable is declared as an untyped **row** variable, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
    row a_row;
EXEC SQL END DECLARE SECTION;
```

The following code fragment uses a SELECT statement to initialize the **a_row** variable with data-type information *before* it uses the **row** variable in an UPDATE statement:

```
EXEC SQL allocate row :a_row;

/* obtain the data-type information */
EXEC SQL select row_col into :a_row from tab_row;

/* update row values in the untyped row variable */
EXEC SQL update table(:a_row) set fld1 = 3;
```

The field name **fld1**, which refers to a field of **:a_row**, comes from the definition of the row column in the **tab_row** table.

For more information about the ALLOCATE ROW statement, see "Managing Memory for Rows" on page 9-35.

You can use the same untyped **row** variable to successively store different row types, but you must select the associated row-type column into the **row** variable for each new row type.

Using Named Row Types

A named row type associates a name with the row structure. For a database you create a named row type with the CREATE ROW TYPE statement. If the database contains more than one row type with the same structure but with distinctly different names, the database server cannot properly enforce structural equivalence when it compares named row types. To resolve this ambiguity, specify a row-type name in the declaration of the **row** variable.

A named Informix ESQL/C **row** variable can be typed or untyped.

The Informix ESQL/C preprocessor does not check the validity of a row-type name and Informix ESQL/C does not use this name at runtime. Informix ESQL/C just sends this name to the database server to provide information for type resolution. Therefore, Informix ESQL/C treats the **a_row** variable in the following declaration as an untyped **row** variable even though a row-type name is specified:

```
EXEC SQL BEGIN DECLARE SECTION;
   row 'address_t' a_row;
EXEC SQL END DECLARE SECTION;
```

If you specify both the row-type name and a row structure in the declaration (a typed named **row** variable), the row-type name overrides the structure. For example, suppose the database contains the following definition of the **address_t** named row type:

```
CREATE ROW TYPE address_t
(
   line1       char(20),
   line2       char(20),
   city        char(20),
   state       char(2),
   zipcode     integer
);
```

In the following declaration, the **another_row** host variable has **line1** and **line2** fields of type CHAR(20) (from the **address_t** row type:), not CHAR(10) as the declaration specifies:

```
EXEC SQL BEGIN DECLARE SECTION;
   row 'address_t' (line1 char(10), line2 char(10),
      city char(20), state char(2), zipcode integer) another_row;
EXEC SQL END DECLARE SECTION;
```

In a Collection-Derived Table: You cannot specify a named row type to declare a row variable that you use in a collection-derived table. Informix ESQL/C does not have information about the named row type, only the database server does. For example, suppose your database has a named row type, **r1**, and a table, **tab1**, that are defined as follows:

```
CREATE ROW TYPE r1 (i integer);

CREATE TABLE tab1
(
   nt_col INTEGER,
   row_col r1
);
```

To access this column, suppose you declare two row variables, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
row (i integer) row1;
row (j r1) row2;
EXEC SQL END DECLARE SECTION;
```

With these declarations, the following statement succeeds because Informix ESQL/C has the information it needs about the structure of **row1**:

```
EXEC SQL update table(:row1) set i = 31;
checksql("UPDATE Collection Derived Table 1");
```

The following statement fails; however, because Informix ESQL/C does not have the necessary information to determine the correct storage structure of the **row2** row variable.

```
EXEC SQL update table(:row2) set j = :row1;
checksql("UPDATE Collection Derived Table 2");
```

Similarly, the following statement also fails. In this case, Informix ESQL/C treats **r1** as a user-defined type instead of a named row type.

```
EXEC SQL insert into tab1 values (:row2);
checksql("INSERT row variable");
```

You can get around this restriction in either of the following ways:

- Use the actual data types in the row-variable declarations, as the following example shows:

```
EXEC SQL BEGIN DECLARE SECTION;
row (i integer) row1;
row (j row(i integer)) row2;
EXEC SQL END DECLARE SECTION;
```

- Declare an untyped row variable and perform a select so that Informix ESQL/C obtains the data-type information from the database server.

```
EXEC SQL BEGIN DECLARE SECTION;
row (i integer) row1;
row row2_untyped;
EXEC SQL END DECLARE SECTION;
    .
    .
    .
EXEC SQL select row_col into :row2_untyped from tab1;
```

For this method to work, at least one row must exist in table **tab1**.

An UPDATE statement that uses either the **row2** or **row2_untyped** row variable in its collection-derived table clause can now execute successfully.

Client-Side Rows

A **row** variable is sometimes called a *client-side row*. When you declare a **row** variable, you must declare the **row** variable name, allocate memory, and perform operations on the row.

To access the elements of a **row** variable, you specify the variable in the collection-derived table clause of a SELECT or UPDATE statement. When either of these statements contains a collection-derived table clause, Informix ESQL/C performs the select or update operation on the **row** variable; it does *not* send these statements to the database server for execution. For example, Informix ESQL/C executes the update operation on the **row** variable, **a_row**, that the following UPDATE statement specifies:

```
EXEC SQL update table(:a_row) set fld1 = 6;
```

To access fields of a row type, you must use the SELECT or UPDATE statements with the collection-derived table clause.

For more information about the collection-derived table clause, see "Accessing a collection" on page 9-2.

Managing Memory for Rows

Once you declare a **row** variable, Informix ESQL/C recognizes the variable name. For typed **row** variables, Informix ESQL/C also recognizes the associated data type. However, Informix ESQL/C does not automatically allocate or deallocate memory for **row** variables. You must explicitly manage memory that is allocated to a **row** variable. To manage memory for both typed and untyped **row** host variables, use the following SQL statements:

- The ALLOCATE ROW statement allocates memory for the specified **row** variable.

  This **row** variable can be a typed or untyped row. The ALLOCATE ROW statement sets **SQLCODE** (**sqlca.sqlcode**) to zero if the memory allocation was successful and a negative error code if the allocation failed.

- The DEALLOCATE ROW statement deallocates or frees memory for a specified **row** variable.

Once you free the **row** variable with the DEALLOCATE ROW statement, you can reuse the **row** variable but you must allocate memory for it again. You might, for example, use an untyped row variable to store different row types in succession.

Important: Informix ESQL/C does not implicitly deallocate memory that you allocate with the ALLOCATE ROW statement. You must explicitly perform memory deallocation with the DEALLOCATE ROW statement.

The following code fragment declares the **a_name** host variable as a typed row, allocates memory for this variable, then deallocates memory for this variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (
      fname char(15),
      mi char(2)
      lname char(15)
      ) a_name;
EXEC SQL END DECLARE SECTION;
:
:

EXEC SQL allocate row :a_name;
:
:

EXEC SQL deallocate row :a_name;
```

For syntax information for the ALLOCATE ROW and DEALLOCATE ROW statements, see their descriptions in the *IBM Informix Guide to SQL: Syntax*.

Operating on a Row Variable

The SELECT, and UPDATE statements allow you to access a row-type column as a whole.

An Informix ESQL/C client application can access individual fields as follows:

- Use SQL statements and dot notation to directly select, insert, update, or delete fields in row-type columns of the database with SQL statements, as long as these operations involve literal values.

   Unlike collection columns, the SELECT statement can access individual members of row-type columns. Therefore, an Informix ESQL/C client application can directly select or update fields in row-type columns of the database. For more information, see "Operating on a Row-Type Column" on page 9-44.

- Use a **row** host variable to perform operations on the row as a whole or on individual fields.

Important: You cannot use dot notation in a SELECT statement to access the fields of a nested row in a **row** variable.

With a **row** host variable, you access a row-type column as a *collection-derived table*. The collection-derived table contains a single row in which each column is a field. A collection-derived table allows you to decompose a row into its fields and then access the fields individually.

The application first performs the operations on the fields through the **row** host variable. Once modifications are complete, the application can save the contents of the **row** variable into a row-type column of the database.

This section discusses the following topics on how to use a collection-derived table in an Informix ESQL/C application to access a row-type column:

- How to use the collection-derived table clause in SQL statements to access a **row** host variable
- How to initialize a **row** host variable with a row-type column
- How to select fields from a **row** host variable
- How to update field values in a **row** host variable

Using the Collection-Derived Table Clause on Row Types

The collection-derived table clause allows you to create a collection-derived table from a row-type column. This clause has the following syntax:

```
TABLE(:row_var)
```

The variable *row_var* is a **row** host variable. It can be either a typed or untyped **row** host variable but you must declare it beforehand.

For more information about the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the *IBM Informix Guide to SQL: Syntax*.

Accessing a Row Variable: You can perform the following operations on the **row** host variable with the collection-derived table clause:

- You can select a field or fields from a **row** host variable with the collection-derived table clause in the FROM clause of SELECT statement.

  For more information, see "Selecting from a Row Variable" on page 9-39.

- You can update all or some fields in the **row** host variable collection-derived table clause after the UPDATE keyword in an UPDATE statement.

  For more information, see "Updating a Row Variable" on page 9-41.

The insert and delete operations are not supported on **row** variables. For more information, see 9-39 and 9-41.

Tip: If you only need to insert or update a row-type column with literal values, you do not need to use a **row** host variable. Instead, you can explicitly list the literal-row value in either the INTO clause of the INSERT statement or the SET clause of the UPDATE statement.

For more information, see "Inserting into and Updating Row-Type Columns" on page 9-45.

Once the **row** host variable contains the values you want, update the row-type column with the contents of the host variable. For more information, see "Accessing a Typed Table" on page 9-44. For more information about the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the *IBM Informix Guide to SQL: Syntax*.

Distinguishing Between Columns and Row Variables: When you use the collection-derived table clause with a SELECT or UPDATE statement, Informix ESQL/C processes the statement. It does not send it to the database server. Therefore, some of the syntax checking that the database server performs is not done on SQL statements that include the collection-derived table clause.

In particular, the Informix ESQL/C preprocessor cannot distinguish between column names and host variables. Therefore, when you use the collection-derived

table clause with an UPDATE statement to modify a **row** host variable, the preprocessor does not check that you correctly specify host variables. You must ensure that you use valid host-variable syntax.

If you omit the host-variable symbol (colon (:) or dollar sign ($)), the preprocessor assumes that the name is a column name. For example, the following UPDATE statement omits the colon for the **clob_ins** host variable in the SET clause:

```
EXEC SQL update table(:named_row1)
   set (int_fld, clob_fld, dollar_fld) =
      (10000000, clob_ins, 110.02);
```

Initializing a Row Variable

To perform operations on existing fields in a row-type column, you must first initialize the **row** variable with the field values. To perform this initialization, select the existing fields of the row-type column into a **row** variable with the SELECT statement, as follows:

* Specify the row-column name in the select list of the SELECT statement.
* Specify the **row** host variable in the INTO clause of the SELECT statement.
* Specify the table or view name, not the collection-derived table clause, in the FROM clause of the SELECT statement.

Suppose you create the **tab_unmrow** and **tab_nmrow** tables with the statements in Figure 9-9.

```
EXEC SQL create table tab_unmrow
(
   area integer,
   rectangle row(
           x integer,
           y integer,
           length integer,
           width integer)
);

EXEC SQL create row type full_name
(
   fname char(15),
   mi char(2),
   lname char(15)
);
EXEC SQL create table tab_nmrow
(
   emp_num integer,
   emp_name full_name
);
```

Figure 9-9. Sample Tables with Row-Type Columns

The following code fragment initializes a typed **row** host variable called **a_rect** with the contents of the **rectangle** column in the row whose **area** column is 1234:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (
        x integer,
        y integer,
        length integer,
        width integer
     ) a_rect;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL allocate row :a_rect;
EXEC SQL select rectangle into :a_rect from tab_unmrow
   where area = 1234;
```

When you use a typed **row** host variable, the data types of the row-type column (the field types) must be compatible with the corresponding data types of the typed **row** host variable. The SELECT statement in the preceding code fragment successfully retrieves the **rectangle** column because the **a_rect** host variable has the same field types as the **rectangle** column.

The following SELECT statement *fails* because the data types of the fields in the **emp_name** column and the **a_rect** host variable do *not* match:

```
/* This SELECT generates an error */
EXEC SQL select emp_name into :a_rect from tab_nmrow;
```

You can select any row into an untyped **row** host variable. The following code fragment uses an untyped **row** host variable to access the **emp_name** and **rectangle** columns that Figure 9-9 on page 9-38 defines:

```
EXEC SQL BEGIN DECLARE SECTION;
   row an_untyped_row;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate row :an_untyped_row;
EXEC SQL select rectangle into :an_untyped row
   from tab_unmrow
   where area = 64;
:

EXEC SQL select emp_name into :an_untyped_row
   from tab_nmrow
   where row{'Tashi'} in (emp_name.fname);
```

Both SELECT statements in this code fragment can successfully retrieve row-type columns into the **an_untyped_row** host variable. However, Informix ESQL/C does not perform type checking on an untyped **row** host variable because its elements do not have a predefined data type.

Once you have initialized the **row** host variable, you can use the collection-derived table clause to select or update existing fields in the row. For more information, see the following sections.

Inserting into a Row Variable

You cannot insert to a **row** variable using an INSERT statement. The **row** variable represents a single table row in the form of a collection-derived table. Each field in the row type is like a column in this virtual table. Informix ESQL/C returns an error if you attempt to insert to a **row** variable.

You can, however, use the UPDATE statement to insert new field values into a **row** variable. For more information, see "Updating a Row Variable" on page 9-41.

Selecting from a Row Variable

The SELECT statement and the collection-derived table clause allow you to select a particular field or group of fields in the **row** variable. The INTO clause identifies the host variables that holds the field values selected from the row-type variable. The data type of the host variable in the INTO clause must be compatible with the field type.

For example, Figure 9-10 contains a code fragment that puts the value of the **width** field (in the **row** variable **myrect**) into the **rect_width** host variable.

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x int, y int, length float, width float) myrect;
   double rect_width;
EXEC SQL END DECLARE SECTION;
   .
   .
   .
EXEC SQL select rect into :myrect from rectangles
   where area = 200;
EXEC SQL select width into :rect_width
   from table(:myrect);
```

Figure 9-10. Selecting from a Row Variable

The SELECT statement on a **row** variable (one that contains a collection-derived table clause) has the following restrictions:

- No expressions are allowed in the select list.
- The select list must be an asterisk (*) if the row contains elements of opaque, distinct, or built-in data types.
- Column names in the select list must be simple column names.

  These columns cannot use the *database@server*:*table*.*column* syntax.

- The select list cannot use dot notation to access fields of the row.
- The following SELECT clauses are not allowed: GROUP BY, HAVING, INTO TEMP, ORDER BY, and WHERE.
- The FROM clause has no provisions to do a join.
- Row-type columns cannot be specified in a comparison condition in a WHERE clause.

If the **row** variable is a nested row, a SELECT statement cannot use dot notation to access the fields of the inner row. Instead, you must declare a **row** variable for each row type. The code fragment in Figure 9-11 shows how to access the fields of the inner row in the **nested_row** host variable.

```
EXEC SQL BEGIN DECLARE SECTION;
   row (a int, b row(x int, y int)) nested_row;
   row (x int, y int) inner_row;
   integer x_var, y_var;
EXEC SQL END DECLARE SECTION;

EXEC SQL select row_col into :nested_row from tab_row
   where a = 7;
EXEC SQL select b into :inner_row
   from table(:nested_row);
EXEC SQL select x, y into :x_var, :y_var
   from table(:inner_row);
```

Figure 9-11. Sample Nested- Row Variable

The following SELECT statement is *not* valid to access the **x** and **y** fields of the **nested_row** variable because it uses dot notation:

```
EXEC SQL select row_col into :nested_row from tab_row
EXEC SQL select b.x, b.y    /* invalid syntax */
   into :x_var, :y_var from table(:nested_row);
```

An Informix ESQL/C application can use dot notation to access fields of a nested row when a SELECT statement accesses a database column. For more information, see "Selecting Fields of a Row Column" on page 9-45.

Updating a Row Variable

The UPDATE statement and the collection-derived table clause allow you to update a particular field or group of fields in the **row** variable. You specify the new field values in the SET clause. An UPDATE of a field or fields in a **row** variable cannot include a WHERE clause.

For example, the following UPDATE changes the **x** and **y** fields in the **myrect** Informix ESQL/C **row** variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x int, y int, length float, width float) myrect;
   int new_y;
EXEC SQL END DECLARE SECTION;
   :
   :

new_y = 4;
EXEC SQL update table(:myrect)
   set x=3, y=:new_y;
```

You cannot use a **row** variable in the collection-derived table clause of an INSERT statement. However, you can use the UPDATE statement and the collection-derived table clause to insert new field values into a **row** host variable, as long as you specify a value for every field in the row. For example, the following code fragment inserts new field values into the **row** variable **myrect** and then inserts this **row** variable into the database:

```
EXEC SQL update table(:myrect)
   set x=3, y=4, length=12, width=6;
EXEC SQL insert into rectangles
   values (72, :myrect);
```

Deleting from a Row Variable

A delete operation does not apply to a **row** variable because a delete normally removes a row from a table. The **row** variable represents the row-type value as a single table row in the collection-derived table. Each field in the row type is a column in this table. You cannot remove this single table row from the collection-derived table. Therefore, the DELETE statement does not support a **row** variable in the collection-derived table clause. Informix ESQL/C returns an error if you attempt to perform a DELETE operation on a **row** variable.

However, you can use the UPDATE statement to delete existing field values in a **row** variable. For more information, see "Deleting an Entire Row Type" on page 9-47.

Specifying Field Names

Informix ESQL/C is case insensitive with regard to the field names of a **row** variable. In a SELECT or UPDATE statement, Informix ESQL/C always interprets field names of a **row** variable as lowercase. For example, in the following SELECT statement, Informix ESQL/C interprets the fields to select as **x** and **y**, even though the SELECT statement specifies them in uppercase:

```
EXEC SQL select X, Y from table(:myrect);
```

This behavior is consistent with how the database server handles identifier names in SQL statements. To maintain the case of a field name, specify the field name as

a delimited identifier. That is, surround the field name in double quotes and enable the **DELIMIDENT** environment variable before you compile the program.

Informix ESQL/C interprets the fields to select as **X** and **Y** (uppercase) in the following SELECT statement (assuming the **DELIMIDENT** environment variable is enabled):

```
EXEC SQL select "X", "Y" from table(:myrect);
```

For more information about delimited identifiers and the **DELIMIDENT** environment variable, see "SQL Identifiers" on page 1-12.

Use Host Variable Field Names: If the field names of the row column and the row variable are different, you must specify the field names of the row host variable. For example, if the last SELECT statement in the following example referenced field names **x** and **y** instead of the field names of **a_row**, it would generate a runtime error.

```
EXEC SQL BEGIN DECLARE SECTION;
   row (a integer, b float) a_row;
   int i;
   double f;
EXEC SQL END DECLARE SECTION;

EXEC SQL create table tab (row_fld(x integer, y float));
EXEC SQL insert into tab values ('row(9, 3.34e7)');
EXEC SQL select * into a_row from tab;
EXEC SQL select a, b into :i, :f from table(:a_row);
```

Specifying Field Values

You can specify any of the following values for fields in a **row** variable:

- A literal value

  You can also specify literal values directly for a row-type column without first using a **row** variable. For more information, see "Inserting into and Updating Row-Type Columns" on page 9-45.

- A constructed row

  You cannot include complex expressions directly to specify field values. However, a constructed row provides support for expressions as field values.

- An Informix ESQL/C host variable

For information about how to update field values into a **row** variable, see "Updating a Row Variable" on page 9-41. The following sections describe the values you can assign to a field in a **row** variable.

Literal Values as Field Values: You can use a literal value to specify a field value for a **row** variable. The literal values must have a data type that is compatible with the field type. For example, the following UPDATE statement specifies a literal integer as a field value for the **length** field of the **myrect** variable. See "Updating a Row Variable" on page 9-41 for a description of **myrect**.

```
EXEC SQL update table(:myrect) set length = 6;
```

The following UPDATE statement updates the *x*- and *y*-coordinate fields of the **myrect** variable:

```
EXEC SQL update table(:myrect)
   set (x = 14, y = 6);
```

The following UPDATE statement updates a ROW(a INTEGER, b CHAR(5)) host variable called **a_row2** with a quoted string:

```
EXEC SQL update table(:a_row2) set b = 'abcde';
```

The following UPDATE statement updates the **nested_row** host variable (which
Figure 9-11 on page 9-40 defines) with a literal row:

```
EXEC SQL insert into table(:nested_row)
   values (1, row(2,3));
```

Important: The syntax of a literal row for a **row** variable is different from the
syntax of a literal row for a row-type column. A **row** variable does not
need to be a quoted string.

If you only need to insert or update the row-type column with literal values, you
can list the literal values as a literal-row value in the INTO clause of the INSERT
statement or the SET clause of the UPDATE statement.

For more information, see "Inserting into and Updating Row-Type Columns" on
page 9-45.

Constructed Rows: You can use a constructed row to specify an expression as a
field value for a **row** variable. The constructed expression must use a row
constructor and evaluate to a data type that is compatible with the field type of the
field. Suppose you have a nested-row variable that is declared as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (fld1 integer, fld2 row(x smallint, y char(5))) a_nested_row;
EXEC SQL END DECLARE SECTION;
```

The following UPDATE statement uses the ROW constructor to specify expressions
in the value for the **fld2** field of the **a_nested_row** variable:

```
EXEC SQL update table(:a_nested_row)
   set fld2 = row(:an_int, a_func(:a_strng));
```

For more information about the syntax of a row constructor, see the Expression
segment in the *IBM Informix Guide to SQL: Syntax*.

ESQL/C Host Variables as Field Values: You can use an Informix ESQL/C host
variable to specify a field value for a **row** variable. The host variable must be
declared with a data type that is compatible with the data type of the field and
must contain a value that is also compatible. For example, the following UPDATE
statement uses a host variable to update a single value into the **a_row** variable.

```
an_int = 6;
EXEC SQL update table(:a_row) set fld1 = :an_int;
```

To insert multiple values into a **row** variable, you can use an UPDATE statement
for each value or you can specify all field values in a single UPDATE statement:

```
one_fld = 469;
second_fld = 'dog';
EXEC SQL update table(:a_row)
   set fld1 = :one_fld, fld2 = :second_fld;
```

The following variation of the UPDATE statement performs the same task as the
preceding UPDATE statement:

```
EXEC SQL update table(:a_row) set (fld1, fld2) =
   (:one_fld, :second_fld);
```

The following UPDATE statement updates the **nested_row** variable with a literal
field value and a host variable:

```
EXEC SQL update table(:nested_row)
   set b = row(7, :i);
```

Accessing a Typed Table

You can use a **row** variable to access the columns of a typed table. A *typed table* is a table that was created with the OF TYPE clause of the CREATE TABLE statement. This table obtains the information for its columns from a named row type.

Suppose you create a typed table called **names** from the **full_name** named row type that Figure 9-9 on page 9-38 defines:

```
EXEC SQL create table names of type full_name;
```

You can access a row of the **names** typed table with a **row** variable. The following code fragment declares **a_name** as a typed **row** variable and selects a row of the **names** table into this **row** variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (
      fname char(15),
      mi char(2)
      lname char(15)
      ) a_name;
   char last_name[16];
EXEC SQL END DECLARE SECTION;
:

EXEC SQL allocate row :a_name;
EXEC SQL select name_row into :a_name
   from names name_row
   where lname = 'Haven'
    and fname = 'C. K.'
    and mi = 'D';
EXEC SQL select lname into :last_name from table(:a_name);
```

The last SELECT statement accesses the **lname** field value of the **:a_name** row variable. For more information about typed tables, see the CREATE TABLE statement in the *IBM Informix Guide to SQL: Syntax* and the *IBM Informix Guide to SQL: Tutorial*.

The following example illustrates how you can also use an untyped **row** variable to access a row of an untyped table:

```
EXEC SQL BEGIN DECLARE SECTION;
row untyped_row;
int i;
char s[21];
EXEC SQL END DECLARE SECTION;

EXEC SQL create table tab_untyped(a integer, b char(20));
EXEC SQL insert into tab_untyped(1, "junk");
EXEC SQL select tab_untyped into :untyped_row
   from tab_untyped;
EXEC SQL select a, b into :i, :s from table(:untyped);
```

Operating on a Row-Type Column

The **row** variable stores the fields of the row type. The **row** variable, however, has no intrinsic connection with a database column. You must use an INSERT or UPDATE statement to explicitly save the contents of the variable into the row type column.

You can use the SELECT, UPDATE, INSERT, and DELETE statements to access a row-type column (named or unnamed), as follows:

- The SELECT statement fetches all fields or a particular field from a row-type column.
- The INSERT statement inserts a new row into a row-type column.
- The UPDATE statement updates the entire row in a row-type column with new values.

  Use an UPDATE statement on a table or view name and specify the **row** name in the values clause.
- The DELETE statement deletes from a table a row that contains a row-type column, thereby deleting all field values from the row-type column.

For more information about how to use these statements with row-type columns, see the *IBM Informix Guide to SQL: Tutorial*.

Selecting from a Row-Type Column

The SELECT statement allows you to access a row-type column in the following ways:

- Selecting all fields in the row-type column
- Selecting particular fields in the row-type column

Selecting the Entire Row-Type Column: To select *all* fields in a row-type column, specify the row-type column in the select list of the SELECT statement. To access these fields from an Informix ESQL/C application, specify a **row** host variable in the INTO clause of the SELECT statement. For more information, see "Initializing a Row Variable" on page 9-38.

Selecting Fields of a Row Column: You can access an individual field in a row-type column with *dot notation*. Dot notation allows you to qualify an SQL identifier with another SQL identifier. You separate the identifiers with the period (**.**) symbol. The following SELECT statement performs the same task as the two SELECT statements in Figure 9-10 on page 9-40:

```
EXEC SQL select rect.width into :rect_width from rectangles;
```

For more information about dot notation, see the Column Expression section of the Expression segment in the *IBM Informix Guide to SQL: Syntax*.

Inserting into and Updating Row-Type Columns

The INSERT and UPDATE statements support row-type columns as follows:

- To *insert* a new row into a row-type column, specify the new values in the VALUES clause of the INSERT statement.
- To *update* the entire row-type column, specify the new field values in the SET clause of the UPDATE statement.

In the VALUES clause of an INSERT statement or the SET clause of an UPDATE statement, the field values can be in any of the following formats:

- An Informix ESQL/C **row** host variable

  For more information, see "Accessing a Typed Table" on page 9-44.
- A constructed row

  Constructed rows are described with respect to **row** variables in "Constructed Rows" on page 9-43. For information about the syntax of a constructed row, see the Constructed Row segment in the *IBM Informix Guide to SQL: Syntax*.
- A literal-row value

For more information about the syntax of a literal-row value, see the Literal Row segment in the *IBM Informix Guide to SQL: Syntax*.

To represent literal values for a row-type column, you specify a literal-row value. You create a literal-row value or a named or unnamed row type, introduce the value with the ROW keyword and provide the field values in a comma-separated list that is enclosed in parentheses. You surround the entire literal-row value with quotes (double or single). The following INSERT statement inserts the literal row of ROW(0, 0, 4, 5) into the **rectangle** column in the **tab_unmrow** table (that Figure 9-9 on page 9-38 defines):

```
EXEC SQL insert into tab_unmrow values
(
   20, "row(0, 0, 4, 5)"
);
```

The UPDATE statement in Figure 9-12 overwrites the SET values that the previous INSERT added to the **tab_unmrow** table.

```
EXEC SQL update tab_unmrow
   set rectangle = ("row(1, 3, 4, 5)")
   where area = 20;
```

Figure 9-12. Updating a Row-Type Column

Important: If you omit the WHERE clause, the preceding UPDATE statement updates the **rectangle** column in all rows of the **tab_unmrow** table.

If any character value appears in this literal-row value, it too must be enclosed in quotes; this condition creates nested quotes. For example, a literal value for column **row1** of row type ROW(id INTEGER, name CHAR(5)), would be:

```
'ROW(6, "dexter")'
```

To specify nested quotes in an SQL statement in an Informix ESQL/C program, you must escape every double quote when it appears in a single-quote string. The following two INSERT statements show how to use escape characters for inner quotes:

```
EXEC SQL insert into (row1) tab1
   values ('ROW(6, \"dexter\")');

EXEC SQL insert into (row2) tab1
   values ('ROW(1, \"SET{80, 81, 82, 83}\")');
```

When you embed a double-quoted string inside another double-quoted string, you do not need to escape the inner-most quotation marks:

```
EXEC SQL insert into tabx
   values (1, "row(""row(12345)"")");
```

For more information about the syntax of literal values for **row** variables, see "Literal Values as Field Values" on page 9-42. For more information about the syntax of literal-row values, see the Literal Row segment in the *IBM Informix Guide to SQL: Syntax*.

If the row type contains a row type or a collection as a member, the inner row does not need quotes. For example, for column **col2** whose data type is ROW(a INTEGER, b SET (INTEGER)), a literal value would be:

```
'ROW(1, SET{80, 81, 82, 83})'
```

Deleting an Entire Row Type

To *delete* all fields in a row-type column, specify the table, view, or synonym name after the FROM keyword of the DELETE statement and use the WHERE clause to identify the table row(s) that you want to delete.

The following DELETE statement deletes the row in the **tab_unmrow** table that contains the row type that the UPDATE statement in Figure 9-12 on page 9-46 saves:

```
EXEC SQL delete from tab_unmrow
   where area = 20;
```

Chapter 10. Working with opaque data types

The information in these topics apply only if you are using IBM Informix as your database server.

These topics explain how to use the **lvarchar**, **fixed binary**, and **var binary** data types to access an opaque data type from an IBM Informix ESQL/C program. Use these Informix ESQL/C data types to represent an opaque data type as it is transferred to and from IBM Informix.

For information about all the data types that are available for use in an Informix ESQL/C program, see Chapter 3, "Informix ESQL/C data types," on page 3-1. For information about SQL complex data types, see the *IBM Informix Guide to SQL: Reference*.

The SQL opaque Data Type

An *opaque data type* is a user-defined data type that can be used in the same way as the Informix built-in data types. The opaque data type allows you to define new data types for your database applications.

An opaque data type is fully encapsulated; the database server does not know about the internal format of an opaque data type. Therefore, the database server cannot make assumptions about how to access a column having an opaque data type. The database developer defines a data structure that holds the opaque-type information and support functions that tell the database server how to access this data structure.

For more information about how to create an opaque data type, see the description of the CREATE OPAQUE TYPE statement in the *IBM Informix Guide to SQL: Syntax* and in *IBM Informix User-Defined Routines and Data Types Developer's Guide*.

You can access the value of an opaque data type from an Informix ESQL/C application in one of two ways:

*   In the external format, as a character string

Transfer of the external format between the client application and database server is supported by the database server through the input and output support functions of the opaque data type.

- In the internal format, as a data structure in an external programming language (such as C)

  Transfer of the internal format between the client application and database server is supported by the database server through the receive and send support functions of the opaque data type.

The following table shows the Informix ESQL/C data types you can use to access an opaque data type.

Informix Data Type	ESQL/C Host Variable
External format of an opaque data type	**lvarchar** host variable
Internal format of an opaque data type	**fixed binary** host variable
	var binary host variable

This chapter uses an opaque data type called **circle** to demonstrate how Informix ESQL/C **lvarchar** and **fixed binary** host variables access an opaque data type. This data type includes an **x,y** coordinate, to represent the center of the circle, and a radius value. Figure 10-1 shows the internal data structures for the **circle** data type.

```
typedef struct
   {
   double     x;
   double     y;
   } point_t;

typedef struct
   {
   point_t         center;
   double         radius;
   } circle_t;
```

Figure 10-1. Internal Data Structures for the circle Opaque Data Type

Figure 10-2 shows the SQL statements that register the **circle** data type and its input, output, send, and receive support functions in the database.

```
CREATE OPAQUE TYPE circle (INTERNALLENGTH = 24,
   ALIGNMENT = 4);

CREATE FUNCTION circle_in(c_in lvarchar) RETURNS circle
   EXTERNAL NAME '/usr/lib/circle.so(circle_input)'
   LANGUAGE C;
CREATE IMPLICIT CAST (lvarchar AS circle WITH circle_in);

CREATE FUNCTION circle_out(c_out circle) RETURNS lvarchar
   EXTERNAL NAME '/usr/lib/circle.so(circle_output)'
   LANGUAGE C;
CREATE IMPLICIT CAST (circle AS lvarchar WITH circle_out);

CREATE FUNCTION circle_rcv(c_rcv sendrcv) RETURNS circle
   EXTERNAL NAME '/usr/lib/circle.so(circle_receive)'
   LANGUAGE C;
CREATE IMPLICIT CAST (sendrcv AS circle WITH circle_rcv);

CREATE FUNCTION circle_snd(c_snd circle) RETURNS sendrcv
   EXTERNAL NAME '/usr/lib/circle.so(circle_send)'
   LANGUAGE C;
CREATE IMPLICIT CAST (circle AS sendrcv WITH circle_snd);

CREATE FUNCTION radius(circle) RETURNS FLOAT
   EXTERNAL NAME '/usr/lib/circle.so'
   LANGUAGE C;
```

Figure 10-2. Registering the circle Opaque Data Type

Suppose the input and output functions of the **circle** data type define the following external format that Figure 10-3 shows.

(x,y) coordinates
for the center of
the circle

Radius of the
circle
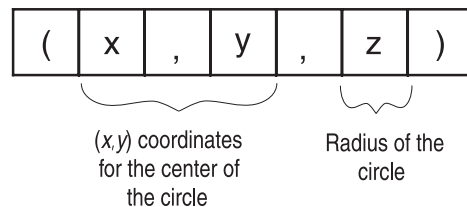
Figure 10-3. External Format of the circle Opaque Data Type

Figure 10-4 shows the SQL statements that create and insert several rows into a table called **circle_tab,** which has a column of type **circle**.

```
CREATE TABLE circle_tab (circle_col  circle);
INSERT INTO circle_tab VALUES ('(12.00, 16.00, 13.00)');
INSERT INTO circle_tab VALUES ('(6.5, 8.0, 9.0)');
```

Figure 10-4. Creating a Column of the circle Opaque Data Type

Accessing the External Format of an Opaque Type

Use the **lvarchar** data type for operations on an opaque-type column that has an external representation of a character string.

To use the external format of an opaque type in an SQL statement, the opaque data type must have input and output support functions defined. When the client application uses an **lvarchar** host variable to transfer data to or from an opaque-type column, the database server invokes the following support functions of the opaque data type:

- The *input* support function describes how to transfer the opaque-type data from the **lvarchar** host variable into the opaque-type column.

  The database server invokes the input support function for operations such as INSERT and UPDATE statements that send the external format of an opaque type to the database server.

- The *output* support function describes how to transfer the opaque-type data from the opaque-type column to the **lvarchar** host variable.

  The database server invokes the output support function for operations such as SELECT and FETCH statements that send the external format of an opaque type to the client application.

Important: If the CREATE OPAQUE TYPE statement specifies a maxlength limit, that value is the maximum length the database server will store for the column, regardless of the size of the data sent by the client application. If the length of the data is more than the maxlength limit, the database server truncates the data and notifies the application.

Follow these steps to transfer the external format of an opaque-type column between the database server and the Informix ESQL/C application:

1. Declare an **lvarchar** host variable
2. Use the **lvarchar** host variable in an SQL statement to perform any select, insert, update or delete operations on the external format of the opaque-type column.

The following sections describe each of these steps in more detail.

Declaring lvarchar Host Variables

Use the **lvarchar** data type to declare a host variable for the external format of an opaque data type. The following diagram illustrates the syntax to declare an **lvarchar** host variable. To declare, use the **lvarchar** keyword as the variable data type, as the following syntax shows.

```
                (1)
►►──────────lvarchar──────────────────────────────────────────►
                     └─'─opaque type─'─┘


     ┌─────────,──────────────────┐
     │         ▼                  │
►──────┬─variable name─[─variable size─]─┬──;──────────────────►◄
       └─*─variable name────────────────┘
```

Notes:

1   Informix extension

Element	Purpose	Restrictions	SQL Syntax
opaque type	Name of the opaque data type whose external format is to be stored in the **lvarchar** variable	Must already be defined in the database	Identifier segment in the *IBM Informix Guide to SQL: Syntax*
variable name	Name of the Informix ESQL/C variable to declare as an **lvarchar** variable		Name must conform to language-specific rules for variable names.
*variable name	Name of an **lvarchar** pointer variable for data of unspecified length	Not equivalent to a C char pointer (char *). Points to an internal ESQL/C representation for this type. You must use the **ifx_var()** functions to manipulate data. For more information, see "The lvarchar pointer and var binary library functions" on page 10-22.	Name must conform to language-specific rules for variable names.
variable size	Number of bytes to allocate for the **lvarchar** variable	Integer value can be between 1 and 32,000 bytes (32 kilobytes).	

> **Tip:** To declare an **lvarchar** host variable for an LVARCHAR column, use the syntax that "The lvarchar Data Type" on page 4-5 shows.

Figure 10-5 shows declarations for four **lvarchar** variables that hold the external formats of opaque-type columns.

```
#define CIRCLESZ 20

EXEC SQL BEGIN DECLARE SECTION;
    lvarchar 'shape' a_polygon[100];
    lvarchar 'circle' circle1[CIRCLESZ],
                circle2[CIRCLESZ];
    lvarchar 'circle' *a_crcl_ptr;
EXEC SQL END DECLARE SECTION;
```

Figure 10-5. Sample lvarchar Host Variables for Opaque Data Type

You can declare several **lvarchar** variables in a single declaration line. However, all variables must have the *same* opaque type, as the declarations for **circle1** and **circle2** in Figure 10-5 show. Figure 10-5 also shows the declaration of an **lvarchar** pointer for the **a_crcl_ptr** host variable.

An lvarchar Host Variable of a Fixed Size

If you do not specify the size of an **lvarchar** host variable, the size is equivalent to a 1-byte C-language **char** data type. If you specify a size, the **lvarchar** host variable is equivalent to a C-language **char** data type of that size. When you specify a fixed-size **lvarchar** host variable, any data beyond the specified size is truncated when the column is fetched. Use an indicator variable to check for truncation.

Because an **lvarchar** host variable of a fixed size is equivalent to a C-language **char** data type, you can use C-language character string operations to manipulate them.

For more information about how to use an **lvarchar** host variable of a fixed size, see "A lvarchar Host Variable of a Fixed Size" on page 4-6.

The lvarchar Pointer Host Variable

The *lvarchar pointer host variable* is designed for inserting or selecting user-defined or opaque types that can be represented in a character-string format. The size of the character-string representation for opaque type columns can vary for each row so that the size of the data is unknown until the column is fetched into a host variable. The size of the data that an **lvarchar** pointer host variable references can range up to 2 gigabytes.

The **lvarchar** pointer type is not equivalent to a C-language **char** pointer. Informix ESQL/C maintains its own internal representation for the **lvarchar** pointer type. This representation is identical to the representation of a **var binary** host variable, except that it supports ASCII data as opposed to binary data. You must use the **ifx_var()** functions to manipulate an **lvarchar** pointer host variable. The **ifx_var()** functions can only be used for lvarchar variables declared as pointers and for **var binary** variables, but not for **lvarchar** variables of a fixed size. For a list of the functions that you can use with **lvarchar** and **var binary** variables, see "The lvarchar pointer and var binary library functions" on page 10-22.

Because the size of the data in opaque type columns can vary from one row in the table to another, you cannot know the maximum size of the data that the database server will return. When you use an **lvarchar** pointer host variable, you can either let Informix ESQL/C allocate memory to hold the data, based on the size of the data coming from the database server, or you can allocate the memory yourself. Use the **ifx_var_flag()** function to specify which method you will use. In either case you must explicitly free the memory, using the **ifx_var_dealloc()** function.

To Allocate Memory Yourself: To specify that you are allocating the memory to store data for an **lvarchar** host variable you must first call **ifx_var_flag()**, giving the address of the **lvarchar** pointer and setting the flag value to zero (0), as the following example shows:

```
ifx_var_flag(&mypoly, 0);
```

Next you must fetch the data into sqlda or a system descriptor structure. You can then use the **ifx_var_getllen()** function to obtain the length of the data and use the **ifx_var_alloc()** function to allocate memory of that size.

```
#include <stdio.h>
exec sql include "polyvar.h" /* includes udt - polygon_type */

main()
{
   exec sql begin declare section;
      lvarchar 'polygon_type' *mypoly1
      char *buffer;
      int size, p_id, len;
   exec sql end declare section;

   ifx_var_flag(&mypoly1, 0); /* specifies that appl. will allocate
memory */
   exec sql select poly into :mypoly1 from polygon_tab where p_id = 1;
   if ( ifx_var_getlen(&mypoly1) > 0 ) { /* If select returns valid data
                                 */
   buffer = (char *)ifx_var_getdata(&mypoly1); /*Access data in
                                          * mypoly1*/
   printf("Length of data : %ld \n", (int)ifx_var_getlen(&mypoly) );
   ifx_var_dealloc(&mypoly1); /* Always users responsibility to free */
}
```

The Opaque Type Name

This *opaque type* name is optional; its presence affects the declaration as follows:

- When you omit *opaque type* from the **lvarchar** declaration, the database server attempts to identify the appropriate support and casting functions to use when it converts between **lvarchar** and the opaque data type.

  You can use the **lvarchar** host variable to hold data for several different opaque types (as long as the database server is able to find the appropriate support functions).

- When you specify *opaque type* in the **lvarchar** declaration, the database server knows precisely which support and casting functions to use when it converts between **lvarchar** and the opaque data type.

  Using *opaque type* can make data conversion more efficient. In this case, however, the **lvarchar** host variable can hold data only for the specified opaque type.

In the declaration of an **lvarchar** host variable, the name of the *opaque type* must be a quoted string.

Important: Both the single quote (') and the double quote (") are valid quote characters in **lvarchar** declarations. However, the beginning quote and ending quote characters must match.

Using lvarchar Host Variables

Your Informix ESQL/C program must manipulate the external data for an **lvarchar** host variable. If the length of the data that come from an opaque type column does not vary, or if you know the maximum length of the data in an opaque type column, you can use a fixed-size **lvarchar** host variable. If the size of the data varies from one table row to another, however, use an **lvarchar** pointer variable and manipulate the data with the **ifx_var()** functions.

The following sections describe how to use both an **lvarchar** variable of a fixed size and an **lvarchar** pointer variable.

Using Fixed-Size lvarchar Host Variables

Figure 10-6 shows how to use a fixed-size **lvarchar** host variable to insert and select data in the **circle_col** column of the **circle_tab** table (see Figure 10-4 on page 10-3).

```
EXEC SQL BEGIN DECLARE SECTION;
   lvarchar 'circle' lv_circle[30];
   char *x_coord;
EXEC SQL END DECLARE SECTION;

/* Insert a new circle_tab row with a literal opaque
 * value */
EXEC SQL insert into circle_tab
   values ('(3.00, 2.00, 11.5)');

/* Insert data into column circle of table circle_tab using an lvarchar host
 * variable */
strcpy(lv_circle, "(1.00, 17.00, 15.25)");
EXEC SQL insert into circle_tab values (:lv_circle);

/* Select column circle in circle_tab from into an lvarchar host variable
 */
EXEC SQL select circle_col into :lv_circle
   from circle_tab
   where radius(circle_col) = 15.25;
```

Figure 10-6. Accessing the External Format of the circle Opaque Data Type

Inserting from a Fixed-Size lvarchar Host Variable: To insert the data from a fixed-size **lvarchar** host variable into an opaque-type column, take the following steps, which are illustrated in Figure 10-6:

1. Define the fixed-size **lvarchar** host variable.

   The example explicitly reserves 30 bytes for the **lv_circle** host variable.

2. Put the character string that corresponds to the external format of the opaque data type into the **lvarchar** host variable.

   When you put data into an **lvarchar** host variable, you must know the external format of the opaque type. For the INSERT statement to succeed, the data in the **lvarchar** host variable **lv_circle** must conform to the external format of the opaque data type (which Figure 10-3 on page 10-3 shows).

3. Insert the data that the **lvarchar** host variable contains into the opaque-type column.

   When the database server executes the INSERT statement, it calls the input support function for the **circle** data type (**circle_in**) to translate the external format of the data that the Informix ESQL/C client application sent to the internal format that it stores on disk.

Figure 10-6 on page 10-8 also shows an INSERT of literal values into the **circle_col** column. Literal values in an INSERT (or UPDATE) statement must also conform to the external format of the opaque data type.

You can use a fixed-size **lvarchar** host variable to insert a null value into an opaque-type column with the following steps:

- Set the **lvarchar** host variable to an empty string.
- Set an indicator variable for the **lvarchar** host variable to **-1**.

The following code fragment inserts a null value into the **circle_col** column with the **lv_circle** host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   lvarchar lv_circle[30];
   int circle_ind;
EXEC SQL END DECLARE SECTION;
   .
   .
   .
```

```
strcpy(lv_circle, "");
circle_ind = -1;
EXEC SQL insert into circle_tab
   values (:lv_circle:circle_ind)l;
```

For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

Selecting into a Fixed-Size lvarchar Host Variable: To select data from an opaque type column into a fixed-size **lvarchar** host variable, the code fragment in Figure 10-6 on page 10-8 takes the following steps:

1. Selects the data that the **circle_col** opaque-type column contains into the **lv_circle** host variable.

   When the database server executes the SELECT statement, it calls the output support function for the **circle** data type (**circle_out**) to translate the internal format that it retrieved from disk to the external format that the Informix ESQL/C application requests. This SELECT statement also uses a user-defined function called **radius** (see Figure 10-2 on page 10-3) to extract the radius value from the opaque-type column. This function must be registered with the database server for this SELECT statement to execute successfully.

2. Accesses the **circle** data from the **lvarchar** host variable.

   After the SELECT statement, the **lv_circle** host variable contains data in the external format of the **circle** data type.

When you select a null value from an opaque-type column into an **lvarchar** host variable, Informix ESQL/C sets any accompanying indicator variable to **-1**.

Using lvarchar Pointer Variables

The following sections illustrate how to insert to and select from an opaque type column with an **lvarchar** pointer host variable. The structural representation of the opaque type column that the examples use is referred to as **polygon_ type**, and is defined in the following lines:

```
struct {
int no_of_edges; /* No of sides in the polygon */
int length[100]; /* Maximum number of edges in this polygon
                  is 100 */
int center_x; /* Center x co-ordinate of the polygon */
int center_y; /* Center y co-ordinate of the polygon */
}
```

The following line illustrates the string representation of this column:

```
"no_of_edges, length_of_edge 1, . . . length_of_edge n, -1, center_x,
center_y"
```

For information about a using an lvarchar pointer host variable with a FETCH or PUT statement that uses a dynamic SQL descriptor, see "Using an lvarchar Pointer Host Variable with a Descriptor" on page 16-6.

Inserting from an lvarchar Pointer Host Variable: The following example code illustrates the steps to insert data from an **lvarchar** pointer host variable to an opaque type column. To simplify the example, the code does not check returned values for errors.

```
#include <stdio.h>
exec sql include "polyvar.h" /* includes udt - polygon_type */

main()
{
```

```
exec sql begin declare section;
    lvarchar 'polygon_type' *mypoly1
    char *buffer;
    int size, p_id, len;
exec sql end declare section;

exec sql create table polygon_tab (p_id int, poly polygon_type);
ifx_var_flag(&mypoly1, 0); /* User does allocation */
buffer = malloc(50);

/* String representation of mypoly1 copied into buffer*/
strcpy(buffer, "5, 10, 20 15, 10, 5, -1, 0, 0");
size = strlen(buffer);
ifx_var_alloc(&myploy1, size+1); /* Allocate memory for data in
                                  * mypoly1 */
ifx_var_setlen(&myploy1, size); /* Set length of data bufferin
                                 * mypoly1 */
ifx_var_setdata(&mypoly1, buffer, size); /* Store data inside mypoly1
                                          */
exec sql insert into polygon_tab values (1, :mypoly1);
ifx_var_setnull(&mypoly1, 1); /* Set data buffer in mypoly1 to NULL
                               */
ifx_var_dealloc(&mypoly1); /* Deallocate the data buffer in mypoly1
                            */
free (buffer);
}
```

The example code performs the following steps:

1. It declares the **lvarchar** pointer host variable, ***mypoly1**.
2. It creates a table that consists of an integer ID column, **p_id**, and a column of polygons, **polygon_type**.
3. It calls the **ifx_var_flag()** function to specify that it will allocate memory for the data buffer (**flag** equals 0).
4. It creates a buffer, copies the string representation of the polygon to it, and sets the **size** variable to the size of the buffer.
5. It calls **ifx_var_alloc()**, **ifx_var_setlen()**, and **ifx_var_setdata()** to allocate the data transfer buffer, set the length of the buffer, and copy the data from the application buffer to the data transfer buffer.
6. It inserts an ID value of 1 and **mypoly1** to the **polygon_tab** table.

Selecting into an lvarchar Pointer Host Variable: The following example code illustrates the steps to select data from an opaque type column into an **lvarchar** pointer host variable. To simplify the example, the code does not check returned values for errors.

```
#include <stdio.h>
exec sql include "polyvar.h" /* includes udt - polygon_type */

main()
{
    exec sql begin declare section;
        lvarchar 'polygon_type' *mypoly1
        char *buffer;
        int size, p_id, len;
    exec sql end declare section;

    ifx_var_flag(&mypoly1, 1); /* ESQL/C run time will do the allocation
                                */

    exec sql select poly into :mypoly1 from polygon_tab where p_id = 1;
```

```
    if ( ifx_var_getlen(&mypoly1) > 0 ) { /* If select returns valid data
                                    */
    buffer = (char *)ifx_var_getdata(&mypoly1); /*Access data in
                                              * mypoly1*/
    printf("Length of data : %ld \n", (int)ifx_var_getlen(&mypoly) );
    printf("Data: %s \n", buffer);
    ifx_var_dealloc(&mypoly1); /* Always users responsibility to free */
}
```

The example code performs the following steps:

1. It declares the **lvarchar** pointer host variable, ***mypoly1**.
2. It calls the **ifx_var_flag()** function to specify that it will let Informix ESQL/C allocate memory for the data buffer (**flag** equals 1). Informix ESQL/C allocates the memory by default if you do not call **ifx_var_flag()**.
3. It selects the column **poly** into the ***mypoly** host variable.
4. It calls **ifx_var_getdata()** to obtain the address of the data buffer, casting the return value to **char *** and storing the address in **buffer**.
5. It calls **ifx_var_getlen()** to illustrate how to obtain the length of the data that was retrieved.
6. It deallocates the memory that Informix ESQL/C allocated for ***mypoly1**.

For an example that uses **lvarchar** pointers as host variables for selecting from collection columns, see "The lvarptr.ec Program" on page 14-29.

Accessing the Internal Format of an Opaque Type

You can access the internal or binary format of an opaque data type with an Informix ESQL/C host variable in two ways:

- Use the **fixed binary** data type to access a fixed-length opaque data type for which you have the C-language data structure that represents the opaque data type.

  A fixed-length opaque data type has a predefined size for its data. This size is equal to the size of the internal data structure for the opaque data type.

- Use the **var binary** data type to access a varying-length opaque data type *or* to access a fixed-length opaque data type for which you do not have the C-language data structure.

  A varying-length data type holds data whose size might vary from row to row or instance to instance.

Both the **fixed binary** and **var binary** data types have a one-to-one mapping between their declaration and the internal data structure of the opaque data type. The database server invokes the following support functions of the opaque data type when the application transfers data in the **fixed binary** or **var binary** host variables:

- The *receive* support function describes how to transfer the opaque-type data from the **fixed binary** or **var binary** host variable into the opaque-type column.

  The database server invokes the receive support function for operations such as INSERT and UPDATE statements that send the internal format of an opaque type to the database server.

- The *send* support function describes how to transfer the opaque-type data from the opaque-type column to the **fixed binary** or **var binary** host variable.

The database server invokes the send support function for operations such as SELECT and FETCH statements that send the internal format of an opaque type to the client application.

The following sections describe the **fixed binary** and **var binary** data types.

Accessing a Fixed-Length Opaque Type

The **fixed binary** data type allows you to access a fixed-length opaque-type column in its internal format. Follow these steps to transfer the internal format of a fixed-length opaque-type column between the database server and the Informix ESQL/C application:

1. Declare a **fixed binary** host variable
2. Use the **fixed binary** host variable in an SQL statement to perform any select, insert, update, or delete operations on the internal format of the fixed-length opaque-type column.

The following sections describe each of these steps in more detail.

Declaring fixed binary Host Variables

Use the **fixed binary** data type to declare host variables that access the internal format of a fixed-length opaque data type. To declare a **fixed binary** host variable, use the following syntax.

```
            (1)
►►──────fixed binary──────────────────────────structure name──────────────►
                        └─'─opaque type─'─┘


        ┌─,─────────────┐
        │               │
►──────▼─variable name──┴──;─────────────────────────────────────────────►◄
```

Notes:

1   Informix extension

Element	Purpose	Restrictions	SQL Syntax
opaque type	Name of the fixed-length opaque data type whose internal format is to be stored in the **fixed binary** variable	Must already be defined in the database.	Identifier segment in the *IBM Informix Guide to SQL: Syntax*
structure name	Name of the C data structure that represents the internal format of the opaque data type	Must be defined in a header (**.h**) file that the source file includes. Must also match the data structure that the database server uses to represent the internal format of the opaque type.	Name must conform to language-specific rules for structure names.
variable name	Name of the ESQL/C variable to declare as a **fixed binary** variable		Name must conform to language-specific rules for variable names.

Important: A **fixed binary** host variable is only valid for a column of a fixed-length opaque data type. If the opaque data type is of varying length, use a **var binary** host variable. If you do not know the internal

data structure of a fixed-length opaque data type, you must also use a **var binary** host variable to access it.

For more information, see "Accessing a Varying-Length Opaque Type" on page 10-15.

To use a **fixed binary** host variable, you must reference a C data structure that maps the internal data structure of the opaque data type. You specify this C data structure as the *structure name* in the **fixed binary** declaration.

It is suggested that you create a C header file (**.h** file) for the C data structure that defines a fixed-length opaque data type. You can then include this header file in each Informix ESQL/C source file that uses **fixed binary** host variables to access the opaque data type.

For example, the following code fragment declares a **fixed binary** host variable called **my_circle** for the **circle** opaque data type:

```
#include <circle.h>   /* contains definition of circle_t */

EXEC SQL BEGIN DECLARE SECTION;
   fixed binary 'circle' circle_t my_circle;
EXEC SQL END DECLARE SECTION;
```

In this example, the **circle.h** header file contains the declaration for the **circle_t** structure (see Figure 10-1 on page 10-2), which is the internal data structure for the **circle** opaque type. The declaration for the **my_circle** host variable specifies *both* the name of the opaque data type, **circle**, and the name of its internal data structure, **circle_t**.

The Opaque Type: When you declare a **fixed binary** host variable, you must specify the *opaque type* as a quoted string.

Important: Both the single quote (') and the double quote (") are valid quote characters. However, the beginning quote and ending quote characters must match.

The *opaque type* name is optional; it affects the declaration as follows:

- When you omit *opaque type* from the **fixed binary** declaration, the database server attempts to identify the appropriate support functions to use when it sends the host variable to the database server for storage in the opaque-type column.

  You can use the **fixed binary** host variable to hold data for several different opaque types (as long as the database server is able to find the appropriate support functions).

- When you specify *opaque type* in the **fixed binary** declaration, the database server knows precisely which support functions to use to read and write to the opaque-type column.

  Using *opaque type* can make data conversion more efficient. In this case, however, the **fixed binary** host variable can hold data only for the specified *opaque type* data type.

You can declare several **fixed binary** variables in a single declaration. However, all variables must have the *same* opaque type, as the following declaration shows:

```
#include <shape.h>;

EXEC SQL BEGIN DECLARE SECTION;
    fixed binary 'shape' shape_t square1, square2;
EXEC SQL END DECLARE SECTION;
```

Using fixed binary Host Variables

Your Informix ESQL/C program must handle all manipulation of the internal data structure for the **fixed binary** host variable; it must explicitly allocate memory and assign field values.

Figure 10-7 shows how to use a **fixed binary** host variable to insert and select data in the **circle_col** column of the **circle_tab** table (see Figure 10-4 on page 10-3).

```
/* Include declaration for circle_t structure */
#include <circle.h>;

EXEC SQL BEGIN DECLARE SECTION;
    fixed binary 'circle' circle_t fbin_circle;
EXEC SQL END DECLARE SECTION;

/* Assign data to members of internal data structure */
fbin_circle.center.x = 1.00;
fbin_circle.center.y = 17.00;
fbin_circle.radius = 15.25;

/* Insert a new circle_tab row with a fixed binary host
 * variable */
EXEC SQL insert into circle_tab values (:fbin_circle);

/* Select a circle_tab row from into a fixed binary
 * host variable */
EXEC SQL select circle_col into :fbin_circle
   from circle_tab
   where radius(circle_col) = 15.25;
if ((fbin_circle.center.x == 1.00) &&
    (fbin_circle.center.y == 17.00))
   printf("coordinates = (%d, %d)\n",
      fbin_circle.center.x,   fbin_circle.center.y);
```

Figure 10-7. Accessing the Internal Format of the circle Opaque Data Type With a fixed binary Host Variable

Inserting from a fixed binary Host Variable: To insert the data that a **fixed binary** host variable contains into an opaque-type column, the code fragment in Figure 10-7 on page 10-14 takes the following steps:

1. Includes the definition of the internal structure of the **circle** opaque data type.

   The definition of the **circle_t** internal data structure, which Figure 10-1 on page 10-2 shows, must be available to your Informix ESQL/C program. Therefore, the code fragment includes the **circle.h** header file, which contains the definition of the **circle_t** structure.

2. Stores the data for the **fixed binary** host variable into the internal data structure, **circle_t**.

   The declaration of the **fixed binary** host variable associates the **circle_t** internal data structure with the **fbin_circle** host variable. The code fragment assigns a value to each member of the **circle_t** data structure.

3. Inserts the data that the **fbin_circle** host variable contains into the **circle_col** opaque-type column.

When the database server executes the INSERT statement, it calls the receive support function for the **circle** data type (**circle_rcv**) to perform any translation necessary between the internal format of the data that the Informix ESQL/C client application has sent (**circle_t**) and the internal format of the **circle** data type on disk.

To insert a null value into an opaque-type column with a **fixed binary** host variable, set an indicator variable to **-1**. The following code fragment inserts a null value into the **circle_col** column with the **fbin_circle** host variable:

```
#include <circle.h>;

EXEC SQL BEGIN DECLARE SECTION;
   fixed binary 'circle' circle_t fbin_circle;
   int circle_ind;
EXEC SQL END DECLARE SECTION;
:

circle_ind = -1;
EXEC SQL insert into circle_tab
   values (:fbin_circle:circle_ind);
```

For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

Selecting into a fixed binary Host Variable: To select the data that an opaque-type column contains into a **fixed binary** host variable, the code fragment in Figure 10-7 on page 10-14 takes the following steps:

1. Selects the data that the **circle_col** opaque-type column contains into the **fbin_circle** host variable.

   When the database server executes the SELECT statement, it calls the send support function for **circle** (**circle_snd**) to perform any translation necessary between the internal format that it retrieved from disk and the internal format that the Informix ESQL/C application uses. This SELECT statement also uses a user-defined function called **radius** (see Figure 10-2 on page 10-3) to extract the radius value from the opaque-type column.

2. Accesses the **circle** data from the **fixed binary** host variable.

   After the SELECT statement, the **fbin_circle** host variable contains data in the internal format of the **circle** data type. The code fragment obtains the value of the (*x,y*) coordinate from the members of the **circle_t** data structure.

When you select a null value from an opaque-type column into a **fixed binary** host variable, Informix ESQL/C sets any accompanying indicator variable to **-1**.

Accessing a Varying-Length Opaque Type

The **var binary** data type allows you to access the internal format of either of the following opaque data types:

- A fixed-length opaque-type column for which you do not have access to the C-structure of the internal format
- A varying-length opaque-type column

Follow these steps to transfer the internal format of either of these opaque data type columns between the database server and the Informix ESQL/C application:

1. Declare a **var binary** host variable

2. Use the **var binary** host variable in an SQL statement to perform any select, insert, update, or delete operations on the internal format of the opaque-type column.

The following sections describe each of these steps in more detail.

Declaring var binary Host Variables

To declare a **var binary** host variable, use the following syntax.
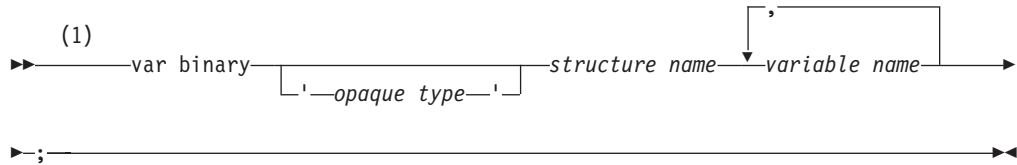
```
         (1)
►►──────var binary───────────────────────structure name───▼──variable name───────►
                      └─'──opaque type──'─┘

►──;────────────────────────────────────────────────────────────────────────────►◄
```

Notes:

1   Informix extension

Element	Purpose	Restrictions	SQL Syntax
opaque type	Name of the opaque data type whose internal format is to be stored in the **var binary** variable.	Must already be defined in the database	Identifier segment in the *IBM Informix Guide to SQL: Syntax*
variable name	Name of the ESQL/C variable to declare as a **var binary** variable		Name must conform to language-specific rules for variable names.

Figure 10-8 shows declarations for three **var binary** variables.

```
#include <shape.h>;
#include <image.h>;

EXEC SQL BEGIN DECLARE SECTION;
   var binary polygon1;
   var binary 'shape' polygon2, a_circle;
   var binary 'image' an_image;
EXEC SQL END DECLARE SECTION;
```

Figure 10-8. Sample var binary Host Variables

In the declaration of a **var binary** host variable, the name of the *opaque type* must be a quoted string.

Important: Both the single quote (') and the double quote (") are valid quote characters. However, the beginning quote and ending quote characters must match.

The *opaque type* name is optional; it affects the declaration as follows:

• When you omit *opaque type* from the **var binary** declaration, the database server attempts to identify the appropriate support functions to use when the application receives the internal data structure from the opaque-type column in a database.

The advantage of the omission of *opaque type* is that you can use the **var binary** host variable to hold data that was selected from several different opaque types (as long as the database server is able to find the appropriate support functions).

The disadvantage of the omission of *opaque type* is that host variables declared in this way cannot be used as parameters to user defined routines (UDRs).

- When you specify *opaque type* in the **var binary** declaration, the database server knows precisely which support functions to use when it sends the internal data structure to the database server for storage in the opaque-type column.

  The loss of ambiguity that the *opaque type* name provides can make data conversion more efficient. However, in this case, the **var binary** host variable can only hold data from the specified *opaque type* data type.

You can declare several **var binary** variables in a single declaration line. However, all variables must have the *same* opaque type, as Figure 10-8 on page 10-16 shows.

Using var binary Host Variables

In an Informix ESQL/C program, the varying-length C structure, **ifx_varlena_t**, stores a binary value for a **var binary** host variable. This data structure allows you to transfer binary data without knowing the exact structure of the internal format for the opaque data type. It provides a *data buffer* to hold the data for the associated **var binary** host variable.

Important: The **ifx_varlena_t** structure is an opaque structure to Informix ESQL/C programs. That is, you should not access its internal structure directly. The internal structure of **ifx_varlena_t** may change in future releases. Therefore, to create portable code, always use the Informix ESQL/C accessor functions for this structure to obtain and store values in the **ifx_varlena_t** structure. For a list of these Informix ESQL/C access functions, see "The lvarchar pointer and var binary library functions" on page 10-22.

This section uses a varying-length opaque data type called **image** to demonstrate how an Informix ESQL/C **var binary** host variable accesses an opaque data type. The image data type encapsulates an image such as a JPEG, GIF, or PPM file. If the image is less than 2 kilobytes, the data structure for the data type stores the image directly. However, if the image is greater than 2 kilobytes, the data structure stores a reference (an LO-pointer structure) to a smart large object that contains the image data. Figure 10-9 shows the internal data structure for the **image** data type in the database.

```
typedef struct
    {
    int      img_len;
    int      img_thresh;
    int      img_flags;
    union
        {
        ifx_lop_t      img_lobhandle;
        char      img_data[4];
        }

    } image_t;

typedef struct
    {
    point_t        center;
    double         radius;
    } circle_t;
```

Figure 10-9. Internal Data Structures for the image Opaque Data Type

Figure 10-10 shows the CREATE TABLE statement that creates a table called **image_tab** that has a column of type **image** and an image identifier.

```
CREATE TABLE image_tab
(
    image_id        integer not null primary key),
    image_col        image
);
```

Figure 10-10. Creating a Column of the image Opaque Data Type

Figure 10-11 shows how to use a **var binary** host variable to insert and select data in the **image_col** column of the **image_tab** table (see Figure 10-10 on page 10-18).

```
#include <image.h>;

EXEC SQL BEGIN DECLARE SECTION;
   var binary 'image' vbin_image;
EXEC SQL END DECLARE SECTION;

struct image_t user_image, *image_ptr;
int imgsz;

/* Load data into members of internal data structure
load_image(&user_image);
imgsz = getsize(&user_image);

/* Allocate memory for var binary data buffer */
ifx_var_flag(&vbin_image, 0);
ifx_var_alloc(&vbin_image, imgsz);

/* Assign data to data buffer of var binary host
 * variable */
ifx_var_setdata(&vbin_image, &user_image, imgsz);

/* Insert a new image_tab row with a var binary host
 * variable */
EXEC SQL insert into image_tab values (1, :vbin_image);

/* Deallocate image data buffer */
ifx_var_dealloc(&vbin_image);

/* Select an image_tab row from into a var binary
 * host variable */
ifx_var_flag(&vbin_image, 1);
EXEC SQL select image_col into :vbin_image
   from image_tab
   where image_id = 1;
image_ptr = (image_t *)ifx_var_getdata(&vbin_image);
unload_image(&user_image);
ifx_var_dealloc(&vbin_image);
```

Figure 10-11. Accessing the Internal Format of the image Opaque Data Type With a var binary Host Variable

For more information about the **ifx_var_flag()**, **ifx_var_alloc()**, **ifx_var_setdata()**, **ifx_var_getdata()**, and **ifx_var_dealloc()** functions, see "The lvarchar pointer and var binary library functions" on page 10-22.

Inserting from a var binary Host Variable: To insert the data that a **var binary** host variable contains into an opaque-type column, the code fragment in Figure 10-11 on page 10-19 takes the following steps:

1. Loads the image data from an external JPEG, GIF, or PPM file into the **image_t** internal data structure.

   The **load_image()** C routine loads the **user_image** structure from an external file. The definition of the **image_t** internal data structure, which Figure 10-9 on page 10-18 shows, must be available to your Informix ESQL/C program. Therefore, the code fragment includes the **image.h** header file, which defines the **image_t** structure.

   The **getsize()** C function is provided as part of the Informix ESQL/C support for the **image** opaque type; it returns the size of the **image_t** structure.

2. Allocates memory for the data buffer of the **var binary** host variable, **vbin_image**.

The **ifx_var_flag()** function with a flag value of 0 notifies Informix ESQL/C that the application will perform memory allocation for the **vbin_image** host variable. The **ifx_var_alloc()** function then allocates for the data buffer the number of bytes that the image data requires (**imgsz**).

3. Stores the **image_t** structure in the data buffer of the **vbin_image** host variable.

   The **ifx_var_setdata()** function saves the data that the **user_image** structure contains into the **vbin_image** data buffer. This function also requires the size of the data buffer, which the **getsize()** function has returned.

4. Inserts the data that the **vbin_image** data buffer contains into the **image_col** opaque-type column.

   When the database server executes the INSERT statement, it calls the receive support function for the **image** data type to perform any translation necessary between the internal format of the data that the Informix ESQL/C client application has sent (**image_t**) and the internal format of the **image** data type on disk.

5. Deallocates the data buffer of the **vbin_image** host variable.

   The **ifx_var_dealloc()** function deallocates the **vbin_image** data buffer.

To insert a null value into an opaque-type column with a **var binary** host variable, you can use either of the following methods:

- Set an indicator variable that is associated with a **var binary** host variable to -1.

  The following code fragment uses the **image_ind** indicator variable and the **vbin_image** host variable to insert a null value into the **circle_col** column:

```
#include <image.h>;

EXEC SQL BEGIN DECLARE SECTION;
   var binary 'image' vbin_image;
   int image_ind;
EXEC SQL END DECLARE SECTION;

image_ind = -1;
EXEC SQL insert into image_tab
   values (:vbin_image:image_ind);
```

  For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

- Use the **ifx_var_setnull()** function to set the data buffer of the **var binary** host variable to a null value.

  For the same **vbin_image** host variable, the following lines use the **ifx_var_setnull()** function to insert a null value into the **circle_col** column:

```
ifx_var_setnull(&vbin_image, 1);
EXEC SQL insert into image_tab values (:vbin_image);
```

  For more information about the **ifx_var_setnull()** function, see B-155.

Selecting into a var binary Host Variable: To select the data that an opaque-type column contains into a **var binary** host variable, the code fragment in Figure 10-11 on page 10-19 takes the following steps:

1. Allocates memory for the data buffer of the **var binary** host variable, **vbin_image**.

   The **ifx_var_flag()** function with a flag value of 1 notifies Informix ESQL/C that it is to allocate a new data buffer for the **vbin_image** host variable. (This data buffer had been deallocated after the INSERT statement completed.) Informix ESQL/C performs this allocation when it receives the data from the SELECT statement.

2. Selects the data that the **image_col** opaque-type column contains into the **vbin_image** data buffer.

   When the database server executes the SELECT statement, it calls the send support function for **image** to perform any translation necessary between the internal format of the **image** data type on disk and the internal format that the Informix ESQL/C client application has sent (**image_t**).

3. Stores the data that the data buffer of the **vbin_image** host variable contains in an **image_t** structure.

   After the SELECT statement, the data buffer of the **vbin_image** host variable contains data in the internal format of the **image** data type. The **ifx_var_getdata()** function returns the contents of this data buffer into an **image_t** data structure. Because the **ifx_var_getdata()** function returns the data buffer as a "**void ***" value, this call to **ifx_var_getdata()** casts this return value as a pointer to an **image_t** structure before it assigns it to the **image_ptr** variable.

4. Unloads the image data from the **image_t** internal data structure to an external JPEG, GIF, or PPM file.

   The **unload_image()** routine unloads the **user_image** structure to an external file.

5. Deallocates the data buffer of the **vbin_image** host variable.

   The **ifx_var_dealloc()** function deallocates the **vbin_image** data buffer. You must explicitly deallocate the data buffer even when Informix ESQL/C allocated it for you.

To check for a null value from an opaque-type column with a **var binary** host variable, you can use either of the following methods:

- Check an indicator variable that is associated with a **var binary** host variable for a value of **-1**.

  The following code fragment uses the **image_ind** indicator variable and the **vbin_image** host variable to check for a null value from the **circle_col** column:

  ```
  #include <image.h>;

  EXEC SQL BEGIN DECLARE SECTION;
      var binary 'image' vbin_image;
      int image_ind;
  EXEC SQL END DECLARE SECTION;

  EXEC SQL select image_col into :vbin_image:image_ind
      from image_tab
      where image_id = 1;
  if (image_ind == -1)
      :
  ```

  For more information about indicator variables, see "Using Indicator Variables" on page 1-21.

- Use the **ifx_var_isnull()** function to check the data buffer of the **var binary** host variable for a null value.

  For the same **vbin_image** host variable, the following lines use the **ifx_var_isnull()** function to check for a null value in the **image_col** column:

  ```
  EXEC SQL select image_col into :vbin_image
      from image_tab
      where image_id = 1;
  if (ifx_var_isnull(&vbin_image) == 1)
      :
  ```

  For more information about the **ifx_var_isnull()** function, see B-153.

The lvarchar pointer and var binary library functions

The following library functions are available in Informix ESQL/C to access the data buffer of an **lvarchar pointer** or **var binary** host variable.

Function Name	Purpose	See
ifx_var_alloc()	Allocates memory for the data buffer.	B-148
ifx_var_dealloc()	Deallocates memory for the data buffer.	B-149
ifx_var_flag()	Determines whether ESQL/C or the application handles memory allocation for the data buffer.	B-150
ifx_var_getdata()	Returns the contents of the data buffer.	B-151
ifx_var_getlen()	Returns the length of the data buffer.	B-152
ifx_var_isnull()	Checks whether the data in the data buffer is null.	B-153
ifx_var_setdata()	Sets the data for the data buffer.	B-154
ifx_var_setlen()	Sets the length of the data buffer.	B-154
ifx_var_setnull()	Sets the data in the data buffer to a null value.	B-155

These **lvarchar** pointer and **var binary** functions are defined in the **sqlhdr.h** header file so you do not need to include a special header file in your Informix ESQL/C programs that use them.

Accessing Predefined Opaque Data Types

IBM Informix implements several built-in data types as *predefined opaque data types*. These data types are opaque data types for which support functions and the database definition are provided. For example, the smart-large-object data types, CLOB and BLOB, as an opaque data type called **clob** and **blob** are implemented. Informix ESQL/C uses the **ifx_lo_t** structure, called an LO-pointer, to access the smart large objects. This structure is defined in the **locator.h** header file.

Therefore, you declare Informix ESQL/C host variables for database columns of type CLOB or BLOB as a **fixed binary** host variable, as follows:

```
EXEC SQL include locator;
  :

EXEC SQL BEGIN DECLARE SECTION;
   fixed binary 'clob' ifx_lo_t clob_loptr;
   fixed binary 'blob' ifx_lo_t blob_loptr;
EXEC SQL END DECLARE SECTION;
  :

EXEC SQL select blobcol into :blob_loptr from tab1;
```

For more information about how to access smart large objects, see Chapter 8, "Working with smart large objects," on page 8-1.

Part 2. Database server communication

Chapter 11. Exception handling

Proper database management requires that you know whether the database server successfully processes your SQL statements as you intend. If a query fails and you do not know it, you might display meaningless data to the user. A more serious consequence might be that you update a customer account to show a payment of $100, and the update fails without your knowledge. The account is now incorrect.

To handle such error situations, your Informix ESQL/C program must check that every SQL statement executes as you intend. These topics describe the following exception-handling information:

- How to interpret the diagnostic information that the database server presents after it executes an SQL statement
- How to use the **SQLSTATE** variable and the GET DIAGNOSTICS statement to check for runtime errors and warnings that your Informix ESQL/C program might generate
- How to use the **SQLCODE** variable and the SQL Communications Area (**sqlca**) to check for runtime errors and warnings that your Informix ESQL/C program might generate
- How to choose an exception-handling strategy that consistently handles errors and warnings in your Informix ESQL/C programs
- How to use the **rgetlmsg()** and **rgetmsg()** library functions to retrieve the message text that is associated with a given Informix error number

The end of these topics present an annotated example program that is called **getdiag**. The **getdiag** sample program demonstrates how to handle exceptions with the **SQLSTATE** variable and the GET DIAGNOSTICS statement.

Obtain diagnostic information after an SQL statement

After your Informix ESQL/C program executes an SQL statement, the database server returns information about the success of the statement. This section summarizes the following information:

- The types of diagnostic information that are available to an Informix ESQL/C program
- The two methods that your Informix ESQL/C program can use to obtain diagnostic information

Types of Diagnostic Information

The database server can return the following types of diagnostic information:

- *Database exceptions* are conditions that the database server returns to describe how successful the execution of the SQL statement was.
- *Descriptive information*, such as the DESCRIBE and GET DIAGNOSTICS statements can provide about certain SQL statements.

Types of Database Exceptions
When the database server executes an SQL statement, it can return one of four types of database exceptions to the application program:

- Success

The SQL statement executed successfully. When a statement that might return data into host variables executes, a success condition means that the statement has returned the data and that the program can access it through the host variables.

- Success, but warning generated

  A *warning* is a condition that does not prevent successful execution of an SQL statement; however, the effect of the statement is limited and the statement might not produce the expected results. A warning can also provide additional information about the executed statement.

- Success, but no rows found

  The SQL statement executed without errors, with the following exceptions:
  - No rows matched the search criteria (the NOT FOUND condition).
  - The statement did not operate on a row (the END OF DATA condition).

- Error

  The SQL statement did not execute successfully and made no change to the database. Runtime errors can occur at the following levels:
  - Hardware errors include controller failure, bad sector on disk, and so on.
  - Kernel errors include file-table overflow, insufficient semaphores, and so on.
  - Access-method errors include duplicated index keys, SQL null inserted into non-null columns, and so on.
  - Parser errors include invalid syntax, unknown objects, invalid statements, and so on.
  - Application errors include user or lock-table overflow, and so on.

Descriptive Information

The following SQL statements can return information about SQL statements:

- A DESCRIBE statement returns information about a prepared SQL statement. This information is useful when you execute dynamic SQL. For more information, see "SQLCODE After a DESCRIBE Statement" on page 11-20.

- A GET DIAGNOSTICS statement, when you call it after you have established a connection to a database environment, can return the name of the database server and the connection. For more information, see "Using GET DIAGNOSTICS" on page 11-4.

The *IBM Informix Guide to SQL: Syntax* fully describes these two statements.

Types of Status Variables

The following methods obtain diagnostic information about the outcome of an SQL statement:

- Access the **SQLSTATE** variable, a five-character string that contains status values that conform to the ANSI and X/Open standards

- Access the **SQLCODE** variable, an **int4** integer that contains Informix-specific status values

When you create applications that must conform to either the ANSI or X/Open standard, use the **SQLSTATE** variable as your primary exception-handling method.

Exception Handling with SQLSTATE

It is recommended that you obtain diagnostic information about SQL statements with the **SQLSTATE** variable and the GET DIAGNOSTICS statement.

> **Important:** **SQLSTATE** is a more effective way to detect and handle error messages than the **SQLCODE** variable because **SQLSTATE** supports multiple exceptions. **SQLSTATE** is also more portable because it conforms to ANSI and X/Open standards. Informix ESQL/C supports the **sqlca** structure and **SQLCODE** for compatibility with earlier versions and for Informix-specific exceptions. For more information about the **sqlca** structure, see "Exception Handling with the sqlca Structure" on page 11-17

After the database server executes an SQL statement, it sets **SQLSTATE** with a value that indicates the success or failure of the statement. From this value, your program can determine if it needs to perform further diagnostics. If **SQLSTATE** indicates a problem, you can use the GET DIAGNOSTICS statement to obtain more information.

This section describes how to use the **SQLSTATE** variable and the GET DIAGNOSTICS statement to perform exception handling. It describes the following topics:

- Using the GET DIAGNOSTICS statement to access fields of the diagnostics area
- Understanding the format of the **SQLSTATE** values
- Using **SQLSTATE** to check for the different types of exceptions

Using GET DIAGNOSTICS

This section briefly summarizes how to use the GET DIAGNOSTICS statement within an Informix ESQL/C program. For a full description of the GET DIAGNOSTICS statement, see the *IBM Informix Guide to SQL: Syntax*.

The GET DIAGNOSTICS statement returns information that is held in the fields of the *diagnostics area*. The diagnostics area is an internal structure that the database server updates after it executes an SQL statement. Each application has one diagnostics area. Although GET DIAGNOSTICS accesses the diagnostics area, it never changes the contents of this area.

To access a field in the diagnostics area, supply a host variable to hold the value and the field keyword to specify the field that you want to access:

```
:host_var = FIELD_NAME
```

Make sure that the data types of the host variable and the diagnostics field are compatible.

The fields of the diagnostics area fall into two categories:

- Statement information describes the overall result of the SQL statement, in particular the number of rows that it has modified and the number of exceptions that result.
- *Exception information* describes individual exceptions that result from the SQL statement.

Statement Information

The GET DIAGNOSTICS statement returns information about the most-recently executed SQL statement. This form of the GET DIAGNOSTICS statement has the following general syntax:

```
EXEC SQL get diagnostics statement_fields;
```

Table 11-1 summarizes the *statement_fields* of the diagnostics area.

Table 11-1. Statement Information from the GET DIAGNOSTICS Statement

Field-Name Keyword	ESQL/C Data Type	Description
NUMBER	**mint**	This field holds the number of exceptions that the diagnostics area contains for the most-recently executed SQL statement. NUMBER is in the range of 1 to 35,000. Even when an SQL statement is successful, the diagnostics area contains one exception.
MORE	**char[2]**	This field holds either an N or a Y (plus a null terminator). An N character indicates that the diagnostics area contains all of the available exception information. A Y character indicates that the database server has detected more exceptions than it can store in the diagnostics area. At present, the database server always returns an N because the database server can store all exceptions.
ROW_COUNT	**mint**	When the SQL statement is an INSERT, UPDATE, or DELETE, this field holds a numeric value that specifies the number of rows that the statement has inserted, updated, or deleted. ROW_COUNT is in the range of 0 to 999,999,999. For any other SQL statement, the value of ROW_COUNT is undefined.

Figure 11-1 shows a GET DIAGNOSTICS statement that retrieves statement information for a CREATE TABLE statement into the host variables **:exception_count** and **:overflow**.

```
EXEC SQL BEGIN DECLARE SECTION;
    mint exception_count;
    char overflow[2];
EXEC SQL END DECLARE SECTION;
    .
    .

EXEC SQL create database db;

EXEC SQL create table tab1 (col1 integer);
EXEC SQL get diagnostics :exception_count = NUMBER,
    :overflow = MORE;
```

Figure 11-1. Using GET DIAGNOSTICS to Return Statement Information

Use the statement information to determine how many exceptions the most-recently executed SQL statement has generated. For more information, see "Multiple Exceptions" on page 11-15.

For more information about the statement fields of the diagnostics area, see "The Statement Clause" in the GET DIAGNOSTICS statement in the *IBM Informix Guide to SQL: Syntax*.

Exception Information

The GET DIAGNOSTICS statement also returns information about the exceptions that the most-recently executed SQL statement has generated. Each exception has an exception number. To obtain information about a particular exception, use the EXCEPTION clause of the GET DIAGNOSTICS statement, as follows:

```
EXEC SQL get diagnostics exception except_num exception_fields;
```

The *except_num* argument can be a literal number or a host variable. An *except_num* of one (1) corresponds to the **SQLSTATE** value that the most-recently executed SQL statement sets. After this first exception, the order in which the database server fills the diagnostics area with exception values is not predetermined. For more information, see "Multiple Exceptions" on page 11-15.

Table 11-2 summarizes the *exception_fields* information of the diagnostics area.

Table 11-2. Exception Information from the GET DIAGNOSTICS Statement

Field Name Keyword	ESQL/C Data Type	Description
RETURNED_SQLSTATE	**char[6]**	This field holds the **SQLSTATE** value that describes the current exception. For information about the values of this field, see "Using the SQLSTATE Variable" on page 11-7.
INFORMIX_SQLCODE	**int4**	This field holds the Informix-specific status code. This code is also available in the global **SQLCODE** variable. For more information, see "Using the SQLCODE Variable" on page 11-19.
CLASS_ORIGIN	**char[255]**	This field holds a variable-length character string that defines the source of the class portion of **SQLSTATE**. If Informix defines the class code, the value is "IX000". If the International Standards Organization (ISO) defines the class code, the value of CLASS_ORIGIN is "ISO 9075". If a user-defined routine has defined the message text of the exception, the value of CLASS_ORIGIN is "U0001".
SUBCLASS_ORIGIN	**char[255]**	This field holds a variable-length character string that contains the source of the subclass portion of **SQLSTATE**. If ISO defines the subclass, the value of SUBCLASS_ORIGIN is "ISO 9075". If Informix defines the subclass, the value is "IX000". If a user-defined routine has defined the message text of the exception, the value is "U0001".
MESSAGE_TEXT	**char[8191]**	This field holds a variable-length character string that contains the message text to describe this exception. This field can also contain the message text for any ISAM exceptions or a user-defined message from a user-defined routine.
MESSAGE_LENGTH	**mint**	This field holds the number of characters that are in the text of the MESSAGE_TEXT string.
SERVER_NAME	**char[255]**	This field holds a variable-length character string that holds the name of the database server that is associated with the actions of a CONNECT or DATABASE statement. This field is blank when no current connection exists. For more information about the SERVER_NAME field, see "Identifying an Explicit Connection" on page 12-20.
CONNECTION_NAME	**char[255]**	This field holds a variable-length character string that holds the name of the connection that is associated with the actions of a CONNECT or SET CONNECTION statement. This field is blank when no current connection or no explicit connection exists. Otherwise, it contains the name of the last successfully established connection. For more information about the CONNECTION_NAME field, see "Identifying an Explicit Connection" on page 12-20.

Use the exception information to save detailed information about an exception. The code fragment in Figure 11-2 retrieves exception information about the first exception of a CREATE TABLE statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   char class_origin_val[255];
   char subclass_origin_val[255];
   char message_text_val[8191];
   mint messlength_val;
EXEC SQL END DECLARE SECTION;

EXEC SQL create database db;

EXEC SQL create table tab1 (col1 integer);
EXEC SQL get diagnostics exception 1
   :class_origin_val = CLASS_ORIGIN,
   :subclass_origin_val = SUBCLASS_ORIGIN,
   :message_text_val = MESSAGE_TEXT,
   :messlength_val = MESSAGE_LENGTH;
```

Figure 11-2. Example of Using GET DIAGNOSTICS to Return Exception Information

For more information about the exception fields, see the GET DIAGNOSTICS statement in the *IBM Informix Guide to SQL: Syntax*.

Using the SQLSTATE Variable

SQLSTATE is a five-character string that the database server sets after it executes each SQL statement. The Informix ESQL/C header file, **sqlca.h**, declares **SQLSTATE** as a global variable. Since the Informix ESQL/C preprocessor automatically includes **sqlca.h** in an Informix ESQL/C program, you do not need to declare **SQLSTATE**.

After the database server executes an SQL statement, the database server automatically updates the **SQLSTATE** variable as follows:

- The database server stores the exception value in the RETURNED_SQLSTATE field of the diagnostics area.
- Informix ESQL/C copies the value of the RETURNED_SQLSTATE field to the global **SQLSTATE** variable.

These updates to the **SQLSTATE** variable are equivalent to the execution of the following GET DIAGNOSTICS statement immediately after an SQL statement:

```
EXEC SQL get diagnostics exception 1 :SQLSTATE = RETURNED_SQLSTATE;
```

Tip: At runtime, *Informix ESQL/C* automatically copies the value of the RETURNED_SQLSTATE field into the global **SQLSTATE** variable. Therefore, you do not usually need to access the RETURNED_SQLSTATE field directly. For more information, see "Multiple Exceptions" on page 11-15.

The value in **SQLSTATE** is the status of the most-recently executed SQL statement *before* the GET DIAGNOSTICS statement executed. If the database server encounters an error when it executes the GET DIAGNOSTICS statement, it sets **SQLSTATE** to "IX001" and sets **SQLCODE** (and **sqlca.sqlcode**) to the value of the error number that corresponds to the error; the contents of the diagnostics area are undefined.

The **SQLSTATE** variable holds the ANSI-defined value for the exception. Each **SQLSTATE** value has an associated Informix-specific status code. You can obtain the value of this Informix-specific status code from either of the following items:

- The **INFORMIX_SQLCODE** field of the diagnostics area
- The **SQLCODE** variable ("Using the SQLCODE Variable" on page 11-19)

The next two sections provide the following information about the format of the **SQLSTATE** value:

- How to use the class and subclass portions of the **SQLSTATE** value
- The list of exception values that **SQLSTATE** can store

Class and Subclass Codes

To determine the success of an SQL statement, your Informix ESQL/C program must be able to interpret the value in the **SQLSTATE** variable. **SQLSTATE** consists of a two-character *class code* and a three-character *subclass* code. In Figure 11-3, IX is the class code and 000 is the subclass code. The value "IX000" indicates an Informix-specific error.
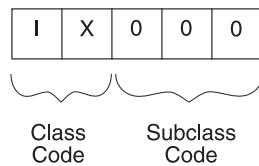
Figure 11-3. The Structure of the SQLSTATE Code with the Value IX000

SQLSTATE can contain *only* digits and capital letters. The class code is unique but the subclass code is not. The meaning of the subclass code depends on the associated class code. The initial character of the class code indicates the source of the exception code, which Table 11-3 summarizes.

Table 11-3. Initial SQLSTATE Class-Code Values

Initial Class-Code Value	Source of Exception Code	Notes
0 to 4 A to H	X/Open and ANSI/ISO	The associated subclass codes also begin in the range 0 to 4 or A to H.
5 to 9	Defined by the implementation	Subclass codes are also defined by the implementation.
I to Z	IBM Informix INFORMIX-ESQL/C	Any of the Informix-specific error messages (those that the X/Open or ANSI/ISO reserved range does *not* support) have an **SQLSTATE** value of "IX000". If a user-defined routine returns an error message was defined by the routine, the **SQLSTATE** value is "U0001".

List of SQLSTATE Class Codes

Table 11-4 lists the valid **SQLSTATE** class and subclass values. This figure lists the first entry for each class code in bold.

Table 11-4. Class and Subclass Codes for SQLSTATE

Class	Subclass	Meaning
00	000	**Success**

Table 11-4. Class and Subclass Codes for SQLSTATE (continued)

Class	Subclass	Meaning
01	000	**Success with warning**
01	002	Disconnect error—transaction rolled back
01	003	Null value eliminated in set function
01	004	String data, right truncation
01	005	Insufficient item descriptor areas
01	006	Privilege not revoked
01	007	Privilege not granted
01	I01	**Database has transactions**
01	I03	ANSI-compliant database selected
01	I04	Database on any database server other than IBM Informix SE selected
01	I05	Float to decimal conversion used
01	I06	Informix extension to ANSI-compliant standard syntax
01	I07	After a DESCRIBE, a prepared UPDATE/DELETE statement does not have a WHERE clause
01	I08	An ANSI keyword was used as cursor name
01	I09	Number of items in select list is not equal to number of items in INTO list
01	I10	Database server is running in secondary mode
01	I11	DATASKIP feature is turned on
01	U01	User-defined warning returned by a user-defined routine
02	000	**No data found or End of data reached**
07	000	**Dynamic SQL error**
07	001	USING clause does not match dynamic parameters
07	002	USING clause does not match target specifications
07	003	Cursor specification cannot be executed
07	004	USING clause is required for dynamic parameters
07	005	Prepared statement is not a cursor specification
07	006	Restricted data type attribute violation
07	008	Invalid descriptor count
07	009	Invalid descriptor index

Table 11-4. Class and Subclass Codes for SQLSTATE (continued)

Class	Subclass	Meaning
08	000	**Connection exception**
08	001	Database server rejected the connection
08	002	Connection name in use
08	003	Connection does not exist
08	004	Client unable to establish connection
08	006	Transaction rolled back
08	007	Transaction state unknown
08	S01	Communication failure
0A	000	**Feature not supported**
0A	001	Multiple database server transactions
21	000	**Cardinality violation**
21	S01	Insert value list does not match column list
21	S02	Degree of derived table does not match column list
22	000	**Data exception**
22	001	String data, right truncation
22	002	Null value, no indicator parameter
22	003	Numeric value out of range
22	005	Error in assignment
22	012	Division by zero
22	019	Invalid escape character
22	024	Unterminated string
22	025	Invalid escape sequence
22	027	Data exception trim error
23	000	**Integrity-constraint violation**
24	000	**Invalid cursor state**
25	000	**Invalid transaction state**
2B	000	**Dependent privilege descriptors still exist**
2D	000	**Invalid transaction termination**
26	000	**Invalid SQL statement identifier**
2E	000	**Invalid connection name**
28	000	**Invalid user-authorization specification**
33	000	**Invalid SQL descriptor name**
34	000	**Invalid cursor name**
35	000	**Invalid exception number**

Table 11-4. Class and Subclass Codes for SQLSTATE (continued)

Class	Subclass	Meaning
37	000	**Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE**
3C	000	**Duplicate cursor name**
4040	000 003	**Transaction rollback**Statement completion unknown
42	000	**Syntax error or access violation**
S0	000	**Invalid name**
S0	001	Base table or view table already exists
S0	002	Base table not found
S0	011	Index already exists
S0	021	Column already exists
S1	001	**Memory-allocation error message**
IX	000	**Informix reserved error message**
IX	001	**GET DIAGNOSTICS statement failed**
U0	001	**User-defined error returned by a user-defined routine**

The ANSI or X/Open standards define all **SQLSTATE** values *except* the following:

- A "IX000" runtime error indicates an Informix-specific error message.
- A "IX001" runtime error indicates the GET DIAGNOSTiCS statement failed.
- A "U0001" runtime error indicates a user-defined error message.
- The "01Ixx" warnings indicate Informix-specific warnings.
- The "01U01" warning indicates a user-defined warning message.

For more information about non-standard error values, see "Runtime Errors in SQLSTATE" on page 11-15. For more information about non-standard warning values, see "Warnings in SQLSTATE" on page 11-13.

Checking for Exceptions with SQLSTATE

After an SQL statement executes, the **SQLSTATE** value can indicate one of the four conditions that Table 11-5 shows.

Table 11-5. Exceptions That SQLSTATE Returns

Exception Condition	SQLSTATE Value
Success	"00000"
Success, but no rows found	"02000"
Success, but warnings generated	Class code = "01"
	Subclass code = "000" to "006" (for ANSI and X/Open warnings)
	Subclass code = "I01" to "I11" (for Informix-specific warnings)
	Subclass code = "U01" (for user-defined warnings)
Failure, runtime error generated	Class code > "02" (for ANSI and X/Open errors)
	Class code = "IX" (for Informix-specific errors)
	Class code = "U0" (for user-defined errors)

For a general introduction to these four conditions, see "Types of Database Exceptions" on page 11-2. To determine the cause of an exception in **SQLSTATE**, use the GET DIAGNOSTICS statement.

To determine the cause of an exception in SQLSTATE:

1. Use GET DIAGNOSTICS to obtain the statement information such as the number of exceptions that the database server has generated.
2. For each exception, use the EXCEPTION clause of GET DIAGNOSTICS to obtain detailed information about the exception.

The following sections discuss how **SQLSTATE** indicates each condition.

Success in SQLSTATE

When the database server executes an SQL statement successfully, it sets **SQLSTATE** to "00000" (class = "00", subclass = "000"). To check for successful execution, your code needs to verify only the first two characters of **SQLSTATE**.

Tip: After a CONNECT, SET CONNECTION, DATABASE, CREATE DATABASE, or START DATABASE statement, the **SQLSTATE** variable has a class value of "01" and an Informix-specific subclass value to provide information about the database and connection. For more information, see Table 11-7 on page 11-14.

The **getdiag** sample program in 11-30 uses the **sqlstate_err()** function to compare the first two characters of **SQLSTATE** with the string "00" to check for successful execution of an SQL statement. The **sqlstate_exception()** function shown in Figure 11-8 on page 11-27 checks for a success in **SQLSTATE** with the system **strncmp()** function.

NOT FOUND in SQLSTATE

When a SELECT or FETCH statement encounters NOT FOUND (or END OF DATA), the database server sets **SQLSTATE** to "02000" (class = "02"). Table 11-6 lists the conditions that cause SQL statements to yield NOT FOUND.

Table 11-6. SQLSTATE Values That Are Set When SQL Statements Do Not Return Any Rows

SQL Statement That Generates the Indicated SQLSTATE Result	Result for ANSI-Compliant Database	Result for non-ANSI-Compliant Database
FETCH statement: the last qualifying row has already been returned (the end of data was reached).	"02000"	"02000"
SELECT statement: no rows match the SELECT criteria.	"02000"	"02000"
DELETE and DELETE...WHERE statement (not part of multistatement PREPARE): no rows match the DELETE criteria.	"02000"	"00000"
INSERT INTO *tablename* SELECT statement (not part of multistatement PREPARE): no rows match the SELECT criteria.	"02000"	"00000"
SELECT... INTO TEMP statement (not part of multistatement PREPARE): no rows match the SELECT criteria.	"02000"	"00000"
UPDATE and UPDATE...WHERE statement (not part of multistatement PREPARE): no rows match the UPDATE criteria.	"02000"	"00000"

Table 11-6 on page 11-13 shows that the value that the NOT FOUND condition generates depends, in some cases, on whether the database is ANSI compliant.

To check for the NOT FOUND condition, your code needs to verify only the class code of **SQLSTATE**. The subclass code is always "000". The **getdiag** sample program in 11-30 uses the **sqlstate_err()** function to perform exception handling. To check for a warning in an SQL statement, **sqlstate_err()** compares the first two characters of **SQLSTATE** with the string "02".

Warnings in SQLSTATE

When the database server executes an SQL statement successfully, but encounters a warning condition, it sets the class code of **SQLSTATE** to "01". The subclass code then indicates the cause of the warning. This warning can be either of the following types:

- An ANSI or X/Open warning message has a subclass code in the range "000" to "006".

  The **CLASS_ORIGIN** and **SUBCLASS_ORIGIN** exception fields of the diagnostics area have a value of "ISO 9075" to indicate ANSI or X/Open as the source of the warning.

- An Informix-specific warning message has a subclass code in the range "I01" to "I11" (see Table 11-7).

  The **CLASS_ORIGIN** and **SUBCLASS_ORIGIN** exception fields of the diagnostics area have a value of "IX000" to indicate an Informix-specific exception as the source of the warning.

- A user-defined warning message from a user-defined routine has a subclass code of "U01".

  The **CLASS_ORIGIN** and **SUBCLASS_ORIGIN** exception fields of the diagnostics area have a value of "U0001" to indicate a user-defined routine as the source of the warning.

Table 11-7 lists the Informix-specific warning messages and the SQL statements and conditions that generate the warning.

Table 11-7. SQL Statements That Set an Informix-Specific Warning for a Given Condition

Warning Value	SQL Statement	Warning Condition
"01I01"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	Your application opened a database that uses transactions.
"01I03"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	Your application opened an ANSI-compliant database.
"01I04"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	Your application opened a database that the IBM Informix manages.
"01I05"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	Your application opened a database that is on a host database server that requires float-to-decimal conversion for FLOAT columns (or smallfloat-to-decimal conversions for SMALLFLOAT columns).
"01I06"	All statements	The statement just executed contains an Informix extension to SQL (*only* when the DBANSIWARN environment variable is set).
"01I07"	PREPARE DESCRIBE	A prepared UPDATE or DELETE statement has no WHERE clause. The operation will affect all rows of the table.
"01I09"	FETCH SELECT...INTO EXECUTE...INTO	The number of items in the select list does not equal the number of host variables in the INTO clause.
"01I10"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	The database server is currently running in secondary mode. The database server is a secondary server in a data-replication pair; therefore, the database server is available only for read operations.
"01I11"	Other statements (when your application activates the DATASKIP feature)	A data fragment (a dbspace) was skipped during query processing.

For a list of the ANSI and X/Open warning messages, see "List of SQLSTATE Class Codes" on page 11-8.

To check for a warning, your code only needs to verify the first two characters of **SQLSTATE**. However, to identify the particular warning, you need to examine the subclass code. You might also want to use the GET DIAGNOSTICS statement to obtain the warning message from the **MESSAGE_TEXT** field.

For example, the block of code in Figure 11-4 determines what kind of database a CONNECT statement has opened.

```
int trans_db, ansi_db, online_db = 0;
 .
 .
 .

msg = "CONNECT stmt";
EXEC SQL connect to 'stores7';
if(!strncmp(SQLSTATE, "02", 2)) /* < 0 is an error */
   err_chk(msg);
if (!strncmp(SQLSTATE, "01", 2))
   {
   if (!strncmp(SQLSTATE[2], "I01", 3))
      trans_db = 1;
   if (!strncmp(SQLSTATE[2], "I03", 3))
      ansi_db = 1;
   if (!strncmp(SQLSTATE[2], "I04", 3))
      online_db = 1;
   }
```

Figure 11-4. Using SQLSTATE to Check for Informix-Specific Warnings

The code fragment in Figure 11-4 checks **SQLSTATE** with the system **strncmp()**
function. The **getdiag** sample program (11-30) uses the **sqlstate_err()** function to
check the success of an SQL statement by comparing the first two characters of
SQLSTATE with the string "01". For more information about the values of
SQLSTATE that the CONNECT, CREATE DATABASE, DATABASE, and SET
CONNECTION statements set, see "Determining Features of the Database Server"
on page 12-18.

Runtime Errors in SQLSTATE

When an SQL statement results in a runtime error, the database server stores a
value in **SQLSTATE** whose class code is greater than "02". The actual class and
subclass codes identify the particular error. Table 11-4 on page 11-8 lists the class
and subclass codes for **SQLSTATE**. To retrieve the error message text, use the
MESSAGE_TEXT field of the GET DIAGNOSTICS statement. The
CLASS_ORIGIN and **SUBCLASS_ORIGIN** exception fields have a value of "ISO
9075" to indicate the source of the error.

If the SQL statement generates an error that the ANSI or X/Open standards do *not*
support, **SQLSTATE** might contain either of the following values:

- An **SQLSTATE** value of "IX000" indicates an Informix-specific error.

  The **SQLCODE** variable contains the Informix-specific error code, and the
  MESSAGE_TEXT field contains the error message text and any ISAM message
  text. The **CLASS_ORIGIN** and **SUBCLASS_ORIGIN** exception fields have a
  value of "IX000" to indicate the source of the error.

- An **SQLSTATE** value of "U0001" indicates a user-defined error message from a
  user-defined routine.

  The **MESSAGE_TEXT** field contains the error message text. The
  CLASS_ORIGIN and **SUBCLASS_ORIGIN** exception fields have a value of
  "U0001" to indicate the source of the error.

GET DIAGNOSTICS Failure: If the GET Diagnostics statement fails, SQLState
contains a value of ix001. No other failure returns this value. The sqlcode indicates
the specific error that caused the failure.

Multiple Exceptions

The database server can generate multiple exceptions for a single SQL statement. A
significant advantage of the GET DIAGNOSTICS statement is its ability to report
multiple exception conditions.

To find out how many exceptions the database server has reported for an SQL statement, retrieve the value of the **NUMBER** field from the statement information of the diagnostics area. The following GET DIAGNOSTICS statement retrieves the number of exceptions that the database server generated and stores the number in the **:exception_num** host variable.

```
EXEC SQL get diagnostics :exception_num = NUMBER;
```

Once you know the number of exceptions that occurred, you can initiate a loop to report each of them. Execute GET DIAGNOSTICS within this loop and use the number of exceptions to control the loop. Figure 11-5 illustrates one way to retrieve and report multiple exception conditions after an SQL statement.

```
EXEC SQL get diagnostics :exception_count = NUMBER,
    :overflow = MORE;
printf("NUMBER: %d\n", exception_count);
printf("MORE : %s\n", overflow);
for (i = 1; i <= exception_count; i++)
    {
    EXEC SQL get diagnostics  exception :i
        :sqlstate = RETURNED_SQLSTATE,
        :class = CLASS_ORIGIN, :subclass = SUBCLASS_ORIGIN,
        :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;

        printf("SQLSTATE: %s\n",sqlstate);
        printf("CLASS ORIGIN: %s\n",class);
        printf("SUBCLASS ORIGIN: %s\n",subclass);
        message[messlen] ='\0'; /* terminate the string. */
        printf("TEXT: %s\n",message);
        printf("MESSAGE LENGTH: %d\n",messlen);
    }
```

Figure 11-5. Reporting Multiple Exception Conditions

Do not confuse the **RETURNED_SQLSTATE** value with the **SQLSTATE** global variable. The **SQLSTATE** variable provides a general status value for the most-recently executed SQL statement. The **RETURNED_SQLSTATE** value is associated with one particular exception that the database server has encountered. For the first exception, **SQLSTATE** and **RETURNED_SQLSTATE** have the same value. However, for multiple exceptions, you must access **RETURNED_SQLSTATE** for *each* exception.

To define a host variable in your application that receives the **RETURNED_SQLSTATE** value, you must define it as a character array with a length of six (five for the field plus one for the null terminator). You can assign this variable whatever name you wish.

The following statements define such a host variable and assign it the name **sql_state**:

```
EXEC SQL BEGIN DECLARE SECTION;
    char sql_state[6];
EXEC SQL END DECLARE SECTION;
```

A database system that is compliant with X/Open standards must report any X/Open exceptions before it reports any Informix-specific errors or warnings. Beyond this, however, the database server does not report the exceptions in any particular order. The **getdiag** sample program (11-30) includes the **disp_sqlstate_err()** function to display multiple exceptions.

Exception Handling with the sqlca Structure

An alternative way to obtain diagnostic information is through the *SQL Communications Area*. When an SQL statement executes, the database server automatically returns information about the success or failure of the statement in a C structure that is called **sqlca**. To obtain exception information, your Informix ESQL/C program can access the **sqlca** structure or the **SQLCODE** variable as follows:

- **The sqlca structure.** You can use C statements to obtain additional exception information. You can also obtain information relevant to performance or the nature of the data that is handled. For some statements, the **sqlca** structure contains warnings.

- **The SQLCODE variable directly.** You can obtain the status code of the most-recently executed SQL statement. **SQLCODE** holds an Informix-specific error code, which is copied from the **sqlca.sqlcode** field.

Important: Informix ESQL/C supports the **sqlca** structure for compatibility with earlier versions. It is recommended, however, that new applications use the **SQLSTATE** variable with the GET DIAGNOSTICS statement to perform exception checking. This method conforms to X/Open and ANSI SQL standards and supports multiple exceptions.

The next three sections describe how to use the **SQLCODE** variable and the **sqlca** structure to perform exception handling. These sections cover the following topics:

- Understanding the **sqlca** structure
- Using the **SQLCODE** variable to obtain error codes
- Checking for the different types of exceptions with the **sqlca** structure

Fields of the sqlca structure

The **sqlca** structure is defined in the **sqlca.h** header file. The IBM Informix ESQL/C preprocessor automatically includes the **sqlca.h** header file in an Informix ESQL/C program.

The following table illustrates the fields of the **sqlca** structure.

Table 11-8. Fields of the sqlca structure

Field	Type	Value	Value description
sqlcode	int4	0	Indicates success.
		>=0, < 100	After a DESCRIBE statement, represents the type of SQL statement that is described.
		100	After a successful query that returns no rows, indicates the NOT FOUND condition. NOT FOUND can also occur in an ANSI-compliant database after an INSERT INTO/SELECT, UPDATE, DELETE, or SELECT... INTO TEMP statement fails to access any rows. For more information, see "NOT FOUND in SQLSTATE" on page 11-12.
		<0	Error code.
sqlerrm	character (72) or character (600)		When working with IBM Informix database servers this field is 72 characters long and contains the error message parameter. This parameter is used to replace a **%s** token in the actual error message. If an error message requires no parameter, this field is blank.
sqlerrp	character (8)		Internal use only.

Table 11-8. Fields of the sqlca structure (continued)

Field	Type	Value	Value description
sqlerrd	array of 6 int4s	[0]	After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor is opened, this field contains the estimated number of rows affected.
		[1]	When **SQLCODE** contains an error code, this field contains either zero or an additional error code, called the ISAM error code, that explains the cause of the main error.
			After a successful insert operation of a single row, this field contains the value of any SERIAL value generated for t hat row.
		[2]	After a successful multirow insert, update, or delete operation, this field contains the number of rows that were processed.
			After a multirow insert, update, or delete operation that ends with an error, this field contains the number of rows that were successfully processed before the error was detected.
		[3]	After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor was opened, this field contains the estimated weighted sum of disk accesses and total rows processed.
		[4]	After a syntax error in a PREPARE, EXECUTE IMMEDIATE, DECLARE, or static SQL statement, this field contains the offset in the statement text where the error was detected.
		[5]	After a successful fetch of a selected row, or a successful insert, update, or delete operation, this field contains the rowid (physical address) of the last row that was processed. Whether this rowid value corresponds to a row that the database server returns to the user depends on how the database server processes a query, particularly for SELECT statements.
sqlwarn	array of 8 characters	When Opening a Database:	
		sqlwarn0	Set to W when any other warning field is set to W. If blank, others need not be checked.
		sqlwarn1	Set to W when the database now open uses a transaction log.
		sqlwarn2	Set to W when the database now open is ANSI compliant.
		sqlwarn3	Set to W when the database server is any server other than IBM Informix SE.
		sqlwarn4	Set to W when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT data types).
		sqlwarn5	Reserved.
		sqlwarn6	Set to W when the application is connected to a database server that is running in secondary mode. The database server is a secondary server in a data-replication pair (the database server is available only for read operations).
		sqlwarn7	Set to W when client **DB_LOCALE** does not match the database locale. For more information, see the chapter on Informix ESQL/C in the *IBM Informix GLS User's Guide*.

Table 11-8. Fields of the sqlca structure (continued)

Field	Type	Value	Value description
sqlwarn	array of 8 characters	All Other Operations:	
		sqlwarn0	Set to W when any other warning field is set to W. If blank, other fields in **sqlwarn** need not be checked.
		sqlwarn1	Set to W if a column value is truncated when it is fetched into a host variable using a FETCH or a SELECT...INTO statement. On a REVOKE ALL statement, set to W when not all seven table-level privileges are revoked.
		sqlwarn2	Set to W when a FETCH or SELECT statement returns an aggregate function (SUM, AVG, MIN, MAX) value that is null.
		sqlwarn3	On a SELECT...INTO, FETCH...INTO, or EXECUTE...INTO statement, set to W when the number of items in the select list is not the same as the number of host variables given in the INTO clause to receive them. On a GRANT ALL statement, set to W when not all seven table-level privileges are granted.
		sqlwarn4	Set to W after a DESCRIBE statement if the prepared statement contains a DELETE statement or an UPDATE statement without a WHERE clause.
		sqlwarn5	Set to W following execution of a statement that does not use ANSI-standard SQL syntax (provided the **DBANSIWARN** environment variable is set).
		sqlwarn6	Set to W when a data fragment (a dbspace) has been skipped during query processing (when the DATASKIP feature is on).
		sqlwarn7	Reserved.

Using the SQLCODE Variable

The **SQLCODE** variable is an **int4** that indicates whether the SQL statement succeeded or failed. The Informix ESQL/C header file, **sqlca.h**, declares **SQLCODE** as a global variable. Since the Informix ESQL/C preprocessor automatically includes **sqlca.h** in an Informix ESQL/C program, you do not need to declare **SQLCODE**.

When the database server executes an SQL statement, the database server automatically updates the **SQLCODE** variable as follows:

1. The database server stores the exception value in the **sqlcode** field of the **sqlca** structure.

2. Informix ESQL/C copies the value of **sqlca.sqlcode** to the global **SQLCODE** variable.

Tip: For readability and brevity, use **SQLCODE** in your Informix ESQL/C program in place of **sqlca.sqlcode**.

The **SQLCODE** value can indicate the following types of exceptions:

SQLCODE = 0

Success

SQLCODE = 100

NOT FOUND condition

SQLCODE < 0

Runtime error

For information about the values of **SQLCODE** (and **sqlca.sqlcode**) and their corrective actions, use the **finderr** or **Informix Error Messages** utility or view *IBM Informix Error Messages* at the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp. For information about how to handle these exceptions, see "Checking for Exceptions with sqlca" on page 11-20.

The following sections provide additional information about **SQLCODE**.

SQLCODE in Pure C Modules

To return the same values that the **SQLCODE** status variable in Informix ESQL/C modules returns, you can use **SQLCODE** in pure C modules (modules with the **.c** extension) that you link to an Informix ESQL/C program. To use **SQLCODE** in a pure C module, declare **SQLCODE** as an external variable, as follows:

```
extern int4 SQLCODE;
```

SQLCODE and the exit() Call

To return an error code to a parent process, do *not* attempt to use the **SQLCODE** value as an argument to the **exit()** system call. When Informix ESQL/C passes back the argument of **exit()** to the parent, it passes only the lower eight bits of the value. Since **SQLCODE** is a four-byte (**long**) integer, the value that Informix ESQL/C returns to the parent process might not be what you expect.

To pass error information between processes, use the exit value as an indication that some type of error has occurred. To obtain information about the actual error, use a temporary file, a database table, or some form of interprocess communication.

SQLCODE After a DESCRIBE Statement

The DESCRIBE statement returns information about a prepared statement before the statement executes. It operates on a statement ID that a PREPARE statement has previously assigned to a dynamic SQL statement.

After a successful DESCRIBE statement, the database server sets **SQLCODE** (and **sqlca.sqlcode**) to a nonnegative integer value that represents the type of SQL statement that DESCRIBE has examined. The **sqlstype.h** header file declares constant names for each of these return values. For a list of possible **SQLCODE** values after a DESCRIBE statement, see "Determining the Statement Type" on page 15-9.

Because the DESCRIBE statement uses the **SQLCODE** field differently than any other statement, you might want to revise your exception-handling routines to accommodate this difference.

Checking for Exceptions with sqlca

After an SQL statement executes, the **sqlca** structure can indicate one of the four possible conditions that Table 11-9 shows.

Table 11-9. Exceptions That the sqlca Structure Returns

Exception Condition	sqlca Value
Success	**SQLCODE** (and **sqlca.sqlcode**) = 0
Success, but no rows found	**SQLCODE** (and **sqlca.sqlcode**) **=** 100
Success, but warnings generated	**sqlca.sqlwarn.sqlwarn0** = 'W' To indicate specific warning: • One of **sqlwarn1** to **sqlwarn7** in the **sqlca.sqlwarn** structure is also set to W
Failure, runtime error generated	**SQLCODE** (and **sqlca.sqlcode**) < 0

For a general introduction to these four conditions, see "Types of Database Exceptions" on page 11-2. The following sections discuss how **sqlca** indicates each condition.

Success in sqlca

When the database server executes an SQL statement successfully, it sets **SQLCODE** (**sqlca.sqlcode**) to zero (0). The database server might also set one or more of the following informational fields in **sqlca** after a successful SQL statement:

- After a PREPARE for a SELECT, DELETE, INSERT, or UPDATE:
  - **sqlca.sqlerrd[0]** indicates an estimated number of rows affected.
  - **sqlca.sqlerrd[3]** contains the estimated weighted sum of disk accesses and total rows processed.
- After an INSERT, **sqlca.sqlerrd[1]** contains the value that the database server has generated for a SERIAL column.
- After a SELECT, INSERT, DELETE, or UPDATE:
  - **sqlca.sqlerrd[2]** contains the number of rows that the database server processed.
  - **sqlca.sqlerrd[5]** contains the rowid (physical address) of the last row that was processed. Whether this rowid value corresponds to a row that the database server returns to the user depends on how the database server processes a query, particularly for SELECT statements.
- After a CONNECT, SET CONNECTION, DATABASE, CREATE DATABASE, or START DATABASE, the **sqlca.sqlwarn.sqlwarn0** field is set to W and other fields of **sqlca.sqlwarn** provide information about the database and connection. For more information, see "Warnings in sqlca.sqlwarn" on page 11-22.

For more information about these additional fields, see "Fields of the sqlca structure" on page 11-17. In addition, the **SQLCODE** value for some SQL statements has special meaning. For more information, see "Using the SQLCODE Variable" on page 11-19.

NOT FOUND in SQLCODE

When a SELECT or FETCH statement encounters NOT FOUND (or END OF DATA), the database server sets **SQLCODE** (**sqlca.sqlcode**) to 100. Table 11-10 lists conditions that cause SQL statements to yield NOT FOUND.

Table 11-10. SQLCODE Values That Are Set When SQL Statements Do Not Return Any Rows

SQL Statement Where SQLCODE Gets the Indicated Result	Result for ANSI-Compliant Database	Result for Non-ANSI-Compliant Database
FETCH statement: the last qualifying row has already been returned (the end of data was reached).	100	100
SELECT statement: no rows match the SELECT criteria.	100	100
DELETE and DELETE...WHERE statement (not part of multistatement PREPARE): no rows match the DELETE criteria.	100	0
INSERT INTO *tablename* SELECT statement (not part of multistatement PREPARE): no rows match the SELECT criteria.	100	0
SELECT... INTO TEMP statement (not part of multistatement PREPARE): no rows match the SELECT criteria.	100	0
UPDATE...WHERE statement (not part of multistatement PREPARE): no rows match the UPDATE criteria.	100	0

Table 11-10 on page 11-22 shows that what the NOT FOUND condition generates depends, in some cases, on whether the database is ANSI compliant.

In the following example, the INSERT statement inserts into the **hot_items** table any stock item that has an order quantity greater than 10,000. If no items have an order quantity that great, the SELECT part of the statement fails to insert any rows. The database server returns 100 in an ANSI-compliant database and 0 if the database is not ANSI compliant.

```
EXEC SQL 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;
```

For readability, use the constant SQLNOTFOUND for the END OF DATA value of 100. The **sqlca.h** header file defines the SQLNOTFOUND constant. The following comparison checks for the NOT FOUND and END OF DATA conditions:

```
if(SQLCODE == SQLNOTFOUND)
```

Warnings in sqlca.sqlwarn

When the database server executes an SQL statement successfully, but encounters a warning condition, it updates the following two fields in the **sqlca.sqlwarn** structure:

- It sets the **sqlca.sqlwarn.sqlwarn0** field to the letter W.
- It sets one other field within the **sqlwarn** structure (**sqlwarn1** to **sqlwarn7**) to the letter W to indicate the specific warning condition.

These warnings are Informix specific. Table 11-8 on page 11-17 contains two sets of warning conditions that can occur in the fields of the **sqlca.sqlwarn** structure. The

first set of warnings, shown in Table 11-8, occur after the database server opens a database or establishes a connection. For more information about these conditions, see "Determining Features of the Database Server" on page 12-18. The second set of warnings are for conditions that can occur as a result of other SQL statements.

To test for warnings, check whether the first warning field (**sqlwarn0**) is set to W. Once you determine that the database server has generated a warning, you can check the values of the other fields in **sqlca.sqlwarn** to identify the specific condition. For example, if you want to find out what kind of database a CONNECT statement has opened, you can use the code that Figure 11-6 shows.

```
int trans_db, ansi_db, us_db = 0;
  .
  .
  .

msg = "CONNECT stmt";
EXEC SQL connect to 'stores7';
if(SQLCODE < 0)      /* < 0 is an error */
   err_chk(msg);
if (sqlca.sqlwarn.sqlwarn0 == 'W')
   {
   if (sqlca.sqlwarn.sqlwarn1 == 'W' )
      trans_db = 1;
   if (sqlca.sqlwarn.sqlwarn2 == 'W' )
      ansi_db = 1;
   if (sqlca.sqlwarn.sqlwarn3 == 'W' )
      us_db = 1;
   }
```

Figure 11-6. Code Fragment That Checks for Warnings After a CONNECT Statement

Runtime Errors in SQLCODE

When an SQL statement results in a runtime error, the database server sets **SQLCODE** (and **sqlca.sqlcode**) to a negative value. The actual number identifies the particular error. The error message documentation lists the Informix-specific error codes and their corrective actions. To see the error-message documentation, use the **finderr** or **Informix Error Messages** utility or view *IBM Informix Error Messages* at the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

From within your Informix ESQL/C program, you can retrieve error message text that is associated with a negative **SQLCODE** (**sqlca.sqlcode**) value with the **rgetlmsg()** or **rgetmsg()** library function. See "Library Functions for Retrieving Error Messages" on page 11-29.

When the database server encounters a runtime error, it might also set the following other fields in the **sqlca** structure:

- **sqlca.sqlerrd[1]** to hold the additional ISAM error return code. You can also use the **rgetlmsg()** and **rgetmsg()** library functions to obtain ISAM error message text.
- **sqlca.sqlerrd[2]** to indicate the number of rows processed *before* the error occurred in a multirow INSERT, UPDATE, or DELETE statement.
- **sqlca.sqlerrm** is used differently depending on what type of database server is using it.

  If the server is an Informix database server this is set to an error message parameter. This value is used to replace a **%s** token in the error message.

For example, in the following error message, the name of the table (**sam.xyz**) is saved in **sqlca.sqlerrm**:

```
310: Table (sam.xyz) already exists in database.
```

If the server is a DB2 database server this field is set to the complete error message.

- **sqlca.sqlerrd[4]** after a PREPARE, EXECUTE IMMEDIATE, or DECLARE statement that encountered an error. For more information, see "Errors After a PREPARE Statement" on page 11-24.

Tip: You can also test for errors with the WHENEVER SQLERROR statement. For more information, see "The WHENEVER Statement" on page 11-28.

Errors After a PREPARE Statement: When the database server returns an error for a PREPARE statement, this error is usually because of a syntax error in the prepared text. When this occurs, the database server returns the following information:

- The **SQLCODE** variable indicates the cause of the error.
- The **sqlca.sqlerrd[4]** field contains the offset into the prepared statement text at which the error occurs. Your program can use the value in **sqlca.sqlerrd[4]** to indicate where the syntax of the dynamically prepared text is incorrect.

If you prepare multiple statements with a single PREPARE statement, the database server returns an error status on the *first* error in the text, even if it encounters several errors.

Important: The **sqlerrd[4]** field, which is the offset of the error into the SQL statement, might not always be correct because the Informix ESQL/C preprocessor converts the embedded SQL statements into host-language format. In so doing, the preprocessor might change the relative positions of the elements within the embedded statement.

For example, consider the following statement, which contains an invalid WHERE clause:

```
EXEC SQL INSERT INTO tab VALUES (:x, :y, :z)
   WHERE i = 2;
```

The preprocessor converts this statement to a string like the following string:

```
" insert into tab values ( ? , ? , ? ) where i = 2 "
```

This string does not have the EXEC SQL keywords. Also, the characters **?, ?, ?** have replaced **:x, :y, :z** (five characters instead of eight). The Informix ESQL/C preprocessor has also dropped a newline character between the left parenthesis (")") and the WHERE keyword. Thus, the offset of error in the SQL statement that the database server sees is different than the offset of the error in the embedded SQL statement.

The **sqlca.sqlerrd[4]** field also reports statement-offset values for errors in the EXECUTE IMMEDIATE and DECLARE statements.

SQLCODE After an EXECUTE Statement: After an EXECUTE statement, the database server sets **SQLCODE** to indicate the success of the prepared statement as follows:

- If the database server cannot execute a prepared statement successfully, it sets **SQLCODE** to a value less than 0. The **SQLCODE** variable holds the error that the database server returns from the statement that failed.

- If the database server can successfully execute the prepared statement in the block, it sets **SQLCODE** to 0; if the prepared block includes multiple statements, all of the statements succeeded.

Displaying Error Text (Windows)

Your Informix ESQL/C application can use the Informix **ERRMESS.HLP** file to display text that describes an error and its corrective action. You can call the Windows API **WinHelp()** with the following WinHelp parameters.

WinHelp Parameter	Data
HELP_CONTEXT	Error number from SQLCODE or **sqlca.sqlcode**
HELP_CONTEXTPOPUP	Error number from SQLCODE or **sqlca.sqlcode**
HELP_KEY	Pointer to string that contains error number from SQLCODE or **sqlca.sqlcode** and is converted to ASCII with **sprintf()** or **wsprintf()**
HELP_PARTIALKEY	Pointer to string that contains error number from SQLCODE or **sqlca.sqlcode** and is converted to ASCII with **sprintf()** or **wsprintf()**

Choosing an Exception-Handling Strategy

By default, an Informix ESQL/C application does not perform *any* exception handling for SQL statements. Therefore, unless you explicitly provide such code, execution continues when an exception occurs. While this behavior might not be too serious for successful execution, warnings, and NOT FOUND conditions, it can have very serious consequences in the event of a runtime error.

A runtime error might halt the program execution. Unless you check for and handle these errors in the application code, this behavior can cause the end user great confusion and annoyance. It also can leave the application in an inconsistent state.

Within an Informix ESQL/C application, choose a consistent strategy for exception handling. You can choose one of the following exception-handling strategies:

- You can check after each SQL statement, which means that you include code to test the value of **SQLSTATE** (or **SQLCODE**) after *each* SQL statement.
- You can use the WHENEVER statement to associate a response to take each time a particular type of exception occurs.

Important: Consider how to perform exception handling in an application before you begin development so that you take a consistent and maintainable approach.

Checking After Each SQL Statement

To check for an exception, you can include code to explicitly test the value of **SQLSTATE** (or **SQLCODE**).

Tip: Decide whether to use **SQLSTATE** (and the diagnostics area) or **SQLCODE** (and the **sqlca** structure) to determine exception values. Use the chosen exception-handling variables consistently. If you mix these two variables unnecessarily, you create code that is difficult to maintain. Keep in mind that **SQLSTATE** is the more flexible and portable of these two options.

For example, if you want to use **SQLSTATE** to check whether a CREATE
DATABASE statement has executed as expected, you can use the code that
Figure 11-7 shows.

```
EXEC SQL create database personnel with log;
if(strncmp(SQLSTATE, "02", 2) > 0) /* > 02 is an error */
{
EXEC SQL get diagnostics exception 1
:message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
message[messlen] ='\0'; /* terminate the string. */

printf("SQLSTATE: %s, %s\n", SQLSTATE, message);
exit(1);
}
```

Figure 11-7. Using SQLSTATE to Test Whether an Error Occurred During an SQL Statement

As an alternative, you can write an exception-handling function that processes any
exception. Your program can then call this single exception-handling function after
each SQL statement.

The **sqlstate_exception()** function, which Figure 11-8 shows, is an example of an
exception-handling function that uses the **SQLSTATE** variable and the diagnostics
area to check for warnings, the NOT FOUND condition, and runtime errors. It is
designed to be called after each SQL statement.

```
EXEC SQL select * from customer where fname not like "%y";
sqlstate_exception("select");
  .
  .
  .
int4 sqlstate_exception(s)
char *s;
{
   int err = 0;

   if(!strncmp(SQLSTATE, "00", 2) ||
         !strncmp(SQLSTATE,"02",2))
      return(SQLSTATE[1]);

   if(!strncmp(SQLSTATE, "01", 2))
      printf("\n********Warning encountered in %s********\n",
         statement);
   else /* SQLSTATE class > "02" */
      {
      printf("\n********Error encountered in %s********\n",
         statement);
      err = 1;
      }

   disp_sqlstate_err(); /* See the getdiag sample program */
   if(err)
      {
      printf("********Program terminated*******\n\n");
      exit(1);
      }

 /*
  * Return the SQLCODE
  */
 return(SQLCODE);
}
```

Figure 11-8. Example of an Exception-Handling Function That Uses SQLSTATE

The **sqlstate_exception()** function, which Figure 11-8 on page 11-27 shows, handles exceptions as follows:

- If the statement was successful, **sqlstate_exception()** returns zero.
- If a NOT FOUND condition occurs after a SELECT or a FETCH statement, **sqlstate_exception()** returns a value of 2.
- If a warning or a runtime error occurs—that is, if the first two bytes of **SQLSTATE** are "01" (warning) or are greater than "02" (error)—the **sqlstate_exception()** function calls the **disp_sqlstate_err()** function to display exception information. (For the code of the **disp_sqlstate_err()** function, see 11-34.)
- If **SQLSTATE** indicates an error, the **sqlstate_exception()** function uses the **exit()** system call to exit the program. Without this call to **exit()**, execution would continue at the next SQL statement after the one that had generated the error.

To handle errors, the **sqlstate_exception()** function can alternatively omit the **exit()** call and allow execution to continue. In this case, the function must return the **SQLSTATE** or **SQLCODE** (for Informix-specific errors) value so the calling program can determine what action to take for a runtime error.

The WHENEVER Statement

You can use the WHENEVER statement to trap for exceptions that occur during the execution of SQL statements. The WHENEVER statement provides the following information:

- What *condition* to check for:
  - SQLERROR checks whether an SQL statement has failed. The application performs the specified *action* when the database server sets **SQLCODE** (**sqlca.sqlcode**) to a negative value and the class code of **SQLSTATE** to a value greater than "02".
  - NOT FOUND checks whether specified data has not been found. The application performs the specified *action* when the database server sets **SQLCODE** (**sqlca.sqlcode**) to SQLNOTFOUND and the class code of **SQLSTATE** to "02".
  - SQLWARNING checks whether the SQL statement has generated a warning. The application performs the specified *action* when the database server sets **sqlca.sqlwarn.sqlwarn0** (and some other field of **sqlca.sqlwarn**) to W and sets the class code of **SQLSTATE** to "01".

  In a Windows environment, do not use the WHENEVER ERROR STOP construction in an Informix ESQL/C program that you want to compile as a DLL.

- What *action* to take when the specified *condition* occurs:
  - CONTINUE ignores the exception and continues execution at the next statement after the SQL statement.
  - GO TO *label* transfers execution to the section of code that the specified *label* introduces.
  - STOP stops program execution immediately.
  - CALL *function name* transfers execution to the specified *function name*.

If no WHENEVER statement exists for a given condition, the Informix ESQL/C preprocessor uses CONTINUE as the default action. To execute the **sqlstate_exception()** function (shown in Figure 11-8 on page 11-27) every time an error occurs, you can use the GOTO action of the WHENEVER SQLERROR statement. If you specify the SQLERROR condition of WHENEVER, you obtain the same behavior as if you check the **SQLCODE** or **SQLSTATE** variable for an error after each SQL statement.

The WHENEVER statement for the GOTO action can take the following two forms:

- The ANSI-standard form uses the keywords GOTO (one word) and introduces the label name with a colon (:):

  ```
  EXEC SQL whenever goto :error_label;
  ```

- The Informix extension uses the keywords GO TO (two words) and specifies just the label name:

  ```
  EXEC SQL whenever go to error_label;
  ```

With the GOTO action, your program automatically transfers control to the **error_label** label when the SQL statement generates an exception. When you use the GOTO *label* action of the WHENEVER statement, your code must contain the label and appropriate logic to handle the error condition. In the following example, the logic at *label* is simply a call to the **sqlstate_exception()** function:

```
error_label:
   sqlstate_exception (msg);
```

You must define this **error_label** label in *each* program block that contains SQL statements. If your program contains more than one function, you might need to include the **error_label** label and code in *each* function. Otherwise, the preprocessor generates an error when it reaches the function that does not contain the **error_label**. It tries to insert the code that the WHENEVER...GOTO statement has requested, but the function has not defined the **error_label** label.

To remove the preprocessor error, you can put the labeled statement with the same label name in each function, you can issue another action for the WHENEVER statement to reset the error condition, or you can replace the GOTO action with the CALL action to call a separate function.

You can also use the CALL keyword in the WHENEVER statement to call the **sqlstate_exception()** function when errors occur. (The CALL option is an Informix extension to the ANSI standard.)

If you want to call the **sqlstate_exception()** function every time an SQL error occurs in the program, take the following steps:

- Modify the **sqlstate_exception()** function so that it does not need any arguments. Functions that the CALL action specifies cannot take arguments. To pass information, use global variables instead.
- Put the following WHENEVER statement in the early part of your program, before any SQL statements:

```
EXEC SQL whenever sqlerror call sqlstate_exception;
```

Tip: In the preceding code fragment, you do not include the parentheses after the **sqlstate_exception()** function.

Make sure, however, that all functions that the WHENEVER...CALL affects can find a declaration of the **sqlstate_exception()** function. For details of the syntax and how to use the WHENEVER statement, see the *IBM Informix Guide to SQL: Syntax*.

Library Functions for Retrieving Error Messages

Each **SQLCODE** value has an associated message. Error message files in the **$INFORMIXDIR/msg** directory store the message number and its text. For more information about error messages, use the **finderr** or **Informix Error Messages** utility or view *IBM Informix Error Messages* at the IBM Informix Information Center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

When you use **SQLCODE** and the **sqlca** structure, you can retrieve error message text with the **rgetlmsg()** and **rgetmsg()** functions. Both of these functions take the **SQLCODE** error code as input and return the associated error message.

Tip: When you use **SQLSTATE** and the GET DIAGNOSTICS statement, you can access information in the **MESSAGE_TEXT** field of the diagnostics area to retrieve the message text that is associated with an exception. For more information, see "Exception Information" on page 11-5.

The following pages describe the **rgetlmsg()** and **rgetmsg()** functions.

Important: Use **rgetlmsg()** in any new Informix ESQL/C code that you write. Informix ESQL/C provides the **rgetmsg()** function primarily for compatibility with earlier versions.

Displaying Error Text in a Windows Environment

Your Informix ESQL/C application can use the Informix **ERRMESS.HLP** file to display text that describes an error and its corrective action. You can call the Windows API **WinHelp()** with the following WinHelp parameters.

WinHelp Parameter	Data
HELP_CONTEXT	Error number from SQLCODE or **sqlca.sqlcode**
HELP_CONTEXTPOPUP	Error number from SQLCODE or **sqlca.sqlcode**
HELP_KEY	Pointer to string that contains error number from SQLCODE or **sqlca.sqlcode** and is converted to ASCII with **sprintf()** or **wsprintf()**
HELP_PARTIALKEY	Pointer to string that contains error number from SQLCODE or **sqlca.sqlcode** and is converted to ASCII with **sprintf()** or **wsprintf()**

A Program That Uses Exception Handling

The **getdiag.ec** program contains exception handling on each of the SQL statements that the program executes. This program is a modified version of the **demo1.ec** program, which Chapter 1 of this publication explains. The version that this section lists and describes uses the following exception-handling methods:

- The **SQLSTATE** variable and the GET DIAGNOSTICS statement to obtain exception information.
- The SQLWARNING and SQLERROR keywords of the WHENEVER statement to call the **whenexp_chk()** function for warnings and errors.

The **whenexp_chk()** function displays the error number and the accompanying ISAM error, if one exists. The **exp_chk.ec** source file contains this function and its exception-handling functions. The **getdiag.ec** source file (11-30) includes the **exp_chk.ec** file (11-32).

Compiling the Program

Use the following command to compile the **getdiag** program:

```
esql -o getdiag getdiag.ec
```

The **-o getdiag** option tells **esql** to name the executable program **getdiag**. Without the **-o** option, the name of the executable program defaults to **a.out**. For more information about the **esql** preprocessor command, see "Using the esql Command" on page 2-4.

Guide to the getdiag.ec File

The annotations in this section primarily describe the exception-handling statements.

```
=======================================================================
  1. #include <stdio.h>
  2. EXEC SQL define FNAME_LEN 15;
  3. EXEC SQL define LNAME_LEN 15;
  4. int4 sqlstate_err();
  5. extern char statement[20];
  6. main()
  7. {
  8.     EXEC SQL BEGIN DECLARE SECTION;
```

```
 9.          char fname[ FNAME_LEN + 1 ];
10.          char lname[ LNAME_LEN + 1 ];
11.          EXEC SQL END DECLARE SECTION;
12.          EXEC SQL whenever sqlerror CALL whenexp_chk;
13.          EXEC SQL whenever sqlwarning CALL whenexp_chk;
14.          printf("GETDIAG Sample ESQL program running.\n\n");
15.          strcpy (statement, "CONNECT stmt");
16.          EXEC SQL connect to 'stores7';
17.          strcpy (statement, "DECLARE stmt");
18.          EXEC SQL declare democursor cursor for
19.             select fname, lname
20.             into :fname, :lname;
21.             from customer
22.             where lname < 'C';
23.          strcpy (statement, "OPEN stmt");
24.          EXEC SQL open democursor;
25.          strcpy (statement, "FETCH stmt");
26.          for (;;)
27.          {
28.             EXEC SQL fetch democursor;
29.             if(sqlstate_err() == 100)
30.                break;
31.             printf("%s %s\n", fname, lname);
32.          }
33.          strcpy (statement, "CLOSE stmt");
34.          EXEC SQL close democursor;
========================================================================
```

Line 4

Line 4 declares an external global variable to hold the name of the most-recently executed SQL statement. The exception-handling functions use this information (see "Lines 169 to 213" on page 11-40).

Lines 12 and 13

The WHENEVER SQLERROR statement tells the Informix ESQL/C preprocessor to add code to the program to call the **whenexp_chk()** function whenever an SQL statement generates an error. The WHENEVER SQLWARNING statement tells the Informix ESQL/C preprocessor to add code to the program to call the **whenexp_chk()** function whenever an SQL statement generates a warning. The **whenexp_chk()** function is in the **exp_chk.ec** file, which line 40 includes.

Line 15

The **strcpy()** function copies the string `"CONNECT stmt"` to the global **statement** variable. If an error occurs, the **whenexp_chk()** function uses this variable to print the name of the statement that caused the failure.

Lines 17, 23, 25, and 33

These lines copy the name of the current SQL statement into the **statement** variable before the DECLARE, OPEN, FETCH, and CLOSE statements execute. This action enables the **whenexp_chk()** function to identify the statement that failed if an error occurs.

```
========================================================================
35.          strcpy (statement, "FREE stmt");
36.          EXEC SQL free democursor;
37.          strcpy (statement, "DISCONNECT stmt");
38.          EXEC SQL disconnect current;
```

```
39.        printf("\nGETDIAG Sample Program Over.\n");
40. }        /* End of main routine */
41. EXEC SQL include exp_chk.ec;
```

===

Lines 35 and 37

These lines copy the name of the current SQL statement into the **statement** variable
before the FREE and DISCONNECT statements execute. The **whenexp_chk()**
function uses the **statement** variable to identify the statement that failed if an error
occurs.

Line 41

The **whenexp_chk()** function examines the **SQLSTATE** status variable to determine
the outcome of an SQL statement. Because several demonstration programs use the
whenexp_chk() function with the WHENEVER statement for exception handling,
the **whenexp_chk()** function and its supporting functions are placed in a separate
source file, **exp_chk.ec**. The **getdiag** program must include this file with the
Informix ESQL/C **include** directive because the exception-handling functions use
Informix ESQL/C statements.

Tip: You should consider putting functions such as **whenexp_chk**() into a library
and include this library on the command line when you compile the Informix
ESQL/C program.

Note: The following section describes the **exp_chk.ec** source file.

Guide to the exp_chk.ec File

The **exp_chk.ec** file contains exception-handling functions for the Informix
ESQL/C demonstration programs. These functions support the following two
types of exception handling:

- A function that a WHENEVER SQLERROR CALL statement specifies performs
  exception handling.

  Functions to support this type of exception handling include **whenexp_chk()**,
  sqlstate_err(), and **disp_sqlstate_err()**. The **getdiag** sample program in this
  chapter uses this form of exception handling.

- A function that an Informix ESQL/C program calls explicitly after each SQL
  statement performs exception handling.

  Functions to support this type of exception handling include **exp_chk()**,
  exp_chk2(), **sqlstate_err()**, **disp_sqlstate_err()**, and **disp_exception()**. The
  dispcat_pic sample program (Chapter 7) uses **exp_chk2()** while the **dyn_sql**
  sample program (Chapter 16) uses **exp_chk()** to perform exception handling.

To obtain exception information, the preceding functions use the **SQLSTATE**
variable and the GET DIAGNOSTICS statement. They use **SQLCODE** only when
they need Informix-specific information.

===

```
1. EXEC SQL define SUCCESS 0;
2. EXEC SQL define WARNING 1;
3. EXEC SQL define NODATA  100;
4. EXEC SQL define RTERROR -1;
5. char statement[80];
6. /*
7.  * The sqlstate_err() function checks the SQLSTATE status variable
     * to see
```

```
 8.   * if an error or warning has occurred following an SQL statement.
 9.   */
10.  int4 sqlstate_err()
11.  {
12.     int4 err_code = RTERROR;
13.     if(SQLSTATE[0] == '0') /* trap '00', '01', '02' */
14.        {
15.        switch(SQLSTATE[1])
16.           {
17.           case '0': /* success - return 0 */
18.              err_code = SUCCESS;
19.              break;
20.           case '1': /* warning - return 1 */
21.              err_code = WARNING;
22.              break;
23.           case '2': /* end of data - return 100 */
24.              err_code = NODATA;
25.              break;
26.           default: /* error - return -1*/
27.              break;
28.           }
29.        }
30.     return(err_code);
31.  }
```

==

Lines 1 to 4

These Informix ESQL/C **define** directives create definitions for the success,
warning, NOT FOUND, and runtime error exceptions. Several functions in this file
use these definitions instead of constants to determine actions to take for a given
type of exception.

Line 5

The **statement** variable is a global variable that the calling program (which
declares it as **extern**) sets to the name of the most-recent SQL statement.

The **whenexp_chk()** function displays the SQL statement name as part of the error
information (see lines 85 and 92).

Lines 6 to 31

The **sqlstate_err()** function returns a status of 0, 1, 100, or **-1** to indicate if the
current exception in **SQLSTATE** is a success, warning, NOT FOUND, or runtime
error. The **sqlstate_err()** function checks the first two characters of the global
SQLSTATE variable. Because Informix ESQL/C automatically declares the
SQLSTATE variable, the function does not need to declare it.

Line 13 checks the *first* character of the global **SQLSTATE** variable. This character
determines whether the most-recently executed SQL statement has generated a
nonerror condition. Nonerror conditions include the NOT FOUND condition (or
END OF DATA), success, and warnings. Line 15 checks the *second* character of the
global **SQLSTATE** variable (**SQLSTATE[1]**) to determine the type of nonerror
condition generated.

The **sqlstate_err()** function sets **err_code** to indicate the exception status as follows:

- Lines 17 to 19: If **SQLSTATE** has a class code of "00", the most-recently executed SQL statement was successful. The **sqlstate_err()** function returns 0 (which line 1 defines as SUCCESS).
- Lines 20 to 22: If **SQLSTATE** has a class code of "01", the most-recently executed SQL statement generated a warning. The **sqlstate_err()** function returns 1 (which line 2 defines as WARNING).
- Lines 23 to 25: If **SQLSTATE** has a class code of "02", the most-recently executed SQL statement generated the NOT FOUND (or END OF DATA) condition. The **sqlstate_err()** function returns 100 (which line 3 defines as NODATA).

If **SQLSTATE[1]** contains any character other than '0', '1', or '2', then the most-recently executed SQL statement generated a runtime error. **SQLSTATE** also indicates a runtime error if **SQLSTATE[0]** contains some character other than '0'. In either case, line 30 returns a negative one (-1) (which line 4 defines as RTERROR).

```
========================================================================
32.  /*
33.   * The disp_sqlstate_err() function executes the GET DIAGNOSTICS
34.   * statement and prints the detail for each exception that is
35.   * returned.
36.   */
37.  void disp_sqlstate_err()
38.  {
39.  mint j;
40.  EXEC SQL BEGIN DECLARE SECTION;
41.      mint exception_count;
42.      char overflow[2];
43.      int exception_num=1;
44.      char class_id[255];
45.      char subclass_id[255];
46.      char message[8191];
47.      mint messlen;
48.      char sqlstate_code[6];
49.      mint i;
50.  EXEC SQL END DECLARE SECTION;
51.      printf("--------------------------------");
52.      printf("------------------------\n");
53.      printf("SQLSTATE: %s\n",SQLSTATE);
54.      printf("SQLCODE: %d\n", SQLCODE);
55.      printf("\n");
56.     EXEC SQL get diagnostics :exception_count = NUMBER,
57.          :overflow = MORE;
58.      printf("EXCEPTIONS:  Number=%d\t", exception_count);
59.      printf("More? %s\n", overflow);
60.      for (i = 1; i <= exception_count; i++)
61.        {
62.        EXEC SQL get diagnostics  exception :i
63.           :sqlstate_code = RETURNED_SQLSTATE,
64.           :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
65.           :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
66.        printf("- - - - - - - - - - - - - - - - - - -\n");
67.        printf("EXCEPTION %d: SQLSTATE=%s\n", i,
68.           sqlstate_code);
69.        message[messlen-1] = '\0';
70.        printf("MESSAGE TEXT: %s\n", message);
```

```
71.        j = byleng(class_id, stleng(class_id));
72.        class_id[j] = '\0';
73.        printf("CLASS ORIGIN: %s\n",class_id);
74.        j = byleng(subclass_id, stleng(subclass_id));
75.        subclass_id[j] = '\0';
76.        printf("SUBCLASS ORIGIN: %s\n",subclass_id);
77.        }
78.     printf("--------------------------------");
79.     printf("------------------------\n");
80.  }
```

==

Lines 32 to 80

The **disp_sqlstate_err()** function uses the GET DIAGNOSTICS statement to obtain diagnostic information about the most-recently executed SQL statement.

Lines 40 to 50 declare the host variables that receive the diagnostic information. The GET DIAGNOSTICS statement copies information from the diagnostics area into these host variables. Line 48 includes a declaration for the **SQLSTATE** value (called **sqlstate_code**) because the **disp_sqlstate_err()** function handles multiple exceptions. The **sqlstate_code** variable holds the **SQLSTATE** value for *each* exception.

Lines 53 to 55 display the values of the **SQLSTATE** and **SQLCODE** variables. If **SQLSTATE** contains "IX000" (an Informix-specific error), **SQLCODE** contains the Informix-specific error code.

The first GET DIAGNOSTICS statement (lines 56 and 57) stores the statement information in the **:exception_count** and **:overflow** host variables. Lines 58 and 59 then display this information.

The **for** loop (lines 60 to 77) executes for *each* exception that the most-recently executed SQL statement has generated. The **:exception_count** host variable, which holds the number of exceptions, determines the number of iterations that this loop performs.

The second GET DIAGNOSTICS statement (lines 62 to 65) obtains the exception information for a *single* exception. Lines 67 to 70 print out the **SQLSTATE** value (**sqlstate_code**) and its corresponding message text. In addition to SQL error messages, **disp_sqlstate_err()** can display ISAM error messages because the MESSAGE_TEXT field of the diagnostics area also contains these messages. The function uses the MESSAGE_LENGTH value to determine where to place a null terminator in the message string. This action causes only the portion of the message variable that contains text to be output (rather than the full 255-character buffer).

Declare both the class- and the subclass-origin host variables as character buffers of size 255. However, often the text for these variables fills only a small portion of the buffer. Rather than display the full buffer, lines 71 to 73 use the Informix ESQL/C **byleng()** and **stleng()** library functions to display only that portion of **:class_id** that contains text; lines 74 to 76 perform this same task for **:subclass_id**.

==

```
81.  void disp_error(stmt)
82.  char *stmt;
83.  {
84.     printf("\n********Error encountered in %s********\n",
```

```
85.         stmt);
86.     disp_sqlstate_err();
87. }
88. void disp_warning(stmt)
89. char *stmt;
90. {
91.     printf("\n********Warning encountered in %s********\n",
92.         stmt);
93.     disp_sqlstate_err();
94. }
95. void disp_exception(stmt, sqlerr_code, warn_flg)
96. char *stmt;
97. int4 sqlerr_code;
98. mint warn_flg;
99. {
100.     switch(sqlerr_code)
101.         {
102.         case SUCCESS:
103.         case NODATA:
104.             break;
105.         case WARNING:
106.             if(warn_flg)
107.                 disp_warning(stmt);
108.             break;
109.         case RTERROR:
110.             disp_error(stmt);
111.             break;
112.         default:
113.             printf("\n********INVALID EXCEPTION STATE for
    %s********\n",
114.                 stmt);
115. /*          break;
116.         }
117. }
```

==

Lines 81 to 87

The **disp_error()** function notifies the user of a runtime error. It calls the
disp_sqlstate_err() function (line 86) to display the diagnostic information.

Lines 88 to 94

The **disp_warning()** function notifies the user of a warning. It calls the
disp_sqlstate_err() function (line 93) to display the diagnostic information.

Lines 95 to 117

The **disp_exception()** function handles the display of the exception information. It
expects the following three arguments:

stmt   is the name of the most-recently executed SQL statement.

sqlerr_code   is the code that **sqlstate_err()** returns to indicate the type of
                exception encountered.

warn_flg   is a flag to indicate whether to display the diagnostic information
                for a warning.

Lines 102 to 104 handle the SUCCESS and NOData conditions. For either of these
cases, the function displays no diagnostic information. Lines 105 to 108 notify the

user that a warning has occurred. The function checks the **warn_flg** argument to determine whether to call the **disp_warning()** function to display warning information for the most-recently executed SQL statement (lines 137 to 142). Lines 109 to 111 notify the user that a runtime error has occurred. The **disp_err()** function actually handles display of the diagnostic information.

```
========================================================================
118.    * The exp_chk() function calls sqlstate_err() to check the SQLSTATE
119.    * status variable to see if an error or warning has occurred
        * following
120.    * an SQL statement. If either condition has occurred, exp_chk()
121.    * calls disp_sqlstate_err() to print the detailed error
        * information.
122.    *
123.    * This function handles exceptions as follows:
124.    *    runtime errors - call exit()
125.    *    warnings - continue execution, returning "1"
126.    *    success - continue execution, returning "0"
127.    *    Not Found - continue execution, returning "100"
128.    */
129. long exp_chk(stmt, warn_flg)
130. char *stmt;
131. int warn_flg;
132.  {
133.    int4 sqlerr_code = SUCCESS;
134.    sqlerr_code = sqlstate_err();
135.    disp_exception(stmt, sqlerr_code, warn_flg);
136.    if(sqlerr_code == RTERROR)    /* Exception is a runtime error */
137.    {
138.       /* Exit the program after examining the error */
139.       printf("********Program terminated********\n\n");
140.       exit(1);
141.    }
142. /*   else                    /* Exception is "success", "Not Found", */
143.       return(sqlerr_code);   /* or "warning" */
144. }

========================================================================
```

Lines 118 to 144

The **exp_chk()** function is one of three *wrapper* functions that handle exceptions. It analyzes the **SQLSTATE** value to determine the success or failure of the most-recent SQL statement. This function is designed to be called explicitly after each SQL statement. This design requires the following features:

- The **exp_chk()** function passes as an argument the name of the SQL statement that generated the exception.

  Because the WHENEVER statement does not invoke the function, the function is not restricted to using a global variable.

- The **exp_chk()** function returns a value in the event of a successful execution of the SQL statement (0), the NOT FOUND condition (100), or a warning (1).

  Because the calling program explicitly calls **exp_chk()**, the calling program can handle the return value.

- The **exp_chk()** function uses a flag argument (**warn_flg**) to indicate whether to display warning information to the user.

  Because warnings can indicate non-serious errors and, after a CONNECT, can be informational, displaying warning information can be both distracting and

unnecessary to the user. The **warn_flg** argument allows the calling program to determine whether to display warning information that SQL statements might generate.

The **sqlstate_err()** function (line 134) determines the type of exception that **SQLSTATE** contains. The function then calls **disp_exception()** (line 135) and passes the **warn_flg** argument to indicate whether to display warning information. To handle a runtime error, the **sqlstate_err()** function calls the **exit()** system function (lines 136 to 141) to terminate the program. This behavior is the same as what the **whenexp_chk()** function (see lines 170 to 214) provides for runtime errors.

The **dyn_sql** sample program (see "The dyn_sql Program" on page 16-28) also uses **exp_chk()** to handle exceptions.

```
========================================================================
145.   * The exp_chk2() function calls sqlstate_err() to check the
       * SQLSTATE
146.   * status variable to see if an error or warning has occurred
       * following
147.   * an SQL statement. If either condition has occurred, exp_chk2()
148.   * calls disp_sqlstate_err() to print the detailed error
       * information.
149.   *
150.   * This function handles exceptions as follows:
151.   *    runtime errors - continue execution, returning SQLCODE (<0)
152.   *    warnings - continue execution, returning one (1)
153.   *    success - continue execution, returning zero (0)
154.   *    Not Found - continue execution, returning 100
155.   */
156. int4 exp_chk2(stmt, warn_flg)
157. char *stmt;
158. mint warn_flg;
159.  {
160.    int4 sqlerr_code = SUCCESS;
161.    int4 sqlcode;
162.    sqlcode = SQLCODE;    /* save SQLCODE in case of error */
163.    sqlerr_code = sqlstate_err();
164.    disp_exception(stmt, sqlerr_code, warn_flg);
165.    if(sqlerr_code == RTERROR)
166. /*    sqlerr_code = sqlcode;
167.    return(sqlerr_code);
168.  }
========================================================================
```

Lines 145 to 168

The **exp_chk2()** function is the second of the three exception-handling *wrapper* functions in the **exp_chk.ec** file. It performs the same basic task as the **exp_chk()** function. Both functions are designed to be called after each SQL statement and both return a status code. The only difference between the two is in the way they respond to runtime errors. The **exp_chk()** function calls **exit()** to terminate the program (line 140), while the **exp_chk2()** function returns the **SQLCODE** value to the calling program (lines 165 to 166).

The **exp_chk2()** function returns **SQLCODE** rather than **SQLSTATE** to allow the program to check for particular Informix-specific error codes. A possible enhancement might be to return both the **SQLSTATE** and **SQLCODE** values.

The **dyn_sql** sample program, described in 16-28, also uses **exp_chk2()** to handle exceptions.

```
========================================================================
169.    *
170. * The whenexp_chk() function calls sqlstate_err() to check the
     * SQLSTATE
171. * status variable to see if an error or warning has occurred
     * following
172. * an SQL statement. If either condition has occurred, whenerr_chk()
173. * calls disp_sqlstate_err() to print the detailed error
     * information.
174. *
175. * This function is expected to be used with the WHENEVER SQLERROR
176. * statement: it executes an exit(1) when it encounters a negative
177. * error code. It also assumes the presence of the "statement"
     * global
178. * variable, set by the calling program to the name of the statement
179. * encountering the error.
180. */
181. whenexp_chk()
182. {
183.    int4 sqlerr_code = SUCCESS;
184.    mint disp = 0;
185.    sqlerr_code = sqlstate_err();
186.    if(sqlerr_code == WARNING)
187.       {
188.         disp = 1;
189.         printf("\n********Warning encountered in %s********\n",
190.             statement);
191.       }
192.    else
193.         if(sqlerr_code == RTERROR)
194.         {
195.             printf("\n********Error encountered in %s********\n",
196.                 statement);
197.             disp = 1;
198.         }
199.    if(disp)
200.         disp_sqlstate_err();
201.    if(sqlerr_code == RTERROR)
202.       {
203.         /* Exit the program after examining the error */
204.         printf("********Program terminated*******\n\n");
205.         exit(1);
206.       }
207.    else
208.       {
209.       if(sqlerr_code == WARNING)
210.         printf("\n********Program execution
     continues********\n\n");
211.       return(sqlerr_code);
212.       }
213. }

========================================================================
```

Lines 169 to 213

The **whenexp_chk()** function is the third exception-handling *wrapper* function in the **exp_chk.ec** file. It too analyzes the **SQLSTATE** values and uses the GET DIAGNOSTICS statement for exception handling. However, this function is designed to be called with the following WHENEVER statements:

```
EXEC SQL whenever sqlerror call whenexp_chk;
EXEC SQL whenever sqlwarning call whenexp_chk;
```

The WHENEVER statement imposes the following restrictions on the design of the **whenexp_chk()** function:

- The **whenexp_chk()** function cannot receive arguments; therefore, the function uses a global variable, **statement**, to identify the SQL statement that generated the exception (lines 190 and 196).

  To use arguments with the **whenexp_chk()** function, you can use the GOTO clause of the WHENEVER statement.

  ```
  EXEC SQL whenever sqlerror goto :excpt_hndlng;
  ```

  where the label **:excpt_hndlng** would have the following code:

  ```
  :excpt_hndlng
     whenexp_chk(statement);
  ```

- The **whenexp_chk()** function cannot return any value; therefore, it cannot return the particular exception code to the main program.

  For this reason, **whenexp_chk()** handles runtime errors instead of the main program; **whenexp_chk()** calls the **exit()** function when it encounters a runtime error. To have the main program access the error code, you can modify **whenexp_chk()** to set a global variable.

The **getdiag** sample program, which this chapter describes, uses **whenexp_chk()** to handle exceptions. See lines 11 and 12 of the **getdiag.ec** file in "Guide to the getdiag.ec File" on page 11-30.

The **sqlstate_err()** function (line 185) returns an integer that indicates the success of the most-recently executed SQL statement. This return value is based on the **SQLSTATE** value.

Lines 186 to 198 display a special line to bring attention to the exception information that was generated. The **disp** variable is a flag that indicates whether to display exception information. The function displays exception information for warnings (WARNING) and runtime errors (RTERROR) but not for other exception conditions. The calls to the **printf()** function (lines 189 and 195) display the name of the SQL statement that generated the warning or error. A global variable (called **statement**) must store this statement name because the function cannot receive it as an argument.

The **disp_sqlstate_err()** function (lines 199 and 200) displays the information that the diagnostics area contains *only* if **SQLSTATE** indicates a warning or a runtime error (**disp** = 1).

Lines 201 to 206 handle a runtime error. They notify the user of the program termination and then use the **exit()** system call (line 205) to terminate the program. The call to the **disp_sqlstate_err()** function (line 200) has already displayed information about the cause of the runtime error.

Chapter 12. Working with the database server

These topics explain how an IBM Informix ESQL/C program can interact with a database server. It contains the following information:

- A description of the client-server architecture of an Informix ESQL/C application
- An overview of the ways an Informix ESQL/C program can interact with the database server
- The syntax of the Informix ESQL/C library functions that control the database server

The end of these topics present an annotated example program that is called **timeout**. The **timeout** sample program demonstrates how to interrupt an SQL request.

The client-server architecture of ESQL/C applications

When an Informix ESQL/C program executes an SQL statement, it effectively passes the statement to a database server. The database server receives SQL statements from the database application, parses them, optimizes the approach to data retrieval, retrieves the data from the database, and returns the data and status information to the application.

The Informix ESQL/C program and the database server communicate with each other through an interprocess-communication mechanism. The Informix ESQL/C program is the *client* process in the dialogue because it requests information from the database server. The database server is the *server* process because it provides

information in response to requests from the client. The division of labor between the client and server processes is particularly advantageous in networks where data might not reside on the same computer as the client program that needs it.

When you compile an Informix ESQL/C program, it is automatically equipped to communicate with database servers that reside either on the same computer (*local*) or over a network on other computers (*remote*). Figure 12-1 shows a connection between an Informix ESQL/C application and local database servers.
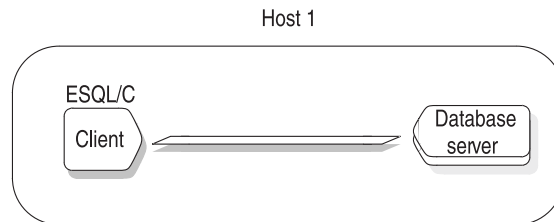
Figure 12-1. ESQL/C Application That Connects to a Local Database Server

Figure 12-2 illustrates an Informix ESQL/C application that connects across a network to a remote database server.
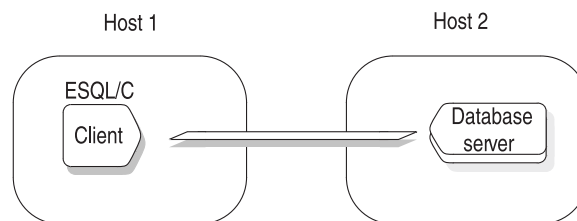
Figure 12-2. ESQL/C Application That Connects to a Remote Database Server

To establish a connection to a database server, your application must take the following actions:
- Identify database server connections that have been defined for the client-server environment of the application
- Execute an SQL statement to connect to a database server

The Client-Server Connection

An Informix ESQL/C application can establish a connection to any valid database environment. A *database environment* can be a database, a database server, or a database and a database server. Every database must have a database server to manage its information. To establish connections, the client application must be able to locate information about the available database servers. This information is in the **sqlhosts** file or registry. At runtime, the application must also be able to access information about environment variables relevant for connection. The following environment variables are accessed:
- INFORMIXCONTIME

  Specifies that the client should try connection attempts for the minimum of seconds specified
- INFORMIXCONRETRY

  Defines the number of connection attempts made by the client during the time specified by INFORMIXCONTIME

- INFORMIXSQLHOSTS

  Defines where to find the sqlhosts information. The sqlhosts information contains a list of valid database servers that the client can connect to, the type of connection to be used, and the server machine name on which each database server resides. On a UNIX operating system, this is a path to a file. In a Windows environment, this is the name of the machine on the network that contains the central registry which is accessible to the client application.

- INFORMIXSERVER

  Specifies the name of the default database server that the client should connect to. This value identifies which entry in the **sqlhosts** file or registry to use to establish the database connection.

Important: The client application connects to the default database server when the application does not explicitly specify a database server for the connection. You must set the *INFORMIXSERVER* environment variable even if the application does not establish a connection to the default database server. For more information, see "The Default Database Server" on page 12-16.

The client also sends environment variables so that the database server can determine the server-processing locale. For more information about how the database server establishes the server-processing locale, see the *IBM Informix GLS User's Guide*.

The database server uses appropriate environment information when it processes the application requests. It ignores any information that is not relevant. For example, if the application sends environment variables for a database with Asian Language Support (ALS), but it connects to a non-ALS database, the database server ignores the ALS information.

For information about how to set environment variables, see the *IBM Informix Guide to SQL: Reference* for your operating system.

Sources of Connection information about a UNIX Operating System

- The **sqlhosts** file, which contains definitions for all valid database servers in the network environment
- The **INFORMIXSERVER** environment variable, which specifies the default database server for the application

Many other environment variables can customize the database environment. For more information, see the *IBM Informix Guide to SQL: Reference* and to your *IBM Informix Administrator's Guide*.

Accessing the sqlhosts File

To establish a connection to a database server, the application process must be able to locate an entry for the database server in the **sqlhosts** file. The **sqlhosts** file defines database server connections that are valid for the client-server environment. For each database server, this file defines the following information:

- The name of the database server
- The type of connection to make between the client application and the database server
- The name of the host computer where the database server resides

- The name of a system file or program to use to establish a connection

The application expects to find the **sqlhosts** file in the **$INFORMIXDIR/etc** directory; however, you can change this location or the name of the file with the **INFORMIXSQLHOSTS** environment variable. If the database server does not reside on the computer where the client program runs, an **sqlhosts** file must reside on the host computers of both the Informix ESQL/C client program and the database server.

The client application can connect to any database server that the **sqlhosts** file defines. If your application needs to connect to a database server that **sqlhosts** does not define, you might need assistance from your database administrator (DBA) to create the necessary entries in this file. In addition to the **sqlhosts** file, you might also need to configure system network files to support connections. Your *IBM Informix Administrator's Guide* describes how to create a database server entry in the **sqlhosts** file.

If you are enabling single-sign on (SSO), additional steps are in "Configuring ESQL/C and ODBC Drivers for SSO" in *IBM Informix Security Guide*.

Specifying the Default Database Server

For your Informix ESQL/C application to communicate with *any* database server, you must set the **INFORMIXSERVER** environment variable to specify the name of the default database server. Therefore, the name of the default database server must exist in the **sqlhosts** file and **sqlhosts** must exist on the computer that runs the application. For more information about **sqlhosts**, see "Accessing the sqlhosts File" on page 12-4.

Sources of connection information in a Windows environment

To establish a connection to a database server, an IBM Informix ESQL/C application in a Windows environment performs the following tasks:
- Provide information about the connection with the registry, the **ifx_putenv()** function, or the **InetLogin** structure
- Use a central registry for connection information
- Perform connection authentication for the application user

In Windows environments, Informix ESQL/C obtains the configuration information from the **InetLogin** structure or the in-memory copy of the registry.

If the application has initialized a field in **InetLogin**, Informix ESQL/C sends this value to the database server. For any field the application has not set in the **InetLogin** structure, Informix ESQL/C uses the corresponding information in the Informix subkey of the registry. For more information, see "Setting and Retrieving Environment Variables in Windows Environments" on page 1-31 and "Precedence of Configuration Values" on page 1-37.

Important: Because the application needs configuration information to establish a connection, you must set any **InetLogin** configuration values before the SQL statement that establishes the connection.

In Windows environments, the registry contains the following configuration information:
- The values of the Informix environment variables

- Connection information (the **sqlhosts** and **.netrc** files define this information about UNIX operating systems)

When a client Informix ESQL/C application establishes a connection to a database server (see "Connecting to a Database Server" on page 12-10), it sends the configuration information to the database server.

Setting Environment Variables for Connection in a Windows Environment

The registry provides default values for most environment variables. For a description of environment variables and their default values, see the *IBM Informix Guide to SQL: Reference* and the *IBM Informix GLS User's Guide*. To change the value of an environment variable in the Registry, use the Environment tab of the **Setnet32** utility, which the *IBM Informix Client Products Installation Guide* describes.

For more information about how to change the environment variable for the current process, see "Setting and Retrieving Environment Variables in Windows Environments" on page 1-31. For more information about **InetLogin**, see "InetLogin structure" on page 1-33.

The sqlhosts information in a Windows environment

The registry contains the following connection information:

- The **sqlhosts** information defines a connection to an established database server.

  This information includes the name of the host computer, the type of protocol to use, and the name of the connection. The Registry stores the **sqlhosts** information in the **SqlHosts** subkey of the **Informix** key. To store **sqlhosts** information in the Registry, use the **Server Information** tab of the **Setnet32** utility.

- The **.netrc** information defines a valid user for a remote connection.

  On UNIX operating systems, this file resides in the home directory of the user and specifies the name and password for the user account. In Windows environments, the **NETRC** subkey of the **Informix** key in the Registry stores the same account information. To store **.netrc** information in the Registry, use the **Host Information** tab of the **Setnet32** utility.

The client sends network parameters to establish a connection to a database server. The first step in establishing a connection is to log onto the correct host computer. The protocol software uses the network parameters for the current database server. The client locates the network parameters for the current database server in either of the following ways:

1. If the SQL statement that requests the connection (such as a CONNECT or DATABASE) specifies the name of a database server, the client sends the network parameters for this specified database server.

   If the **InfxServer** field of **InetLogin** contains the name of the specified database server, the client checks **InetLogin** for the network parameters. Otherwise, the client obtains network parameters for that database server from the in-memory copy of the Registry.

2. If the SQL statement does not specify a database server, the client sends the network parameters for the default database server.

   If the **InfxServer** field of **InetLogin** contains the name of a database server, the client checks **InetLogin** for the network parameters. Otherwise, the client determines the default database server from the **INFORMIXSERVER** value in the in-memory copy of the Registry. It then sends network parameter values from the Registry for that database server.

Informix ESQL/C checks the network parameter fields of **InetLogin** for any of these network parameters that the application has currently set. For any fields (including the name of the default database server) that are not set, Informix ESQL/C obtains the values from the in-memory copy of the Registry. (For more information, see "Precedence of Configuration Values" on page 1-37.)

For example, the code fragment in the following figure initializes the **InetLogin** structure with information for the **mainsrvr** database server; **mainsrvr** is the default database server.

```
void *cnctHndl;
.
.
.
strcpy(InetLogin.InfxServer, "mainsrvr");
strcpy(InetLogin.User, "finance");
strcpy(InetLogin.Password, "in2money");
EXEC SQL connect to 'accounts';
.
.
.

QL connect to 'custhist@bcksrvr';
```

Figure 12-3. Code fragment that shows precedence of network parameters

When execution reaches the first CONNECT statement in Figure 12-3, the client application requests a connection to the **accounts** database on the **mainsrvr** database server. The CONNECT statement does not specify a database server, so the client sends the following network parameters for default database server:

- The default database server is **mainsrvr** because **InfxServer** is set in **InetLogin**.
- The **User** and **Password** values are finance and in2money because the application sets them in **InetLogin**.
- The **Host**, **Service**, **Protocol**, and **AskPassAtConnect** values are from the **mainsrvr** subkey of the Registry values, because the application does not set them in **InetLogin**.

The second CONNECT statement in Figure 12-3 requests a connection to the **custhist** database on the **bcksrvr** database server. For this connection, the client sends the network parameters for the specified database server, **bcksrvr**. Because the **InetLogin** structure currently contains network parameters for **mainsrvr**, the client must obtain *all* these parameters from the in-memory copy of the Registry. Therefore, the application does not use the **finance** user account for this second connection (unless the Registry specifies **User** and **Password** values of finance and in2money for the **bcksrvr** database server).

If you are enabling single-sign on (SSO), the process differs. Details and additional steps for configuration are in "Configuring ESQL/C and ODBC Drivers for SSO" in *IBM Informix Security Guide*.

Using a Central Registry
You can specify the **sqlhosts** information in one of the following locations:
- The local registry is the registry that resides on the same Windows computer as your Informix ESQL/C application.
- The central registry is a registry that two or more Informix ESQL/C applications can access to obtain **sqlhosts** information.

The central registry can be on the Domain Server or on any Windows workstation on the Microsoft network. It might be local to one application and remote to all others. A central registry enables you to maintain a single copy of the **sqlhosts** information for use by all Informix ESQL/C applications in Windows environments.

To use a central registry, you must set the **INFORMIXSQLHOSTS** environment variable on your computer. This environment variable specifies the name of the computer on which the central registry resides. To set this environment variable, you can use **Setnet32**, the **ifx_putenv()** function ("Setting and Retrieving Environment Variables in Windows Environments" on page 1-31), or the **InetLogin** structure ("InetLogin structure" on page 1-33).

In a Windows environment, an Informix ESQL/C application uses the following precedence to locate **sqlhosts** information when it requests a connection:

1. The **sqlhosts** information in the central registry, on computer that the **INFORMIXSQLHOSTS** environment variable indicates (if **INFORMIXSQLHOSTS** is set)
2. The **sqlhosts** information in the local registry

Connection Authentication Functionality in a Windows Environment

Once an Informix ESQL/C application has obtained the information about the connection (from either the registry or the **InetLogin** structure), the ESQL client-interface DLL performs the following steps:

1. It copies connection information from the **InetLogin** structure (or from the registry for undefined **InetLogin** fields) into a **HostInfoStruct** structure (see Table 12-1).
2. It passes a pointer to the **HostInfoStruct** to the **sqlauth()** function in the **esqlauth.dll** to verify connection authentication.

If **sqlauth()** returns TRUE, the connection is verified and the user can access the server computer. However, if **sqlauth()** returns FALSE, the connection is refused and access denied. By default, the **sqlauth()** function returns a value of TRUE.

The parameter passed to **sqlauth()** is a pointer to a **HostInfoStruct** structure, which the **login.h** header file defines. This structure contains the subset of the **InetLogin** fields that Table 12-1 shows.

Table 12-1. Fields of the HostInfoStruct Structure

HostInfoStruct Field	Data Type	Purpose
InfxServer	char[19]	Specifies the value for the INFORMIXSERVER network parameter
Host	char[19]	Specifies the value for the HOST network parameter
User	char[19]	Specifies the value for the USER network parameter passed into the **sqlauth()** function
Pass	char[19]	Specifies the value for the PASSWORD network parameter passed into the **sqlauth()** function
AskPassAtConnect	char[2]	Indicates whether **sqlauth()** should request a password at connection time passed into the **sqlauth()** function
Service	char[19]	Specifies the value for the SERVICE network parameter passed into the **sqlauth()** function
Protocol	char[19]	Specifies the value for the PROTOCOL network parameter passed into the **sqlauth()** function
Options	char[20]	*Reserved for future use*

Within **sqlauth()**, you can access the fields of **HostInfoStruct** with the **pHostInfo** pointer, as follows:

```
if (pHostInfo->AskPassAtConnect)
```

You can edit all the **HostInfoStruct** field values. ESQL/C, however, checks only the **User** and **Pass** fields of **HostInfoStruct**.

Figure 12-4 shows the default **sqlauth()** function, which the **esqlauth.c** file contains.

```
BOOL __declspec( dllexport ) sqlauth ( HostInfoStruct *pHostInfo )
{
    return TRUE;
}
```

Figure 12-4. Default Code for sqlauth() Function

This default action of **sqlauth()** means that Informix ESQL/C performs *no* authentication verification when it establishes a connection. To provide verification, you can customize the **sqlauth()** function. You might want to customize **sqlauth()** to perform one of the following verification tasks:

- Validation of the user name

  The function can compare the current user name against a list of valid or invalid user names.

- Prompt for a password

  The function can check the value of the **AskPassAtConnect** field in the **HostInfoStruct** structure when this field is set to Y or y. You can code **sqlauth()** to display a window that prompts the user to enter a password.

The following steps describe how to create a customized **sqlauth()** function:

1. Bring up the **esqlauth.c** source file in your system editor. This file is located in the **%INFORMIXDIR%\demo\esqlauth** directory.

2. Add to the body of the **sqlauth()** function the code that performs the desired connection verification. Of the fields in Table 12-1 on page 12-9, the **sqlauth()**

function can modify only the **User** and **Pass** fields. Make sure that **sqlauth()** returns TRUE or FALSE to indicate whether to continue with the connection request. Do *not* modify other code in this file.

Create a new version of the **esqlauth.dll** by compiling the **esqlauth.c** file and specifying the **-target:dll** (or **-wd**) command-line option of the **esql** command processor.For an example of how to define the **sqlauth()** function, see the **esqlauth.c** file in the **%INFORMIXDIR%\demo\esqlauth** directory.

Connecting to a Database Server

When an Informix ESQL/C application begins execution, it has no connections to any database server. For SQL statements to execute, however, such a connection must exist. To establish a connection to a database server, the Informix ESQL/C program must take the following actions:

- Use an SQL statement to establish a connection to the database server
- Specify, in the SQL statement, the name of the database server to which to connect

Establishing a Connection

The following two groups of SQL statements can establish connections to a database environment:

- The SQL *connection statements* are CONNECT, SET CONNECTION, and DISCONNECT. These statements conform to ANSI SQL and X/Open standards for the creation of connections.
- The SQL *database statements* include DATABASE, CREATE DATABASE, CLOSE DATABASE, and START DATABASE. These statements are an Informix-specific way to establish connections.

Important: It is recommended that you use the CONNECT, DISCONNECT, and SET CONNECTION connection statements for new applications of Version 6.0 and later. For pre-6.0 versions, the SQL database statements (such as DATABASE, START DATABASE, and CLOSE DATABASE) remain valid for compatibility with earlier versions.

The type of connection that the application establishes depends on which of these types of statements executes first in the application:

- If the first SQL statement is a connection statement (CONNECT, SET CONNECT) statement, the application establishes an *explicit connection*.
- If the first statement is an SQL database statement (DATABASE, CREATE DATABASE, START DATABASE), the application establishes an *implicit connection*.

The following sections briefly describe explicit and implicit connections. For more information, see the entries for the CONNECT and DISCONNECT statements in the *IBM Informix Guide to SQL: Syntax*.

The Explicit Connection: When you use the CONNECT statement to connect to a database environment, you establish an *explicit connection*. The application connects directly to the database server that you specify. If you do not specify the name of a database server in the CONNECT statement, the application establishes an explicit connection to the default database server (that the **INFORMIXSERVER** environment variable identifies).

An explicit connection enables an application to support *multiple connections* to one or more database environments. Although the application can connect to several database environments during its execution, only one connection can be *current* at a time. *Dormant* connections are connections that the application has established but is not currently using. The application must have a current connection to execute SQL statements.

The following SQL connection statements establish and manage explicit connections:

- The CONNECT statement establishes an explicit connection between a database environment and the application.
- The SET CONNECTION statement switches between explicit connections. It makes a dormant connection the current connection.
- The DISCONNECT statement terminates a connection to a database environment.

These connection statements provide the following benefits, which allow you to create more portable applications:

- Compliance with ANSI and X/Open standards for database connections
- A uniform syntax for local and remote data access for use in a distributed client-server environment
- Support for multiple connections within a single application

Because the CONNECT, DISCONNECT, and SET CONNECTION statements include Informix extensions to ANSI-standard syntax, these statements generate ANSI-extension warning messages at the following times:

- At runtime, if you have set the **DBANSIWARN** environment variable
- At compile time, if you have compiled the Informix ESQL/C source file with the **-ansi** preprocessor option

The Informix ESQL/C application, *not* the database server, processes these connection statements. Therefore, the application cannot use them in a PREPARE or an EXECUTE IMMEDIATE statement.

Important: Use of the DATABASE, CREATE DATABASE, START DATABASE, CLOSE DATABASE, and DROP DATABASE statements is still valid with an explicit connection. However, in this context, refer only to databases that are local to the current connection in these statements; do not use the **@server** or **//server** syntax.

The Implicit Connection: When one of the following SQL statements is the first SQL statement that the application executes, the statement establishes an *implicit connection*:

- The DATABASE statement creates an implicit connection to a database environment and opens the specified database.
- The CREATE DATABASE statement creates an implicit connection and creates a new database.
- The DROP DATABASE statement creates an implicit connection and drops (removes) the specified database.
- A single-statement PREPARE of one of the preceding statements also establishes an implicit connection.

When you execute one of the preceding statements, the application first connects to the default database server (that the **INFORMIXSERVER** environment variable indicates). The default database server parses the database statement. If the statement specifies the name of a database server, the application then connects to the specified database server. To establish an implicit connection to a specified database server, an application must therefore connect to *two* database servers. An explicit connection only requires a connection to a single database server, and therefore involves less overhead.

If an implicit connection already exists, these database statements close it before they establish the new connection. The new implicit connection remains open after the SQL statement completes. This behavior contrasts with explicit connections (12-10), which allow multiple connections to the same or to a different database environment.

The CLOSE DATABASE statement closes the database and, in pre-Version 6.0 applications, also closes the implicit connection to the database. If you precede these statements with a CONNECT, each can also operate in the context of the current explicit connection.

Use of an implicit connection provides a smooth migration path for older applications into the connection-oriented environment that CONNECT, DISCONNECT, and SET CONNECTION statements support. For more information about implicit connections, see the CONNECT statement in the *IBM Informix Guide to SQL: Syntax*.

Summary of Connection Types: Table 12-2 summarizes the methods that Informix ESQL/C supports to connect to a database server.

Table 12-2. Statements and Functions That Start the Database Server

SQL Statement or ESQL/C Function	Type of Connection		Effect on a Connection to the Database Server	
	Implicit	Explicit	Establishes a Connection	Opens a Database
If first SQL statement in the program is:				
DATABASE	Y		Y	Y
CREATE DATABASE	Y		Y	Y
START DATABASE	Y		Y	Y
DROP DATABASE	Y		Y	
sqlstart()	Y		Y	
CONNECT TO DEFAULT		Y	Y	
CONNECT TO '@*servername*'		Y	Y	
CONNECT TO '*dbname*'		Y	Y	Y
CONNECT TO '*dbname@servername*'		Y	Y	Y

For information about the **sqlstart()** library function, see B-231.

Establishing an Explicit Connection in a Windows Environment
This section provides the following information specific to Windows environments:
- A brief explanation of the uses of an explicit connection in the Windows environment

- A description of the Informix ESQL/C connection library functions that support explicit connections

When to Use an Explicit Connection in a Windows Environment: With an *implicit* connection, one connection to the database server can exist for each Informix ESQL/C module and this connection cannot be shared. An *explicit* connection allows multiple connections within a client application. You might want to design an application that needs to perform multiple connections for one of the following reasons:

- When you want multiple Informix ESQL/C modules (either **.exe** or **.dll**) to use the same connection to manipulate database data

  Figure 12-5 shows scenarios in which multiple applications use the same connection to a database server.

- When you want one Informix ESQL/C module to create two or more connections to one or more databases, which includes sharing an ESQL DLL between two C applications

  Figure 12-6 shows a single application that establishes connections to multiple database servers.

Figure 12-5 shows the following two scenarios in which multiple applications share a single connection to the database server:

- The scenario on the left requires that **APP1.EXE** establish an explicit connection to the **dbserverA** database server. Once this connection is established, APP1 can pass the connection information required to set the connection in the APP2 DLL.

- The scenario on the right requires that **APP3.EXE** establish an explicit connection to the **dbserverB** database server. Both the APP4 and APP5 DLLs can share this connection when APP3 passes the appropriate connection information.
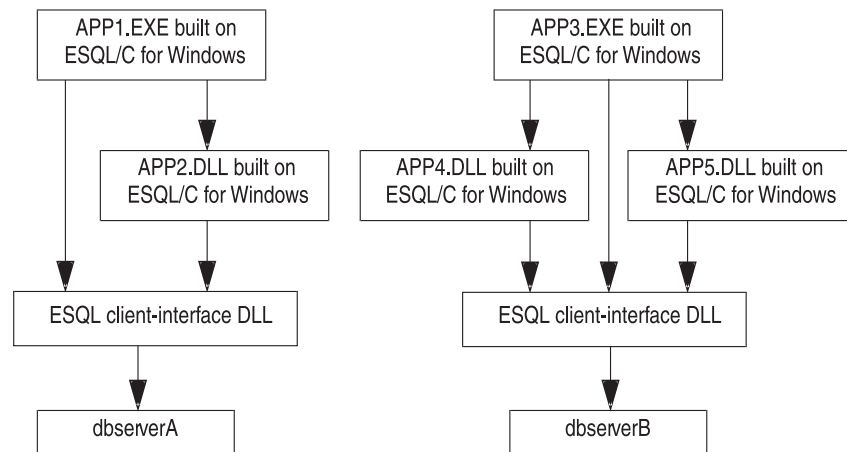
Figure 12-5. Two Scenarios in Which Multiple Applications Use a Single Connection to a Database Server

You can also use explicit connections if you want one application to establish connections to two separate database servers at the same time, as Figure 12-6 shows.

```
┌─────────────────────┐
│  APP8.EXE built on  │
│  ESQL/C for Windows │
└─────────────────────┘
           │
           ▼
┌─────────────────────┐
│ ESQL client-interface DLL │
└─────────────────────┘
      │           │
      ▼           ▼
┌──────────┐  ┌──────────┐
│ dbserverA │  │ dbserverB │
└──────────┘  └──────────┘
```
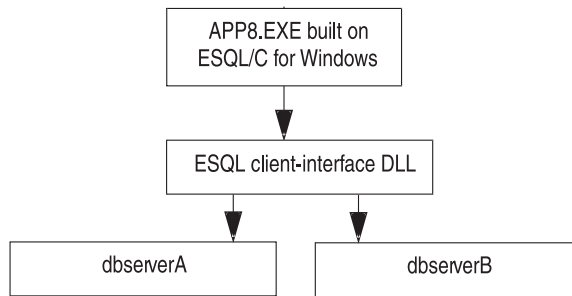
Figure 12-6. One Application That Uses Connections to More Than One Database Server at the Same Time

Using Password Encryption

When a client application sends a password to the database server for authentication the password is not encrypted unless you request password encryption through the *simple password communications support module* (SPWDCSM). You activate password encryption by specifying it on the configuration for the database server name, or an alias, in the **sqlhosts** file or registry. To activate password encryption, specify the following value in the `Options` field of the **sqlhosts** entry:

`csm=(SPWDCSM)`

The client or the database server uses the string "SPWDCSM" as a key to look up the entry that describes the CSM in the CSS/CSM configuration file.

Once activated, the SPWDCSM requires a password, effectively overriding any trusted host mechanisms. It is contradictory to specify a password encryption mechanism if a trusted host policy is in place.

For more information about implementing password encryption and for information about communication support services (CSS), see your *IBM Informix Administrator's Guide*.

Using Pluggable Authentication Modules (PAM)

To use a Pluggable Authentication Module (PAM) service for client-server authentication you must re-write your client application so that it registers a callback function. The callback function must support any challenge-response mechanisms of the PAM service you intend to use. For the details of how to use PAM, see the *IBM Informix Security Guide*.

The demonstration program **pamdemo.ec** is provided as an example of the use of a callback function.

Using LDAP Authentication

You can use Lightweight Directory Access Protocol (LDAP) authentication on Windows with Informix ESQL/C. Use the LDAP Authentication Support module when you want to use an LDAP server to authenticate your system users. The module contains source code that you can modify for your specific situation. For information about installing and customizing the LDAP Authentication Support module, see the *IBM Informix Security Guide*.

Using Multiplexed Connections

A *multiplexed connection* enables an Informix ESQL/C application to establish multiple connections to different databases on the same database server, using a minimum amount of communications resources. When you initiate a multiplexed connection, the database server uses a single connection to the client for multiple SQL connections (CONNECT statement). Without multiplexing, each SQL connection creates a new database-server connection.

Client Requirements for Execution: To implement a multiplexed connection, set the multiplexing option, in the client **sqlhosts** file or registry, on the **dbservername** parameter of the database server to which you will connect. To specify multiplexing, set the m option to 1. The following **dbservername** parameter specifies a multiplexed connection to the **personnel** database server:

Servername	nettype	hostname	servicename	options
personnel	onsoctcp	corp	prsnl_ol	m=1

Setting the multiplexing option to zero (m = 0), which is the default, disables multiplexing for the specified database server.

To use multiplexed connections for any application that was compiled prior to version 9.13 of Informix ESQL/C for UNIX or version 9.21 of Informix ESQL/C for Windows, you must relink it. Applications that you compiled prior to these versions of Informix ESQL/C *can* connect to a multiplexing database server, however. The database server establishes a non-multiplexed connection in this case.

On Windows platforms, in addition to setting the multiplexing option in the **sqlhosts** registry you must also define the ifx_session_mux environment variable. If you do not define the **ifx_session_mux** environment variable, the database server will ignore the multiplexing option and will not multiplex connections.

Warning: On Windows, a multithreaded application must not use the multiplexed-connection feature. If a multithreaded application enables the multiplexing option in the **sqlhosts** registry entry and also defines the **IFX_SESSION_MUX** environment variable, it can produce disastrous results, including crashing and data corruption.

If a multithreaded application and a single-threaded application are running on the same Windows computer, the single-threaded application can use a multiplexed connection in the following two ways:

- Use a different **sqlhosts** file or registry
- Use a dbserver alias in the **sqlhosts** file that does not specify the multiplexing option

  For example, you could use the following configuration.

Servername	Nettype	Host Name	Servicename	Options
personnel	onsoctcp	corp	prsnl_01	m=1
personnel_nomux	onsoctcp	corp	prsnl_02	

Any multithreaded application connecting to the *personnel* server should use the servername *personnel_nomux* while single-threaded applications can continue to use the servername *personnel*.

For information about the requirements to implement multiplexed connections on the database server, see your *IBM Informix Administrator's Guide*.

Limitations for Multiplexed Connections: Informix ESQL/C imposes the following limitations on multiplexed connections:

- Shared memory connections are not supported.
- Multithreaded applications are not supported.
- The database server ignores the **sqlbreak()** function on a multiplexed connection. If you call it, the database server does not interrupt the connection and does not return an error.

Identifying the Database Server

To connect to a database environment (with, for example, a CONNECT statement), the Informix ESQL/C application can identify the database server in one of two ways:

- The application can specify the name of the database server in the SQL statement. Such a database server is a *specific* database server.
- The application can omit the name of the database server in the SQL statement. Such a database server is the *default* database server. The **INFORMIXSERVER** environment variable specifies the name of the default database server.

A Specific Database Server

An Informix ESQL/C application can establish a connection to a *specific* database server when it lists the database server name, and optionally the database name, in an SQL statement, as follows:

- The CONNECT statement establishes an *explicit connection* to the database server.

  Each of the following CONNECT statements establishes an explicit connection to a database server that is called **valley**:

  ```
  EXEC SQL connect to 'stores7@valley';
  EXEC SQL connect to '@valley';
  ```

- When one of the SQL database statements (such as DATABASE or START DATABASE) is the first SQL statement of the application, it can establish an *implicit connection*.

  Each of the following SQL statements establishes an implicit connection to the **stores7** database in a specific database server that is called **valley**:

  ```
  EXEC SQL database '//valley/stores7';
  EXEC SQL database stores7@valley;
  ```

  For the UNIX operating system, use the following statement:

  ```
  EXEC SQL database '/usr/dbapps/stores7@valley';
  ```

  For a Windows environment, use the following statement:

  ```
  EXEC SQL database 'C:\usr\dbapps\stores@valley';
  ```

For more information about connections to a specific database server, see the CONNECT and DATABASE statements in the *IBM Informix Guide to SQL: Syntax*.

The Default Database Server

An Informix ESQL/C application can establish a connection to a default database server when it omits the database server name from the database environment in an SQL statement, as follows:

- The CONNECT statement can establish an *explicit default connection* with the keyword DEFAULT or when it omits the database server name.

Each of the following CONNECT statements establishes an explicit default connection:

```
EXEC SQL connect to 'stores7';
EXEC SQL connect to default;
```

In a UNIX operating system, use the following statement:

```
EXEC SQL connect to '/usr/dbapps/stores7';
```

In a Windows environment, use the following statement:

```
EXEC SQL connect to 'C:\usr\dbapps\stores7';
```

- When one of the SQL database statements (such as DATABASE or START DATABASE) is the first SQL statement of the application, it can establish an *implicit default connection*.

  Each of the following SQL statements establishes an implicit default connection to a database that is called **stores7** on the default database server:

```
EXEC SQL database stores7;
EXEC SQL start database stores7 with no log;
```

The **INFORMIXSERVER** environment variable determines the name of the database server. For more information, see "Specifying the Default Database Server" on page 12-5.

Important: You must set the **INFORMIXSERVER** environment variable even if the application does not establish a default connection.

You can also use the **DBPATH** environment variable to specify a list of database server names to use as default database servers. The application searches for these database servers after it searches for the database server that **INFORMIXSERVER** specifies. For instructions on how to set the **INFORMIXSERVER** or **DBPATH** environment variable with the name of the default database server, see the *IBM Informix Guide to SQL: Reference*.

Interacting with the Database Server

Within your Informix ESQL/C program, you can interact with the database server in the following ways:

- Start a new database server process. This process does not exist when an application begins execution.
- Switch between multiple connections. An application can establish several connections.
- Identify an explicit connection. An application can obtain the name of the database server and connection.
- Identify the databases that the database server of the current connection can access.
- Check on the status of the database server process. For some actions the database server must be busy, for others the database server must be idle.
- Detaching from the current connection. An application must detach a child process from the current connection.
- Interrupt the database server process. If an SQL request executes for a long time, the application can interrupt it.
- Terminate the database server process. The application can close an unused connection to free resources.

Informix ESQL/C supports Secure Sockets Layer (SSL) connections. For information about using the SSL protocol, see the "Secure Sockets Layer Communication Protocol" section of the *IBM Informix Security Guide*.

Determining Features of the Database Server

You can check on features of the database server after you execute one of the following SQL statements.

CONNECT	DATABASE
CREATE DATABASE	SET CONNECTION

When the database server establishes a connection with one of these statements, it can obtain the following information about the database server:

- Is a long identifier or long user name truncated?
- Does the open database use a transaction log?
- Is the open database an ANSI-compliant database?
- What is the database server name?
- Does the database store the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types)?
- Is the database server in secondary mode? (If the database server is in secondary mode, it is a secondary server in a data-replication pair and is available only for read operations.)

Does the value of the **DB_LOCALE** environment variable set by the client application match the value of the database locale of the open database? The following table summarizes the values that the **SQLSTATE** variable and the **sqlca** structure take to indicate these conditions.

Database Feature	SQLSTATE Value	sqlca Value
Long identifier or long username has truncated	"01004"	**sqlca.sqlwarn.sqlwarn1** is 'W'
Database has transactions	"01I01"	**sqlca.sqlwarn.sqlwarn1** is 'W'
Database is ANSI compliant	"01I03"	**sqlca.sqlwarn.sqlwarn2** is 'W'
Database server is *not* IBM Informix SE	"01I04"	**sqlca.sqlwarn.sqlwarn3** is 'W'
FLOAT represented as DECIMAL	"01I05"	**sqlca.sqlwarn.sqlwarn4** is 'W'
Database server in secondary mode	"01I06"	**sqlca.sqlwarn.sqlwarn6** is 'W'
Mismatched database locales	*undefined*	**sqlca.sqlwarn.sqlwarn7** is 'W'

The **SQLSTATE** variable might return multiple exceptions after these connection statements. For more information about the **SQLSTATE** variable and the **sqlca** structure, see Chapter 11, "Exception handling," on page 11-1.

Switching Between Multiple Database Connections

An Informix ESQL/C application can make a number of simultaneous database connections with a CONNECT statement. These connections can be to several database environments or can be multiple connections to the same database environment. To switch between connections, the Informix ESQL/C application must follow these steps:

1. Establish a connection with the CONNECT STATEMENT
2. Handle any active transactions

   If the current connection has an active transaction, you can switch connections only if the CONNECT statement with the WITH CONCURRENT TRANSACTION clause establishes the current connection.
3. Make a connection current with the SET CONNECTION or CONNECT statement

Making a Connection Current®

When multiple connections exist, the application can only communicate with one connection at a time. This connection is the *current* connection. All other established connections are *dormant*. Your application can make another connection current with either of the following connection statements:

- The CONNECT statement establishes a *new* connection and makes the connection current.
- The SET CONNECTION statement switches to a *dormant* connection and makes the connection current.

When you make a connection dormant and then current again, you perform an action similar to when you disconnect and then reconnect to the database environment. However, if you make a connection dormant you can typically avoid the need for the database server to perform authentication again, and thereby save the cost and use of resources that are associated with the connection.

Tip: A thread-safe Informix ESQL/C application can have multiple current connections, one current connection per thread. However, only one current connection is active at a time.

For more information about thread-safe applications, see Chapter 13, "Using Informix libraries," on page 13-1.

For more information, see the entries for CONNECT and SET CONNECTION in the *IBM Informix Guide to SQL: Syntax*.

Handling Transactions

If the CONNECT statement with the WITH CONCURRENT TRANSACTION clause has established the connection, the application can switch to another connection even if the current connection contains an active transaction.

For an example of an Informix ESQL/C program that uses the WITH CONCURRENT TRANSACTION clause, see the CONNECT statement in the *IBM Informix Guide to SQL: Syntax*.

For connections that are *not* established with the CONNECT...WITH CONCURRENT TRANSACTION statement, the application must end the active transaction *before* it switches to another connection. Any attempt to switch while a transaction is active causes the CONNECT or SET CONNECTION statement to fail (error number -1801). The transaction in the current connection remains active.

To maintain the integrity of database information, explicitly end the active transaction in one of the following ways:

- Commit the transaction with the COMMIT WORK statement to ensure that the database server saves any changes that have been made to the database within the transaction.

- Roll back the transaction with the ROLLBACK WORK statement to ensure that the database server backs out any changes that have been made to the database within the transaction.

The COMMIT WORK or ROLLBACK WORK statement applies *only* to the transaction that is within the current connection, not to transactions that are in any dormant connection. For more information about how to handle transactions, see the COMMIT WORK and ROLLBACK WORK statements. For more information about the WITH CONCURRENT TRANSACTION clause of CONNECT, see the CONNECT and SET CONNECTION statements. Entries for these SQL statements can be found in the *IBM Informix Guide to SQL: Syntax*.

Identifying an Explicit Connection

From within an Informix ESQL/C application, you can obtain the name of the database server and the name of the explicit connection with the GET DIAGNOSTICS statement. When you use GET DIAGNOSTICS after an SQL connection statement (CONNECT, SET CONNECTION, and DISCONNECT), GET DIAGNOSTICS puts this database server information in the diagnostics area in the SERVER_NAME and CONNECTION_NAME fields.

Figure 12-7 shows a code fragment that saves connection information in the **srvrname** and **cnctname** host variables.

```
EXEC SQL connect to :dbname;
if(!strncmp(SQLSTATE, "00", 2)
    {
    EXEC SQL get diagnostics exception 1
        :srvrname = SERVER_NAME, :cnctname = CONNECTION_NAME;
    printf("The name of the server is '%s'\n", srvrname);
    }
```

Figure 12-7. Code Fragment That Saves Connection Information

For more information, see the entry for GET DIAGNOSTICS in the *IBM Informix Guide to SQL: Syntax*.

From within an Informix ESQL/C application, you can obtain the name of the current connection with the **ifx_getcur_conn_name()** function. This function returns the name of the current connection into a user-defined character buffer. The function is useful to determine the current connection among a group of active connections in a Informix ESQL/C application that has multiple threads.

For example, the following code consists of a callback function, **cb()**, that two **sqlbreakcallback()** calls use in two different threads:

```
void
cb(mint status)
{
   mint res;
   char *curr_conn = ifx_getcur_conn_name();

   if (curr_conn && strcmp(curr_conn, "con2") == 0)
      {
      res = sqlbreak();
      printf("Return status of sqlbreak(): %d\n", res);
      }
}

void
thread_1()
```

```
{
EXEC SQL BEGIN DECLARE SECTION;
   mint res;
EXEC SQL END DECLARE SECTION;

   EXEC SQL connect to 'db' as 'con1' ;
   sqlbreakcallback(100, cb);
   EXEC SQL SELECT count(*) INTO :res FROM x, y;
   if (sqlca.sqlcode == -213)
      printf("Connection con1 fired an sqlbreak().\n");
   printf("con1: Result of count(*) = %d\n", res);
   EXEC SQL set connection 'con1' dormant ;
}

void
thread_2()
{
EXEC SQL BEGIN DECLARE SECTION;
   mint res;
EXEC SQL END DECLARE SECTION;

   EXEC SQL connect to 'db' as 'con2' ;
   sqlbreakcallback(100, cb);
   EXEC SQL SELECT count(*) INTO :res FROM x, y;
   if (sqlca.sqlcode == -213)
      printf("Connection con2 fired an sqlbreak().\n");
   printf("con2: Result of count(*) = %d\n", res);
   EXEC SQL set connection 'con2' dormant ;
}
```

The **cb()** callback function uses the **ifx_getcur_conn_name()** to check which
connection is current. For the syntax of **ifx_getcur_conn_name()**, see B-72. For
more information about current and active connections, see "Concurrent active
connections" on page 13-10.

Obtaining Available Databases

From within an Informix ESQL/C application, you can obtain the name of the
databases that are available from a specified database server with the **sqgetdbs()**
function. This function returns the names of the databases that are available in the
database server of the current connection. For more information about **sqgetdbs()**,
see 12-31.

Checking the Status of the Database Server

Some interactions with the database server cannot execute unless the database
server is idle. Other actions assume that the database server is busy processing a
request. You can check whether the database server is currently processing an SQL
request with the **sqldone()** function. This function returns zero (0) if the database
server is idle and a negative value if it is busy. For more information about
sqldone(), see B-228.

Detaching from a Connection

When your application forks a process, the child process inherits the database
connections of the parent. If you leave these connections open, both parent and
child processes use the same connection to communicate with the same database
server. Therefore, the child process needs to establish a separate database
connection.

To establish a separate database connection for the child process:

1. Call **sqldetach()** to detach the child process from the database server connection in the parent process.
2. Establish a new connection in the child process (if one is needed).

For more information about the **sqldetach()** library function, see B-228.

Interrupting an SQL Request

Sometimes you might need to cancel an SQL request. If, for example, you inadvertently provide the wrong search criteria for a long query, you want to cancel the SELECT statement rather than wait for unneeded data. While the database server executes an SQL request, the Informix ESQL/C application is blocked. To regain control, the application must *interrupt* the SQL request.

To interrupt the database server, you can use the **sqlbreak()** library function. You might want to interrupt an SQL request for some of the following reasons:

• The application end user wants to terminate the current SQL request.

• The current SQL request has exceeded some timeout interval.

Important: The application must handle any open transactions, cursors, and databases after it interrupts an SQL request.

The following sections summarize how to handle each of these types of interrupt. For more information about the **sqlbreak()** function, see 12-30.

Interruptible SQL Statements

You cannot cancel all SQL statements. Some types of database operations are not interruptible and others cannot be interrupted at certain points. An Informix ESQL/C application can interrupt the following SQL statements.

SELECT	OPEN	ALTER TABLE
UPDATE	EXECUTE FUNCTION	ALTER INDEX
DELETE	CREATE TABLE	EXECUTE PROCEDURE
INSERT	CREATE INDEX	

In addition to the preceding statements, you can also cancel the operation of a loop as it executes within a SPL routine.

The Informix ESQL/C application and the database server communicate through *message requests*. A message request is the full round trip of the message that initiates an SQL task. It can consist of the message that the application sends to the database server as well as the message that the database server sends back in reply. Alternatively, a message request can consist of the message that the database server sends to the application as well as the message that the application sends in acknowledgment.

Most SQL statements require only one message request to execute. The application sends the SQL statement to the database server and the database server executes it. However, an SQL statement that transfers large amounts of data (such as a SELECT, an INSERT, or a PUT), can require more than one message request to execute, as follows:

• In the first message request, the application sends the SQL statement to the database server to execute.

- In subsequent message requests, the database server fills a buffer with data and then sends this data to the application. The size of the buffer determines the amount of data that the database server sends in a single message request.

In addition, the OPEN statement always requires two message requests.

The database server decides when to check for an interrupt request. Therefore, the database server might not immediately terminate execution of an SQL statement and your application might not regain control as soon as it sends the interrupt request.

Allowing a User to Interrupt

When the database server processes a large query, you might want to allow the user to interrupt the query request with the Interrupt key (usually CTRL-C). To do this, you must set up a *signal-handler function*. The signal-handler function is a user-defined function that the application process calls when it receives a specific signal.

To allow the user to interrupt an SQL request, you define a signal-handler function for the SIGINT signal. This function must have the following declaration:

```
void sigfunc_ptr();
```

The user-defined signal-handler function can contain the Informix ESQL/C control functions **sqlbreak()** and **sqldone()**. For more information about the **sqlbreak()** and **sqldone()** functions, see B-220 and B-228. If you use any other Informix ESQL/C control function or any SQL statement in the signal handler while the database server is processing, Informix ESQL/C generates an error (-439).

The Informix ESQL/C application must determine how to continue execution after the signal handler completes. One possible method is to set up a nonlocal *go to* with the **setjmp()** and **longjmp()** system functions. These functions work together to support low-level interrupts, as follows:

- The **setjmp()** function saves the current execution environment and establishes a *return point* for execution after the **longjmp()** call.
- The **longjmp()** call resides in the signal-handler function. Use **longjmp()** in a signal-handling function only if **sqldone()** returns 0 (the database server is idle).

See your UNIX operating system documentation for more information about the **setjmp()** and **longjmp()** system functions.

To associate the user-defined signal handler with a system signal, use the **signal()** system function, as follows:

```
signal(SIGINT, sigfunc_ptr);
```

When the Informix ESQL/C application receives the SIGINT signal, it calls the function that *sigfunc_ptr* indicates. For more information about the **signal()** system function, see your UNIX operating system documentation.

To disassociate the signal-handler function from the SIGINT signal, call **signal()** with SIG_DFL as the function pointer, as follows:

```
signal(SIGINT, SIG_DFL);
```

SIG_DFL is the default signal-handling action. For the SIGINT signal, the default action is to stop the process and to generate a core dump. You might instead want to specify the SIG_IGN action to cause the application to ignore the signal.

Important: On most systems, the signal handler remains in effect after the application catches the signal. On these systems, you need to disassociate the signal handler explicitly if you do not want it to execute the next time the same signal is caught.

On a few (mostly older) systems, however, when a signal handler catches a signal, the system reinstates the SIG_DFL action as the handling mechanism. On these systems, it is up to the signal handler to reinstate itself if you want it to handle the same signal the next time the signal is caught. For information about how your system handles signals, check your system documentation.

Setting Up a Timeout Interval

When the database server processes a large query, you might want to prompt the user periodically to determine whether to continue the request. To do this, you can use the **sqlbreakcallback()** function to provide the following information:

- A *timeout interval* is the period of time to wait for an SQL request to execute before the application regains control.
- A *callback function* is the user-defined function to call each time the timeout interval has elapsed.

Warning: Do not use the **sqlbreakcallback()** function if your Informix ESQL/C application uses shared memory (**olipcshm**) as the nettype in a connection to an instance of the database server. Shared memory is not a true network protocol and does not handle the nonblocking I/O that support for a callback function requires. When you use **sqlbreakcallback()** with shared memory, the function call appears to register the callback function successfully (it returns zero), but during SQL requests, the application never calls the callback function.

The Timeout Interval: With the **sqlbreakcallback()** function, you specify a *timeout interval*. A timeout interval is the amount of time (in milliseconds) for which the database server can process an SQL request before the application regains control. The application then calls the callback function that you specify and executes it to completion.

Once the callback function completes, the application resumes its wait until one of the following actions take place:

- The database server returns control to the application under one of the following conditions:
  - It has completed the SQL request. The database server returns the status of the request in the SQLCODE and SQLSTATE variables.
  - It has discontinued processing of the SQL request because it has received an interrupt request from the **sqlbreak()** function in the callback function. For more information about how the database server responds to **sqlbreak()**, see 12-30.
- The next timeout interval elapses. When the application resumes execution, it calls the callback function again.

  The application calls the callback function each time the timeout interval elapses until the database server completes the request or is interrupted.

The Callback Function: With the **sqlbreakcallback()** function, you also specify a *callback function* to be called at several points in the execution of an SQL request. A

callback function is a user-defined Informix ESQL/C function that specifies actions to take during execution of an SQL request. This function must have the following declaration:

```
void callbackfunc(status)
mint status;
```

The **integer** *status* variable identifies at what point in the execution of the SQL request the callback function was called. Within the callback function, you can check this *status* variable to determine at which point the function was called. Table 12-3 summarizes the valid *status* values.

Table 12-3. Status Values of a Callback Function

Point at Which Callback Is Called	Callback Argument Value
After the database server has completed the SQL request	0
Immediately after the application sends an SQL request to the database server	1
While the database server is processing an SQL request, after the timeout interval has elapsed	2

Within the callback function, you might want to check the value of the *status* argument to determine what actions the function takes.

Tip: When you register a callback function with **sqlbreakcallback()**, the application calls the callback function each time it sends a message request. Therefore, SQL statements that require more than one message request cause the application to call the callback function more than once.

For more information about message requests, see "Interruptible SQL Statements" on page 12-22.

The callback function, and any of its subroutines, can contain only the following Informix ESQL/C control functions:

- The **sqldone()** library function determines whether the database server is still busy.

   If **sqldone()** returns error **-439**, the database server is still busy and you can proceed with the interrupt. For more information about the **sqldone()** function, see B-228.

- The **sqlbreakcallback()** library function disassociates the callback function from the timeout interval.

   Call **sqlbreakcallback()** with the following arguments:

   ```
   sqlbreakcallback(-1L, (void *)NULL);
   ```

   This step is not necessary if you want the callback function to remain for the duration of the current connection. When you close the current connection, you also disassociate the callback function.

- The **sqlbreak()** library function interrupts the execution of the database server.

If you use any Informix ESQL/C control function other than those in the preceding list, or if you use any SQL statement while the database server is processing, Informix ESQL/C generates an error (-439).

If the application calls a callback function because a timeout interval has elapsed, the function can prompt the user for whether to continue or cancel the SQL request, as follows:

- To continue execution of the SQL request, the callback function skips the call to **sqlbreak()**.

  While the callback function executes, the database server continues processing its SQL request. Once the callback function completes, the application waits for another timeout interval before it calls the callback function again. During this interval, the database server continues execution of the SQL request.

- To cancel the SQL request, the callback function calls the **sqlbreak()** function, which sends an interrupt request to the database server.

  Execution of the callback function continues immediately after **sqlbreak()** sends the request. The application does not wait for the database server to respond until it completes execution of the callback function.

When the database server receives the interrupt request signal, it determines if the current SQL request is interruptible (see 12-22). If so, the database server discontinues processing and returns control to the application. The application is responsible for the graceful termination of the program; it must release resources and roll back the current transaction. For more information about how the database server responds to an interrupt request, see the description of **sqlbreak()** in 12-30.

Use the **sqlbreakcallback()** function to set the timeout interval (in milliseconds) and to register a callback function, as follows:

```
sqlbreakcallback(timeout, callbackfunc_ptr);
```

This *callbackfunc_ptr* must point to a callback function that you already defined (see 12-24). Within the calling program, you must also declare this function, as follows:

```
void callbackfunc_ptr();
```

Important: You must register the callback function after you establish the connection and before you execute the first embedded SQL statement that you want to cancel. Once you close the connection, the callback function is no longer registered.

For information about the **sqlbreakcallback()** function, see B-221. The **timeout** demonstration program, which 12-30 describes, uses the **sqlbreakcallback()** function to establish a timeout interval for a database query.

Error Checking During Data Transfer

The **IFX_LOB_XFERSIZE** environment variable is used to specify the number of kilobytes in a CLOB or BLOB to transfer from a client application to the database server before checking whether an error has occurred. The error check occurs each time the specified number of kilobytes is transferred. If an error occurs, the remaining data is not sent and an error is reported. If no error occurs, the file transfer will continue until it finishes.

The valid range for **IFX_LOB_XFERSIZE** is from 1 to 9223372036854775808 kilobytes. The **IFX_LOB_XFERSIZE** environment variable is set on the client.

For more information about **IFX_LOB_XFERSIZE**, see the *IBM Informix Guide to SQL: Reference*.

Terminating a connection

An Informix ESQL/C program can use the following statements and functions to close a connection:

- The CLOSE DATABASE statement closes a database. For pre-Version 6.0 applications, it also closes the connection. For applications of Version 6.0 and later, the connection remains open after the CLOSE DATABASE statement executes.
- The **sqlexit()** library function closes all current connections, implicit and explicit. If you call **sqlexit()** when any databases are still open, the function causes any open transactions to be rolled back.
- The **sqldetach()** library function closes the database server connection of the child process. It does not affect the database server connection of the parent process.
- The DISCONNECT statement closes a specified connection. If a database is open, DISCONNECT closes it before it closes the connection. If transactions are open, the DISCONNECT statement fails.

For more information about the CLOSE DATABASE and DISCONNECT statements, see of the *IBM Informix Guide to SQL: Syntax*. For information about the **sqldetach()** and **sqlexit()** library functions, see B-223 and B-228.

Optimized Message Transfers

Informix ESQL/C provides a feature called *optimized message transfers*, which allow you to minimize message transfers with the database server for most Informix ESQL/C statements. Informix ESQL/C accomplishes optimized message transfers by chaining messages together and even eliminating some small message packets. When the optimized message transfer feature is enabled, Informix ESQL/C *expects* that SQL statements will succeed. Consequently, Informix ESQL/C chains, and in some cases eliminates, confirmation messages from the database server.

Restrictions on Optimized Message Transfers

Informix ESQL/C does not chain the following SQL statements even when you enable optimized message transfers:

- COMMIT WORK
- DESCRIBE
- EXECUTE
- FETCH
- FLUSH
- PREPARE
- PUT
- ROLLBACK WORK
- SELECT INTO (singleton SELECT)

When Informix ESQL/C reaches one of the preceding statements, it flushes the message out to the database server. Informix ESQL/C then continues message chaining for subsequent SQL statements. Only SQL statements that require network traffic cause Informix ESQL/C to flush the message queue.

SQL statements that do not require network traffic, such as the DECLARE statement, do not cause Informix ESQL/C to send the message queue to the database server.

Enabling Optimized Message Transfers

To enable optimized message transfers, or message chaining, you must set the following variables in the client environment:

1. Set the **OPTMSG** environment variable at run time to enable optimized message transfers for all qualifying SQL statements.

2. Set the **OptMsg** global variable within an Informix ESQL/C application to control which SQL statements use message chaining.

Setting the OPTMSG Environment Variable

The **OPTMSG** environment variable enables the optimized message transfers for all SQL statements in the application. You can assign the following values to the **OPTMSG** environment variable:

1   This value enables optimized message transfers, implementing the feature for any connection that is subsequently.

0   This value disables optimized message transfers. (Default)

The default value of the **OPTOMSG** environment variable is 0 (zero). Setting **OPTMSG** to 0 (zero) explicitly disables message chaining. You might want to disable optimized message transfers for statements that require immediate replies, or for debugging purposes.

To enable optimized message transfers, you must set **OPTMSG** before you start the Informix ESQL/C application.

On UNIX operating systems, you can set **OPTMSG** within the application with the **putenv()** system call (as long as your system supports the **putenv()** function). The following call to **putenv()**, for example, enables optimized message transfers:

```
putenv("OPTMSG=1");
```

In Windows environments, you can set **OPTMSG** within the application with the **ifx_putenv()** function. The following call to **ifx_putenv()**, for example, enables optimized message transfers:

```
ifx_putenv("OPTMSG=1");
```

When you set **OPTMSG** within an application, you can activate or deactivate optimized message transfers for each connection or within each thread. To enable optimized message transfers, you must set **OPTMSG** before you establish a connection.

Setting the OptMsg Global Variable

The **OptMsg** global variable is defined in the Informix ESQL/C **sqlhdr.h** header file.

After you set the **OPTMSG** environment variable to 1, you must set the **OptMsg** global variable to specify whether message chaining takes effect for each subsequent SQL statement. You can assign the following values to **OptMsg**:

1   This value enables message chaining for every subsequent SQL statement.

0   This value disables message chaining for every subsequent SQL statement.

With the **OPTMSG** environment variable set to 1, you must still set the **OptMsg** global variable to 1 to enable the message chaining. If you omit the following statement from your program, Informix ESQL/C does not perform message chaining:

```
OptMsg = 1;
```

When you have set the **OPTMSG** environment variable to 1, you might want to disable message chaining for the following reasons:

- Some SQL statements require immediate replies.

  See "Restrictions on Optimized Message Transfers" on page 12-27 for more information about these SQL statements. Re-enable the OPTMSG feature once the restricted SQL statement completes.

- For debugging purposes

  You can disable the OPTMSG feature when you are trying to determine how each SQL statement responds.

- Before the last SQL statement in the program to ensure that the database server processes all messages before the application exits. If **OPTMSG** is enabled, the message is queued up for the database server but it is not sent for processing.

To avoid unintended chaining, reset the **OptMsg** global variable immediately after the SQL statement that requires it. The following code fragment enables message chaining for the DELETE statement:

```
OptMsg = 1;
EXEC SQL delete from customer;
OptMsg = 0;
EXEC SQL create index ix1 on customer (zipcode);
```

This example enables message chaining because the execution of the DELETE statement is not likely to fail. Therefore, it can be safely chained to the next SQL statement. Informix ESQL/C delays sending the message for the DELETE statement. The example disables message chaining after the DELETE statement so that Informix ESQL/C flushes all messages that have been queued up when the next SQL statement executes. By disabling the message chaining after the DELETE, the code fragment avoids unintended message chaining. When unintended chaining occurs, it can be difficult to determine which of the chained statements has failed.

At the CREATE INDEX statement, Informix ESQL/C sends both the DELETE and the CREATE INDEX statements to the database server.

Error Handling with Optimized Message Transfers

When the OPTMSG feature is enabled, your Informix ESQL/C application cannot perform error handling on any chained statement. If you are not sure whether a particular statement might generate an error, include error-handling code and do not enable message chaining for that statement.

Once an error occurs in a chained statement, the database server stops execution. Any SQL statements that follow the error are not executed. For example, the following code fragment intends to chain five INSERT statements (this fragment assumes that the **OPTMSG** environment variable is set to 1):

```
EXEC SQL create table tab1 (col1 INTEGER);

/* enable message chaining */
OptMsg = 1;
```

```
/* these two INSERT statements execute successfully */
EXEC SQL insert into tab1 values (1);
EXEC SQL insert into tab1 values (2);

/* this INSERT statement generates an error because the data
 * in the VALUES clause is not compatible with the column type */
EXEC SQL insert into tab1 values ('a');

/* these two INSERT statements never execute */
EXEC SQL insert into tab1 values (3);
EXEC SQL insert into tab1 values (4);

/* disable message chaining */
OptMsg = 0;

/* update one of the tab1 rows */
EXEC SQL update tab1 set col1 = 5 where col1 = 2;
if ( SQLCODE < 0 )
  .
  .
  .
```

In this code fragment, Informix ESQL/C flushes the message queue when it
reaches the UPDATE statement, sending the five INSERT statements and the
UPDATE statement to the database server for execution. Because the third INSERT
statement generates an error, the database server executes neither the remaining
INSERT statements nor the UPDATE statement. The UPDATE statement, which is
the last statement in the chained statements, returns the error from the failed
INSERT statement. The **tab1** table contains the rows with **col1** values of 1 and 2.

Using Database Server Control Functions

The following section describes the IBM Informix ESQL/C library functions that
you can use to control the database server sessions. The library functions are
located in Appendix B, "ESQL/C function library," on page B-1.

Function Name	Description	See
ifx_getcur_conn_name()	Returns the name of the current connection.	B-72
sqgetdbs()	Returns the names of databases that a database server can access.	B-217
sqlbreak()	Sends the database server a request to stop processing.	B-220
sqlbreakcallback()	Establishes a timeout interval and a callback function for interrupting an SQL request.	B-220
sqldetach()	Detaches a child process from a database server connection.	B-223
sqldone()	Determines whether the database server is currently processing an SQL request.	B-228
sqlexit()	Terminates a database server connection.	B-228
sqlsignal()	Performs signal handling and cleanup of child processes.	B-229
sqlstart()	Starts a database server connection.	B-231

The Timeout Program

The **timeout** program demonstrates how to set up a timeout interval. This program uses the **sqlbreakcallback()** function to perform the following actions:

- To specify a timeout interval of 200 milliseconds for execution of an SQL request
- To register the **on_timeout()** callback function to be called when an SQL request begins and ends as well as when the timeout interval elapses

If execution of an SQL request exceeds the timeout interval, the callback function uses the **sqldone()** function to ensure that the database server is still busy, prompts the user for confirmation of the interrupt, and then uses the **sqlbreak()** function to send an interrupt request to the database server.

Compiling the Program

Use the following command to compile the **timeout** program:

```
esql -o timeout timeout.ec
```

The **-o timeout** option causes the executable program to be named **timeout**. Without the **-o** option, the name of the executable program defaults to **a.out**. See "Using the esql Command" on page 2-4 for more information about the **esql** command.

Guide to the timeout.ec File

```
=======================================================================
 1. /*
 2.  * timeout.ec *
 3.  */
 4. #include <stdio.h>
 5. #include <string.h>
 6. #include <ctype.h>
 7. #include <decimal.h>
 8. #include <errno.h>
 9. EXEC SQL include sqltypes;
10. #define LCASE(c) (isupper(c) ? tolower(c) : (c))
11. /* Defines for callback mechanism */
12. #define DB_TIMEOUT     200   /* number of milliseconds in timeout */
13. #define SQL_INTERRUPT -213   /* SQLCODE value for interrupted stmt
    */
14. /* These constants are used for the canceltst table, created by
15.  * this program.
16.  */
17. #define MAX_ROWS      10000  /* number of rows added to table */
18. EXEC SQL define CHARFLDSIZE  20; /* size of character columns in
                                   * table */
19. /* Define for sqldone() return values */
20. #define SERVER_BUSY   -439
21. /* These constants used by the exp_chk2() function to determine
22.  * whether to display warnings.
23.  */
24. #define WARNNOTIFY        1
25. #define NOWARNNOTIFY      0
26. int4 dspquery();
27. extern int4 exp_chk2();
28. void on_timeout();
29. main()
```

```
30.  {
31.     char ques[80], prompt_ans();
32.     int4 ret;
33.     mint create_tbl(), drop_tbl();
34.     printf("TIMEOUT Sample ESQL Program running.\n\n");
35.     /*
36.      * Establish an explicit connection to the stores7 database
37.      * on the default database server.
38.      */
39.     EXEC SQL connect to 'stores7';
```

==

Lines 4 to 9

Lines 4 to 8 include the UNIX header files from the **/usr/include** directory. The
Informix ESQL/C **sqltypes.h** header file (line 9) defines names for integer values
that identify SQL and C data types.

Lines 10 to 20

Line 10 defines LCASE, a macro that converts an uppercase character to a
lowercase character. The DB_TIMEOUT (line 12) constant defines the number of
milliseconds in the timeout interval. The SQL_INTERRUPT constant (line 13)
defines the SQLCODE value that the database server returns when it interrupts an
SQL statement.

Lines 17 and 18 define constants that the **create_tbl()** function uses to create the
canceltst table. This table holds the test data needed for the large query (lines 125
to 132). MAX_ROWS is the number of rows that **create_tbl()** inserts into **canceltst**.
You can change this number if you find that the query does not run long enough
for you to interrupt it. CHARFLDSIZE is the number of characters in the character
fields (**char_fld1** and **char_fld2**) of **canceltst**.

Line 20 defines the SERVER_BUSY constant to hold the **sqldone()** return value that
indicates that the database server is busy processing an SQL request. Use of this
constant makes code more readable and removes the explicit return value from the
code.

Lines 24 and 25

The **exp_chk2()** exception-handling function uses the WARNNOTIFY and
NOWARNNOTIFY constants (lines 24 and 25). Calls to **exp_chk2()** specify one of
these as the second argument to indicate whether the function displays SQLSTATE
and SQLCODE information for warnings (WARNNOTIFY) or does not display this
information for warnings (NOWARNNOTIFY). For more information about the
exp_chk2() function, see "Lines 348 to 355" on page 12-43.

Lines 29 to 33

The **main()** program block begins on line 29. Lines 31 to 33 declare variables local
to the **main()** program block.

==
```
40.     if (exp_chk2("CONNECT to stores7", NOWARNNOTIFY) < 0)
41.         exit(1);
42.     printf("Connected to 'stores7' on default server\n");
43.     /*
44.      * Create the canceltst table to hold MAX_ROWS (10,000) rows.
45.      */
46.     if (!create_tbl())
47.         {
```

```
48.           printf("\nTIMEOUT Sample Program over.\n\n");
49.           exit(1);
50.           }
51.     while(1)
52.         {
53.         /*
54.          * Establish on_timeout() as callback function. The callback
55.          * function is called with an argument value of 2 when the
56.          * database server has executed a single SQL request for number
57.          * of milliseconds specified by the DB_TIMEOUT constant
58.          * (0.00333333 minutes by default). Call to  sqlbreakcallback()
59.          * must come after server connection is established and before
60.          * the first SQL statement that can be interrupted.
61.          */
62.         if (sqlbreakcallback(DB_TIMEOUT, on_timeout))
63.             {
64.             printf("\nUnable to establish callback function.\n");
65.             printf("TIMEOUT Sample Program over.\n\n");
66.             exit(1);
67.              }
68.         /*
69.          * Notify end user of timeout interval.
70.          */
71.         printf("Timeout interval for SQL requests is: ");
72.         printf("%0.8f minutes\n", DB_TIMEOUT/60000.00);
73.         stcopy("Are you ready to begin execution of the query?",
74.             ques);
75.         if (prompt_ans(ques) == 'n')
76.             {
77.             /*
78.              * Unregister callback function so table cleanup will not
79.              * be interrupted.
80.              */
81.             sqlbreakcallback(-1L, (void *)NULL);
82.             break;
83.             }
```
==

Lines 43 to 50

The **create_tbl()** function creates the **canceltst** table in the **stores7** database. It
inserts MAX_ROWS number of rows into this table. If **create_tbl()** encounters some
error while it creates **canceltst**, execution of the **timeout** program cannot continue.
The program exits with a status value of 1 (line 49).

Line 51

This **while** loop (which ends on line 97), controls the execution of the query on the
canceltst table. It allows the user to run this query multiple times to test various
interrupt scenarios.

Lines 53 to 67

The first task of the **while** loop is to use **sqlbreakcallback()** to specify a timeout
interval of DB_TIMEOUT (200) milliseconds and to register **on_timeout()** as the
callback function. If this call to **sqlbreakcallback()** fails, the program exits with a
status value of 1. To test different timeout intervals, you can change the
DB_TIMEOUT constant value and recompile the **timeout.ec** source file.

Lines 68 to 72

These **printf()** functions notify the user of the timeout interval. Notice that the message displays this interval in minutes, not milliseconds. It divides the DB_TIMEOUT value by 60,000 (number of milliseconds in a minute).

Lines 73 to 83

The **prompt_ans()** function asks the user to indicate when to begin execution of the **canceltst** query. If the user enters n (no), the program calls the **sqlbreakcallback()** function to unregister the callback function. This call prevents the SQL statements in the **drop_tbl()** function (lines 322 to 329) from initiating the callback function. For a description of the **prompt_ans()** function, see "Lines 337 to 347" on page 12-43.

```
=======================================================================
84.        /*
85.         * Start display of query output
86.         */
87.        printf("\nBeginning execution of query...\n\n");
88.        if ((ret = dspquery()) == 0)
89.           {
90.           if (prompt_ans("Try another run?") == 'y')
91.              continue;
92.           else
93.              break;
94.           }
95.        else  /* dspquery() encountered an error */
96.           exit(1);
97.        } /* end while */
98.     /*
99.      * Drop the table created for this program
100.     */
101.     drop_tbl();
102.     EXEC SQL disconnect current;
103.     if (exp_chk2("DISCONNECT for stores7", WARNNOTIFY) != 0)
104.        exit(1);
105.     printf("\nDisconnected stores7 connection\n");
106.     printf("\nTIMEOUT Sample Program over.\n\n");
107. }
108. /* This function performs the query on the canceltst table. */
109. int4 dspquery()
110. {
111.    mint cnt = 0;
112.    int4 ret = 0;
113.    int4 sqlcode = 0;
114.    int4 sqlerr_code, sqlstate_err();
115.    void disp_exception(), disp_error(), disp_warning();
116.    EXEC SQL BEGIN DECLARE SECTION;
117.       char fld1_val[ CHARFLDSIZE + 1 ];
118.       char fld2_val[ CHARFLDSIZE + 1 ];
119.       int4 int_val;
120.    EXEC SQL END DECLARE SECTION;
121.    /* This query contains an artificially complex WHERE clause to
122.     * keep the database server busy long enough for an interrupt
123.     * to occur.
124.     */
125.    EXEC SQL declare cancel_curs cursor for
126.       select sum(int_fld), char_fld1, char_fld2
```

```
127.        from canceltst
128.        where char_fld1 matches "*f*"
129.           or char_fld1 matches "*h*"
130.           or char_fld2 matches "*w*"
131.           or char_fld2 matches "*l*"
132.        group by char_fld1, char_fld2;
```

===

Lines 84 to 97

If the user chooses to continue the query, the program calls the **dspquery()**
function (line 88) to run the **canceltst** query. The **prompt_ans()** function displays a
prompt so the user can decide whether to run the program again.

Lines 98 to 101

The **drop_tbl()** function drops the **canceltst** table from the **stores7** database to clean
up after the program.

Lines 108 to 120

The **dspquery()** function runs a query of the **canceltst** table and displays the
results. It returns zero (success) or the negative value of SQLCODE (failure) to
indicate the result of the **canceltst** query.

Lines 121 to 132

Line 125 declares the **cancel_curs** cursor for the query. The actual SELECT (lines
126 to 132) obtains the sum of the **int_fld** column and the values of the two
character columns (**char_fld1** and **char_fld2**). The WHERE clause uses the
MATCHES operator to specify matching rows, as follows:

- All **char_fld1** columns that contain an f or an h with the criteria:

  ```
  char_fld1 matches "*f*"
  or char_fld1 matches "*h*"
  ```

  These criteria match rows with a **char_fld1** value of Informix or "4100 Bohannon
  Dr."

- All **char_fld2** columns that contain a w or an l with the criteria:

  ```
  char_fl2 matches "*w*"
  or char_fld2 matches "*l*"
  ```

  These criteria match rows with a **char_fld2** value of Software or "Menlo Park,
  CA".

This SELECT is artificially complex to ensure that the query takes a long time to
execute. Without a reasonably complex query, the database server finishes
execution before the user has a chance to interrupt it. In a production application,
only use the **sqlbreakcallback()** feature with queries that take a long time to
execute.

===
```
133.     EXEC SQL open cancel_curs;
134.     sqlcode = SQLCODE;
135.     sqlerr_code = sqlstate_err(); /* check SQLSTATE for exception */
136.     if (sqlerr_code != 0)          /* if exception found */
137.        {
138.        if (sqlerr_code == -1)      /* runtime error encountered */
139.           {
140.           if (sqlcode == SQL_INTERRUPT)  /* user interrupt */
141.              {
142.              /* This is where you would clean up resources */
143.              printf("\n      TIMEOUT INTERRUPT PROCESSED\n\n");
```

```
144.              sqlcode = 0;
145.                }
146.            else                        /* serious runtime error */
147.              disp_error("OPEN cancel_curs");
148.            EXEC SQL close cancel_curs;
149.            EXEC SQL free cancel_curs;
150.            return(sqlcode);
151.                }
152.          else if (sqlerr_code == 1)        /* warning encountered */
153.              disp_warning("OPEN cancel_curs");
154.            }
```

==

Line 133

This OPEN statement causes the database server to execute the SELECT that is
associated with the **cancel_curs** cursor. Because the database server executes the
canceltst query at this point, this OPEN is the statement that the user would be
most likely to interrupt. When the FETCH executes, the database server just sends
matching rows to the application, an operation that is not usually time intensive.

Lines 134 to 154

This block of code checks the success of the OPEN. Since the OPEN can be
interrupted, this exception checking must include an explicit check for the
interrupt value of **-213**. The database server sets SQLCODE to **-213** when it has
interrupted an SQL request. On line 140, the program uses the SQL_INTERRUPT
defined constant (which line 13 defines), for this SQLCODE value.

The **sqlstate_err()** function (line 135) uses the GET DIAGNOSTICS statement to
analyze the value of the SQLSTATE variable. If this function returns a non-zero
value, SQLSTATE indicates a warning, a runtime error, or the NOT FOUND
condition. Before the call to **sqlstate_err()**, line 134 saves the SQLCODE value so
that execution of any other SQL statements (such as GET DIAGNOSTICS in
sqlstate_err()) does not overwrite it. The function returns the value of SQLCODE if
the OPEN encounters a runtime error (line 150).

The first **if** statement (line 136) checks if the OPEN encounters any type of
exception (**sqlstate_err()** returns a nonzero value). The second **if** (line 138) checks if
the OPEN has generated a runtime error (return value of **-1**). However, if the
database server has interrupted the OPEN, **sqlstate_err()** also returns **-1**. Since
Informix ESQL/C does *not* handle an interrupted SQL statement as a runtime
error, the third **if** checks explicitly for the SQL_INTERRUPT value (line 140). If the
OPEN was interrupted, line 143 notifies the user that the interrupt request was
successful and then the function resets the saved SQLCODE value (in **sqlcode**) to
zero to indicate that the OPEN did *not* generate a runtime error.

Lines 146 and 147 execute *only* if the OPEN generates a runtime error other than
SQL_INTERRUPT (**-213**). The **disp_error()** function displays the exception
information in the diagnostics area and the SQLCODE value. Lines 148 to 150
clean up after the OPEN. They close and free the **cancel_curs** cursor and then
return the SQLCODE value. The **dspquery()** function does not continue with the
FETCH (line 158) if the OPEN was interrupted.

If **sqlstate_err()** returns one (1), the OPEN has generated a warning. Lines 152 and
153 call the **disp_warning()** function to display warning information from the
diagnostics area. For more informatio7n on the **disp_error()** and **disp_warning()**
functions, see Lines 341 to 348 in 12-43.

```
 =======================================================================
155.      printf("Displaying data...\n");
156.      while(1)
157.          {
158.          EXEC SQL fetch cancel_curs into :int_val, :fld1_val,
       :fld2_val;
159.          if ((ret = exp_chk2("FETCH from cancel_curs", NOWARNNOTIFY))
       == 0)
160.              {
161.             printf("   sum(int_fld) = %d\n", int_val);
162.             printf("   char_fld1 = %s\n", fld1_val);
163.             printf("   char_fld2 = %s\n\n", fld2_val);
164.              }
165.          /*
166.           * Will display warning messages (WARNNOTIFY) but continue
167.           * execution when they occur (exp_chk2() == 1)
168.           */
169.          else
170.              {
171.             if (ret==100)              /* NOT FOUND condition */
172.                 {
173.                 printf("\nNumber of rows found: %d\n\n", cnt);
174.                 break;
175.                 }
176.             if (ret < 0)              /* Runtime error */
177.                 {
178.                 EXEC SQL close cancel_curs;
179.                 EXEC SQL free cancel_curs;
180.                 return(ret);
181.                 }
182.              }
183.          cnt++;
184.          } /* end while */
185.      EXEC SQL close cancel_curs;
186.      EXEC SQL free cancel_curs;
187.      return(0);
188. }
189. /*
190.  *  The on_timeout() function is the callback function. If the user
191.  *  confirms the cancellation, this function uses sqlbreak() to
192.  *  send an interrupt request to the database server.
193.  */
194.  void on_timeout(when_called)
195.  mint when_called;
196.  {
197.  mint ret;
198.  static intr_sent;
 =======================================================================
```

Lines 155 to 182

This **while** loop executes for each row that the **cancel_curs** cursor contains. The
FETCH statement (line 158) retrieves one row from the **cancel_curs** cursor. If the
FETCH generates an error, the function releases the cursor resources and returns
the SQLCODE error value (lines 176 to 181). Otherwise, the function displays the
retrieved data to the user. On the last row (**ret** = 100), the function displays the
number of rows that it retrieved (line 173).

Lines 185 to 187

After the FETCH has retrieved the last row from the cursor, the function releases resources allocated to the **cancel_curs** cursor and returns a *success* value of zero.

Lines 190 to 199

The **on_timeout()** function is the callback function for the **timeout** program. The **sqlbreakcallback()** call on line 62 registers this callback function and establishes a timeout interval of 200 milliseconds. This function is called every time the database server begins and ends an SQL request. For long-running requests, the application also calls **on_timeout()** each time the timeout interval elapses.

```
======================================================================
199.    /* Determine when callback function has been called.  */
200.    switch(when_called)
201.      {
202.      case 0: /* Request to server completed */
203.         printf("+------SQL Request ends");
204.         printf("-----------------------------+\n\n");
205.         /*
206.          * Unregister callback function so no further SQL statements
207.          * can be interrupted.
208.          */
209.         if (intr_sent)
210.            sqlbreakcallback(-1L, (void *)NULL);
211.         break;
212.      case 1: /* Request to server begins */
213.         printf("+------SQL Request begins");
214.         printf("---------------------------+\n");
215.         printf("|                         ");
216.         printf("                          |\n");
217.         intr_sent = 0;
218.         break;
219.      case 2: /* Timeout interval has expired */
220.         /*
221.          * Is the database server still processing the request?
222.          */
223.         if (sqldone() == SERVER_BUSY)
224.            if (!intr_sent)   /* has interrupt already been sent? */
225.               {
226.               printf("|   An interrupt has been received ");
227.               printf("by the application.|\n");
228.               printf("|                          ");
229.               printf("                         |\n");
230.               /*
231.                * Ask user to confirm interrupt
232.                */
233.               if (cancel_request())
234.                  {
235.                  printf("|      TIMEOUT INTERRUPT ");
236.                  printf("REQUESTED                 |\n");
237.               /*
238.                * Call sqlbreak() to issue an interrupt request for
239.                * current SQL request to be cancelled.
240.                */
241.                  sqlbreak();
242.                  }
243.               intr_sent = 1;
```

```
244.                 }
245.         break;
```

===

Lines 199 to 249

This **switch** statement uses the callback function argument, **when_called**, to
determine the actions of the callback function, as follows:

- Lines 202 to 211: If **when_called** is 0, the callback function was called after the
  database server *ends* an SQL request. The function displays the bottom of the
  message-request box to indicate the end of the SQL request, as follows:

  ```
  +------SQL Request ends-------------------------------+
  ```

- Lines 212 to 218: If **when_called** is 1, the callback function was called when the
  database server *begins* an SQL request. The display of the top of the
  message-request box indicates this condition:

  ```
  +------SQL Request begins----------------------------+
  |                                                     |
  ```

  For more information about these message-request boxes, see "Lines 21 to 30"
  on page 12-44. The function also initializes the **intr_sent** flag to 0 because the
  user has not yet sent an interrupt for this SQL request.

- Lines 219 to 245: If **when_called** is 2, the callback function was called because
  the timeout interval has elapsed.

To handle the elapsed timeout interval, the callback function first calls the Informix
ESQL/C **sqldone()** function (line 223) to determine whether the database server is
still busy processing the SQL request. If the database server is idle, the application
does not need to send an interrupt. If **sqldone()** returns SERVER_BUSY (-439), the
database server is still busy.

Line 224 checks if the user has already attempted to interrupt the SQL request that
is currently executing. If an interrupt was sent, **intr_sent** is 1, and the program
does not need to send another request. If an interrupt request has not yet been
sent, the callback function notifies the user that the timeout interval has elapsed
(lines 226 to 229). It then uses the **cancel_request()** function (line 233) to allow the
user to confirm the interrupt. For more information about **cancel_request()**, see
"Lines 251 to 261" on page 12-40.

===

```
246.    default:
247.       printf("Invalid status value in callback: %d\n", when_called);
248.       break;
249.     }
250.     }
251.  /* This function prompts the user to confirm the sending of an
252.   * interrupt request for the current SQL request.
253.   */
254.  mint cancel_request()
255.  {
256.    char prompt_ans();
257.    if (prompt_ans("Do you want to confirm this interrupt?") == 'n')
258.       return(0);    /* don't interrupt SQL request */
259.    else
260.       return(1);    /* interrupt SQL request */
261.  }
262.  /* This function creates a new table in the current database. It
263.   * populates this table with MAX_ROWS rows of data. */
264.  mint create_tbl()
```

```
265.  {
266.    char st_msg[15];
267.    int ret = 1;
268.    EXEC SQL BEGIN DECLARE SECTION;
269.       mint cnt;
270.       mint pa;
271.       mint i;
272.       char fld1[ CHARFLDSIZE + 1 ], fld2[ CHARFLDSIZE + 1 ];
273.    EXEC SQL END DECLARE SECTION;
274.  /*
275.   * Create canceltst table in current database
276.   */
277.    EXEC SQL create table canceltst (char_fld1 char(20),
278.       char_fld2 char(20), int_fld integer);
279.    if (exp_chk2("CREATE TABLE", WARNNOTIFY) < 0)
280.       return(0);
281.    printf("Created table 'canceltst'\n");
282.    /*
283.     * Insert MAX_ROWS of data into canceltst
284.     */
285.    printf("Inserting rows into 'canceltst'...\n");
286.    for (i = 0; i < MAX_ROWS; i++)
287.       {
```

===

Lines 199 to 249 (continued)

If the user confirms the interrupt, the callback function calls the **sqlbreak()**
function to send the interrupt request to the database server. The callback function
does not wait for the database server to respond to the interrupt request. Execution
continues to line 243 and sets the **intr_sent** flag to 1, to indicate that the interrupt
request was sent. If the callback function was called with an invalid argument
value (a value other than 0, 1, or 2), the function displays an error message (line
247).

Lines 251 to 261

The **cancel_request()** function asks the user to confirm the interrupt request. It
displays the prompt:

Do you want to confirm this interrupt?

If the user answers y (yes), **cancel_request()** returns 0. If the user answers n (no),
cancel_request() returns 1.

Lines 262 to 281

The **create_tbl()** function creates the **canceltst** table and inserts the test data into
this table. The CREATE TABLE statement (lines 277 and 278) creates the **canceltst**
table with three columns: **int_fld**, **char_fld1**, and **char_fld2**. If the CREATE TABLE
encounters an error, the **exp_chk2()** function (line 279) displays the diagnostics-area
information and **create_tbl()** returns zero (0) to indicate that an error has occurred.

Lines 282 to 287

This **for** loop controls the insertion of the **canceltst** rows. The MAX_ROWS
constant determines the number of iterations for the loop, and hence the number
of rows that the function inserts into the table. If you cannot interrupt the **canceltst**
query (lines 126 to 132) because it executes too quickly, increase the value of
MAX_ROWS and recompile the **timeout.ec** file.

===

```
288.      if (i%2 == 1)  /* odd-numbered rows */
289.         {
290.         stcopy("4100 Bohannan Dr", fld1);
291.         stcopy("Menlo Park, CA", fld2);
292.         }
293.      else            /* even-numbered rows */
294.         {
295.         stcopy("Informix", fld1);
296.         stcopy("Software", fld2);
297.         }
298.      EXEC SQL insert into canceltst
299.         values (:fld1, :fld2, :i);
300.      if ( (i+1)%1000 == 0 ) /* every 1000 rows */
301.         printf("   Inserted %d rows\n", i+1);
302.      sprintf(st_msg, "INSERT #%d", i);
303.      if (exp_chk2(st_msg, WARNNOTIFY) < 0)
304.         {
305.         ret = 0;
306.         break;
307.         }
308.      }
309.   printf("Inserted %d rows into 'canceltst'.\n", MAX_ROWS);
310.  /*
311.   * Verify that MAX_ROWS rows have added to canceltst
312.   */
313.   printf("Counting number of rows in 'canceltst' table...\n");
314.   EXEC SQL select count(*) into :cnt from canceltst;
315.   if (exp_chk2("SELECT count(*)", WARNNOTIFY) < 0)
316.      return(0);
317.   printf("Number of rows = %d\n\n", cnt);
318.   return (ret);
319. }
320.  /* This function drops the 'canceltst' table */
321.  mint drop_tbl()
322.  {
323.   printf("\nCleaning up...\n");
324.   EXEC SQL drop table canceltst;
325.   if (exp_chk2("DROP TABLE", WARNNOTIFY) < 0)
326.      return(0);
327.   printf("Dropped table 'canceltst'\n");
328.   return(1);
329.  }
```

==

Lines 288 to 292

This **if** statement generates the values for the **char_fld1** and **char_fld2** columns of the **canceltst** table. Lines 290 and 291 execute for odd-numbered rows. They store the strings "4100 Bohannon Dr" and "Menlo Park, CA" in the **fld1** and **fld2** variables.

Lines 293 to 297

Lines 295 and 296 execute for even-numbered rows. They store the strings Informix and Software in the **fld1** and **fld2** variables.

Lines 298 to 307

The INSERT statement inserts a row into the **canceltst** table. It takes the value for the **int_fld** column from the **:i** host variable (the row number), and the values for the **char_fld1** and **char_fld2** columns from the **:fld1** and **:fld2** host variables. The function notifies the user after it inserts every 1000 rows (lines 300 and 301). If the INSERT encounters an error, the **exp_chk2()** function (line 303) displays the diagnostics-area information and **create_tbl()** returns zero to indicate that an error has occurred.

Lines 300 to 317

These lines verify that the program has added the rows to the **canceltst** table and that it can access them. The program does a SELECT on the newly created **canceltst** table and returns the number of rows found. The program checks whether this number matches the number that the function has added, which line 309 displays. If the SELECT encounters an error, the **exp_chk2()** function (line 315) displays the diagnostics-area information, and **create_tbl()** returns 0 to indicate that an error has occurred.

Lines 320 to 329

The **drop_tbl()** function drops the **canceltst** table from the current database. If the DROP TABLE statement (line 324) encounters an error, the **exp_chk2()** function displays the diagnostics-area information and **drop_tbl()** returns 0 to indicate that an error has occurred.

```
========================================================================
330.   /*
331.    * The inpfuncs.c file contains the following functions used in
       this
332.    * program:
333.    *    getans(ans, len) - accepts user input, up to 'len' number of
334.    *              characters and puts it in 'ans'
335.    */
336.   #include "inpfuncs.c"
337.   char prompt_ans(question)
338.   char * question;
339.   {
340.     char ans = ' ';
341.     while(ans != 'y' && ans != 'n')
342.        {
343.        printf("\n*** %s (y/n): ", question);
344.        getans(&ans,1);
345.        }
346.     return ans;
347.   }
348.   /*
349.    * The exp_chk() file contains the exception handling functions to
350.    * check the SQLSTATE status variable to see if an error has
        * occurred
351.    * following an SQL statement. If a warning or an error has
352.    * occurred, exp_chk2() executes the GET DIAGNOSTICS statement and
353.    * displays the detail for each exception that is returned.
354.    */
355.   EXEC SQL include exp_chk.ec;
========================================================================
```

Lines 330 to 336

Several of the Informix ESQL/C demonstration programs also call the **getans()** function. Therefore, this function is broken out into a separate C source file and included in the appropriate demonstration program. Because this function does not contain Informix ESQL/C, the program can use the C **#include** preprocessor statement to include the file. For a description of this function, see "Guide to the inpfuncs.c File" on page 7-42.

Lines 337 to 347

The **prompt_ans()** function displays the string in the **question** argument and waits for the user to enter y (yes) or n (no) as a response. It returns the single-character response.

Lines 348 to 355

The **timeout** program uses the **exp_chk2()**, **sqlstate_err()**, **disp_error()**, and **disp_warning()** functions to perform its exception handling. Because several demonstration programs use these functions, the **exp_chk2()** function and its supporting functions have been placed in a separate **exp_chk.ec** source file. The **timeout** program must include this file with the Informix ESQL/C **include** directive because the exception-handling functions use Informix ESQL/C statements. For a description of the **exp_chk.ec** file, see "Guide to the exp_chk.ec File" on page 11-32.

Tip: In a production environment, you would put functions such as **getans()**, **exp_chk2()**, **sqlstate_err()**, **disp_error()**, and **disp_warning()** into a library and include this library on the command line of the Informix ESQL/C-program compilation.

Example Output

This section includes a sample output of the **timeout** demonstration program. This program performs two runs of the **canceltst** query, as follows:

- Lines 20 to 43: The first run confirms the interrupt request as soon as the confirmation prompt appears. (The user enters y.)
- Lines 44 to 75: The second run does *not* confirm the interrupt request. (The user enters n.)

The numbers that appear in the following output are for explanation only. They do *not* appear in the actual program output.

```
========================================================================
  1.   TIMEOUT Sample ESQL Program running.
  2.   Connected to 'stores7' on default server
  3.   Created table 'canceltst'
  4.   Inserting rows into 'canceltst'...
  5.     Inserted 1000 rows
  6.     Inserted 2000 rows
  7.     Inserted 3000 rows
  8.     Inserted 4000 rows
  9.     Inserted 5000 rows
 10.     Inserted 6000 rows
 11.     Inserted 7000 rows
 12.     Inserted 8000 rows
 13.     Inserted 9000 rows
 14.     Inserted 10000 rows
 15.   Inserted 10000 rows into 'canceltst'.
 16.   Counting number of rows in 'canceltst' table...
```

```
17.   Number of rows = 10000
18.   Timeout interval for SQL requests is: 0.00333333 minutes
19.   *** Are you ready to begin execution of the query? (y/n): y
20.   Beginning execution of query...
21.   +------SQL Request begins----------------------------+
22.   |                                                    |
23.   +------SQL Request ends------------------------------+
24.   +------SQL Request begins----------------------------+
25.   |                                                    |
26.   |    An interrupt has been received by the application.|
27.   |                                                    |
28.   *** Do you want to confirm this interrupt? (y/n): y
29.   |       TIMEOUT INTERRUPT REQUESTED                  |
30.   +------SQL Request ends------------------------------+

========================================================================
```

Lines 3 to 17

The **create_tbl()** function generates these lines. They indicate that the function has successfully created the **canceltst** table, inserted the MAX_ROWS number of rows (1,000), and confirmed that a SELECT statement can access these rows. For a description of the **create_tbl()** function, see the annotation beginning with "Lines 262 to 281" on page 12-40.

Lines 18 to 19

Line 18 displays the timeout interval to indicate that **sqlbreakcallback()** has successfully registered the callback function and established the timeout interval of 200 milliseconds (0.00333333 minutes). Line 19 asks the user to indicate the beginning of the query execution. This prompt prepares the user for the confirmation prompt (lines 28 and 43), which must be answered quickly to send an interrupt while the database server is still executing the query.

Line 20

This line indicates the beginning of the **dspquery()** function, the point at which the database server begins the **canceltst** query.

Lines 21 to 30

The program output uses a *message-request box* to indicate client-server communication:

```
+------SQL Request begins----------------------------+
|                                                    |
+------SQL Request ends------------------------------+
```

Each box represents a single message request sent between the client and the server. The callback function displays the text for a message-request box. (For a description of which parts of the function display the text, see "Lines 199 to 249" on page 12-39.) To execute the OPEN statement, the client and server exchanged two message requests, which the two message-request boxes in the output indicate. For more information about message requests, see "Interruptible SQL Statements" on page 12-22.

The first message-request box (lines 21 to 23) indicates that the first message request completes before the timeout interval elapses. The second message-request box (lines 29 to 30) indicates that execution of this message request exceeds the timeout interval and calls the callback function with a status value of 2. The callback function prompts the user to confirm the interrupt request (line 28).

Line 29 indicates that the **sqlbreak()** function has requested an interrupt. The message request then completes (line 30).

```
=======================================================================
31.    TIMEOUT INTERRUPT PROCESSED
32.    *** Try another run? (y/n): y
33.    Timeout interval for SQL requests is: 0.00333333 minutes
34.    *** Are you ready to begin execution of the query? (y/n): y
35.    Beginning execution of query...
36.    +------SQL Request begins-----------------------------+
37.    |                                                     |
38.    +------SQL Request ends-------------------------------+
39.    +------SQL Request begins-----------------------------+
40.    |                                                     |
41.    |    An interrupt has been received by the application.|
42.    |                                                     |
43.    *** Do you want to confirm this interrupt? (y/n): n
44.    +------SQL Request ends-------------------------------+
45.    Displaying data...
46.      sum(int_fld) = 25000000
47.      char_fld1 = 4100 Bohannan Dr
48.      char_fld2 = Menlo Park, CA
49.      sum(int_fld) = 24995000
50.      char_fld1 = Informix
51.      char_fld2 = Software
52.    Number of rows found: 2
53.    *** Try another run? (y/n): n
54.    Cleaning up...
55.    Dropped table 'canceltst'
56.    Disconnected stores7 connection
57.    TIMEOUT Sample Program over.

=======================================================================
```

Line 31

When the database server actually processes the interrupt request, it sets SQLCODE to -213. Line 31 indicates that the application program has responded to this status.

Line 32

This prompt indicates the end of the first run of the **canceltst** query. The user responds y to the prompt to run the query a second time.

Lines 36 to 41

The message-request box indicates that the first message request completes before the timeout interval elapses. The second message-request box (lines 39 to 44) indicates that execution of this message request again exceeds the timeout interval and calls the callback function (with **when_called** = 2). The callback function prompts the user to confirm the interrupt request (line 43). This time the user answers n.

Lines 45 to 52

Because the user has not interrupted the **canceltst** query, the program displays the row information that the query returns.

Lines 54 and 55

The **drop_tbl()** function generates these lines. They indicate that the function has successfully dropped the **canceltst** table from the database. For a description of the **drop_tbl()** function, see the annotation beginning with "Lines 320 to 329" on page 12-42.

Using ESQL/C Connection Library Functions in a Windows Environment

To establish an *explicit* connection (sometimes called a *direct connection*), IBM Informix ESQL/C supports the SQL connection statements. For a complete description of the SQL connection statements, see the *IBM Informix Guide to SQL: Syntax*. Informix ESQL/C also supports the connection library functions that Table 12-4 lists for establishing an explicit connection from a Windows environment.

Table 12-4. ESQL/C Connection Library Functions and Their SQL Equivalents

ESQL/C for Windows Library Function	Description	SQL Equivalent	See
GetConnect()	Requests an explicit connection and returns a pointer to the connection information	CONNECT TO '@dbservername' WITH CONCURRENT TRANSACTION	B-60
SetConnect()	Switches the connection to an established (dormant) explicit connection	SET CONNECT TO (*without* the DEFAULT option)	B-216
ReleaseConnect()	Closes an established explicit connection	DISCONNECT (*without* the DEFAULT, CURRENT, or ALL options)	B-177

Important: Informix ESQL/C supports the connection library functions for compatibility with Version 5.01 Informix ESQL/C for Windows applications. When you write new Informix ESQL/C applications for Windows environments, use the SQL connection statements (CONNECT, DISCONNECT, and SET CONNECTION) instead of the Informix ESQL/C connection library functions.

Informix ESQL/C uses an internal structure that contains the handle for the connection and other connection information. The Informix ESQL/C connection library functions use the connection handle, together with the information in the internal structure, to pass connection information to and from the application. The application can use the connection handle to identify an explicit connection.

If you use these connection functions to establish explicit connections, keep the following restrictions in mind:

- If you open a cursor in one module (such as a shared DLL), and then use an explicit connection to use that cursor in another module, you must use a host variable for the name of the cursor when you declare the cursor.
- Make sure that your application uses the correct connection handle at all times.

Warning: If an application uses the wrong connection handle, the application can modify the wrong database without the knowledge of the user.

When you compile your Informix ESQL/C program, the **esql** command processor automatically links the Informix ESQL/C connection functions to your program. The library functions are located in Appendix B, "ESQL/C function library," on page B-1.

Chapter 13. Using Informix libraries

These topics describe how to link the static, shared, and thread-safe Informix general libraries with your Informix ESQL/C application.

IBM Informix products use the Informix general libraries for interactions between the client SQL application programming interface (API) products (IBM Informix ESQL/C and IBM Informix ESQL/COBOL) and the database server. You can choose between the following types of Informix general libraries to link with your Informix ESQL/C application:

- Static Informix general libraries

  To link a static library, the linker copies the library functions to the executable file of your Informix ESQL/C program. The static Informix general libraries allow an Informix ESQL/C program on computers that do not support shared memory to access the Informix general library functions.

- Shared Informix general libraries

  To link a shared library, the linker copies information about the location of the library to the executable file of your Informix ESQL/C program. The shared Informix libraries allow several applications to share a single copy of these libraries, which the operating system loads, just once, into shared memory.

- Thread-safe versions of static and shared Informix general libraries

  The thread-safe versions of Informix general libraries allow an Informix ESQL/C application that has several threads to call these library functions simultaneously. The thread-safe versions of Informix libraries are available as both static libraries and shared libraries.

Beginning with CSDK version 3.0, static versions of Informix general libraries are available on Windows and UNIX operating systems. The following table shows the available options.

Table 13-1. Different version of the ESQL/C general library available for UNIX and Windows

Linking options	Thread-safe options	
	Thread-safe	Default
Static	Static, thread-safe general libraries	Static, default general libraries
Shared	Shared, thread-safe general libraries	Shared, default general libraries.

Choosing a version of the Informix general libraries

This section provides information about the following topics:

- What are the IBM Informix general libraries?
- What command-line options of the **esql** command determine the version of the Informix general libraries to link with your Informix ESQL/C program?
- How do you link the static Informix general libraries that are available on UNIX and Windows operating systems with your Informix ESQL/C program?
- How do you link the shared Informix general libraries with your Informix ESQL/C program?
- What are some factors that you need to consider to determine which type of Informix general libraries to use?

The Informix general libraries

The following is a list of the IBM Informix general libraries for Informix ESQL/C on a UNIX operating system.

libgen	Contains functions for general tasks
libos	Contains functions for tasks that are required from the operating system
libsql	Contains functions that send SQL statements between client application and database server
libgls	Contains functions that provide Global Language Support (GLS) to IBM Informix products
libasf	Contains functions that handle communication protocols between client application and database server

Informix general libraries reside in the **$INFORMIXDIR/lib/esql** and **$INFORMIXDIR/lib** directories on UNIX operating systems.

The Informix general library for Informix ESQL/C for Windows is just one DLL named **isqlt09a.dll**. The file is in the **%INFORMIXDIR%\lib** directory.

The static library for Informix ESQL/C for Windows is named **isqlt09s.lib**. The file is in the **$INFORMIXDIR/lib** directory.

On many platforms there is a system library named **libgen.a**. To avoid compilation errors, it is recommended that you do not use the **libgen.a** Informix library. Instead, use **libifgen.a** Informix library which contains a symbolic link to **libgen.a**.

The esql command

To determine which type of IBM Informix general libraries to link with your Informix ESQL/C application, the **esql** command supports the command-line options in the following table.

Table 13-2. The **esql** *command-line options for Informix general libraries*

Version of Informix libraries to link	esql command- line option	Reference and comments
Shared libraries	*No option (default)*	"Link shared Informix general libraries" on page 13-4
Static libraries	**-static**	"Link static Informix general libraries" on page 13-3.
Thread-safe shared libraries	**-thread**	"Linking Thread-Safe Informix General Libraries to an ESQL/C Module on a UNIX Operating System" on page 13-16 and "Linking Thread-Safe Informix General Libraries to an ESQL/C Module in a Windows Environment" on page 13-17
Thread-safe static libraries	**-thread -static**	"Creating a Dynamic Thread Library on UNIX Operating Systems" on page 13-24

For the complete syntax of the **esql** command, see "Using the esql Command" on page 2-4.

Link static Informix general libraries

Beginning with CSDK version 3.0, static versions of IBM Informix general libraries are available on Windows as well as UNIX operating systems.

The static Informix general libraries retain their pre-Version 7.2 names. Static-library names have the following formats:

• A non-thread-safe static Informix general library has a name of the form **lib*xxx*.a**.

• A thread-safe static Informix general library has a name of the form **libth*xxx*.a**.

In these static-library names, *xxx* identifies the particular static Informix general library. With Version 7.2 and later, the static and thread-safe static Informix general libraries use names of this format as their *actual* names. The following sample output shows the actual names for the **libos** static (**libos.a**) and thread-safe static (**libthos.a**) libraries:

```
% cd $INFORMIXDIR/lib/esql
% ls -l lib*os.a
-rw-r--r--  1 informix   145424 Nov 8 01:40 libos.a
-rw-r--r--  1 informix   168422 Nov 8 01:40 libthos.a
```

The **esql** command links the code that is associated with the actual names of the
static Informix general libraries into the Informix ESQL/C application. At runtime,
your Informix ESQL/C program can access these Informix general-library
functions directly from its executable file.

Link static Informix general libraries into an ESQL/C module

To link static IBM Informix general libraries with an Informix ESQL/C module,
compile your program with the **-static** command-line option. The following
command links the static non-thread-safe Informix libraries with the **file.exe**
executable file:

```
esql -static file.ec -o file.exe
```

The **esql** command can also link the code for thread-safe shared Informix general
libraries with an Informix ESQL/C application. For more information, see "Linking
Thread-Safe Libraries" on page 13-16.

Tip: The **esql** command for pre-Version 7.2 Informix ESQL/C products linked
static versions of the Informix general libraries. Because the **esql** command
links shared versions of these libraries by default, you must specify the **-static**
option to link the static versions with your Informix ESQL/C application.

Link shared Informix general libraries

Informix ESQL/C can dynamically link a shared library, which places this library
in shared memory. Once the shared library is in shared memory, other Informix
ESQL/C applications can also use it. Shared libraries are most useful in multiuser
environments where only one copy of the library is required for all applications.

Important: To use shared libraries in your Informix ESQL/C application, your
operating system must support shared libraries. Operating systems that
support shared libraries include Sun and HP versions of UNIX and
Windows. You should be familiar with the creation of shared libraries
and with the compile options that your C compiler requires to build
them.

Symbolic names of linked shared libraries (UNIX)

When the **esql** command links shared or thread-safe shared IBM Informix libraries
with your Informix ESQL/C application, it uses the *symbolic* names of these
libraries. The symbolic names of the Informix shared libraries have the following
formats:

- A non-thread-safe shared Informix general library has a symbolic name of the
  form **lib*xxx.yyy***.
- A thread-safe shared Informix general library has a symbolic name of the form
  libth*xxx.yyy*.

In these static-library names, *xxx* identifies the particular library and *yyy* is a
platform-specific file extension that identifies shared library files.

Tip: To refer to a specific shared-library file, this publication often uses the file
extension for the Sun UNIX operating system, the **.so** file extension. For the
shared-library file extension that your UNIX operating system uses, see your
UNIX operating system documentation.

When you install the Informix ESQL/C product, the installation script makes a symbolic link of the *actual* shared product library name to the file with the symbolic name. Figure 13-1 shows the format for the actual names of shared and thread-safe shared versions of Informix libraries.
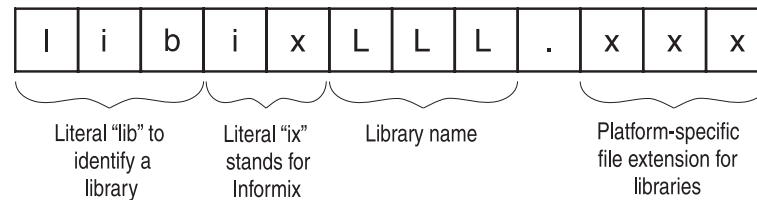
Figure 13-1. Format of an Informix Shared-Library Name

The following sample output shows the symbolic and actual names for the **libos.a** static library and the **libos.so** shared library (on a Sun platform):

```
%ls -l $INFORMIXDIR/esql/libos*
-rw-r--r--  1 informix   145424 Nov 8 01:40 libos.a
lrwxrwxrwx  1 root           11 Nov 8 01:40 libos.so -> iosls07a.so*
```

The **esql** command links the symbolic shared-library names with the Informix ESQL/C application. At runtime, Informix ESQL/C dynamically links the code for the shared Informix general library when the program requires an Informix general-library function.

To link shared Informix general libraries to an ESQL/C module:

1. Set the environment variable that specifies the library search path at runtime so that it includes the **$INFORMIXDIR/lib** and **$INFORMIXDIR/lib/esql** paths on a UNIX operating system; and **%INFORMIXDIR%lib** in a Windows environment.

   On many UNIX operating systems, the **LD_LIBRARY_PATH** environment variable specifies the library search path. The following command sets **LD_LIBRARY_PATH** in a C shell:

   ```
   setenv LD_LIBRARY_PATH $INFORMIXDIR/lib:$INFORMIXDIR/
   lib/esql:/usr/lib
   ```

   In Windows environments, use the following command:

   ```
   set LIB = %INFORMIXDIR%\lib\;%LIB%
   ```

2. Compile your program with the **esql** command.

   To link shared Informix general libraries with an Informix ESQL/C module, you do not need to specify a command-line option. Informix ESQL/C links shared libraries by default. The following command compiles the **file.ec** source file with shared Informix libraries:

   ```
   esql file.ec -o file.exe
   ```

The **esql** command also uses the symbolic name when it links the thread-safe shared Informix general libraries with an Informix ESQL/C application. For more information, see "Linking Thread-Safe Libraries" on page 13-16.

Choosing Between Shared and Static Library Versions

Beginning with CSDK version 3.0, static versions of IBM Informix general libraries are available on Windows as well as UNIX operating systems.

Pre-Version 7.2 Informix ESQL/C products use static versions of the libraries for the Informix general libraries. While static libraries are effective in an environment

that does not require multitasking, they become inefficient when more than one application calls the same functions. Version 7.2 and later of Informix ESQL/C also supports shared versions of the Informix general libraries.

Shared libraries are most useful in multiuser environments where only one copy of the library is required for all applications. Shared libraries bring the following benefits to your Informix ESQL/C application:

- Shared libraries reduce the sizes of executable files because these library functions are linked dynamically, on an as-needed basis.
- At runtime, a single copy of a shared library can be linked to several programs, which results in less memory use.
- The effects of shared libraries in an Informix ESQL/C executable are transparent to the end user.

Although shared libraries save both disk and memory space, when an Informix ESQL/C application uses them it must perform the following overhead tasks:

- Dynamically load the shared library into memory for the first time
- Perform link-editing operations
- Execute library position-independent code

These overhead tasks can incur runtime penalties and are not necessary when you use static libraries. However, these costs can be offset by the savings in input/output (I/O) access time once the operating system has already loaded and mapped an Informix shared library.

Important: You might experience a one-time negative effect on the performance of the client side of the application when you load the shared libraries the first time the application is executed. For more information, consult your operating system documentation.

Because the real I/O time that the operating system needs to load a program and its libraries usually does not exceed the I/O time saved, the apparent performance of a program that uses shared libraries is as good as or better than one that uses static libraries. However, if applications do not share, or if your processor is saturated when your applications call shared-library routines, you might not realize these savings.

You can also link thread-safe versions of the static and shared Informix general libraries with an Informix ESQL/C application. For more information, see "Creating a Dynamic Thread Library on UNIX Operating Systems" on page 13-24.

Compatibility of Preexisting ESQL/C Applications with Current Library Versions

You specify the **esql** command-line options (in Table 13-2 on page 13-3) to tell the **esql** command which version of the IBM Informix libraries to link with the Informix ESQL/C application. Once the **esql** command successfully compiles and links your application, the version of the Informix general libraries is fixed. When you install a new version of Informix ESQL/C, you receive new copies of the Informix general libraries. Whether you need to recompile and relink your existing Informix ESQL/C applications to run with these new copies depends on the factors that the following table describes.

Change to the Informix General LIbrary	Version of the Informix General Library	Need to Recompile or Relink?
New release of the Informix general libraries	Static	Only if the application needs to take advantage of a new feature in the new release
	Thread-safe static	
Informix general libraries in new release have a new major-version number	Shared	Only if the application needs to take advantage of a new feature in the new release
	Thread-safe shared	
Informix general libraries in new release have a new API-version number	Shared	Must recompile and relink
	Thread-safe shared	

On UNIX, you can use the **ifx_getversion** utility to determine the version of an Informix library that is installed on your system. For more information, see "Using the ifx_getversion Utility (UNIX)" on page 13-7.

In Windows environments, use the following **find** command to find the occurrence of the string that contains the version number in the **isqlt09A.dll**. Note that the command needs to be issued from the **%INFORMIXDIR%\bin** directory.

```
C: cd %INFORMIXDIR%\bin
C: find "INFORMIX-SQL" isqlt09a.dll
```

The output of the **find** command is shown below:

```
- - - - - - - - - - ISQLT09A.DLL
INFORMIX-SQL Version 9.20T1N79
```

Using the ifx_getversion Utility (UNIX)

To obtain the complete version name of an IBM Informix library, use the **ifx_getversion** utility. Before you run **ifx_getversion**, set the **INFORMIXDIR** environment variable to the directory in which your IBM Informix product is installed.

The **ifx_getversion** utility has the following syntax.

```
►►──ifx_getversion──┬─libgen.xx──┬──────────────────────►◄
                    ├─libthgen.x─┤
                    ├─libos.xx───┤
                    ├─libthos.x──┤
                    ├─libsql.xx──┤
                    ├─libthsql.x─┤
                    ├─libgls.xx──┤
                    └─libasf.xx──┘
```
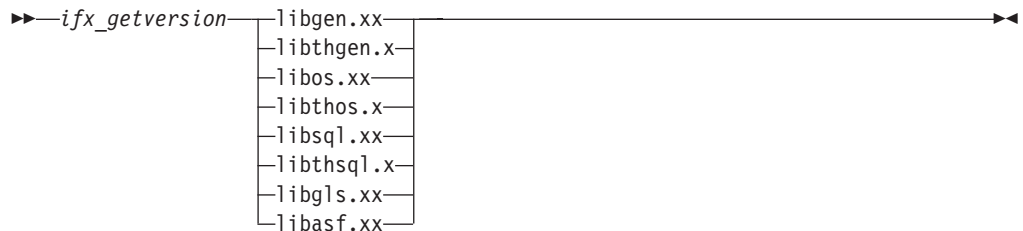
Element	Purpose	Key Considerations
xx	For static libraries, xx specifies the **.a** file extension; for shared libraries, *xx* specifies the platform-specific file extension.	**Additional Information**: For shared libraries, the Sun platform uses the **.so** file extension and the Hewlett-Packard (HP) platform uses the **.sl** file extension.

The following example shows an example of output that the **ifx_getversion** utility generates for the **libgen** Informix library:

```
IBM/Informix-Client SDK Version 3.00.UN191
IBM/Informix LIBGEN LIBRARY Version 3.00.UN191
Copyright (C) 1991-2007 IBM
```

Output of **ifx_getversion** depends on the version of Informix ESQL/C software that is installed on your system.

Checking the API Version of a Library

When you invoke an Informix ESQL/C application that is linked with shared IBM Informix general libraries, the release number of these shared libraries must be compatible with that of the shared libraries in the **$INFORMIXDIR/lib** or the **%INFORMIXDIR%\lib** directory.

In a Windows environment, a developer can easily verify the name of the shared library DLL, namely **isqlt*nnx*.dll**, where *nn* stand for the version number, and *x* stand for the API version.

In the case of an Informix ESQL/C application on UNIX, however, given that the linked libraries get symbolic names, it is not easy to find the version number of the linked library. Therefore, Informix ESQL/C does this check for you. Informix ESQL/C performs an internal check between the API version of the library that the application uses and the API version of the library that is installed as part of your Informix ESQL/C product. Figure 13-1 on page 13-5 shows where the API version appears in the shared library name.

Informix ESQL/C uses an IBM Informix function that is called **checkapi()** to perform this check. The **checkapi()** function is in the **checkapi.o** object file, which is contained in the **$INFORMIXDIR/lib/esql** directory. The **esql** command automatically links this **checkapi.o** object file with every executable that it creates.

To determine the API version of the library that the application uses, Informix ESQL/C checks the values of special macro definitions in the executable file. When the Informix ESQL/C preprocessor processes a source file, it copies the macro definitions from the **sqlhdr.h** header file into the C source file (**.c**) that it generates. The following example shows sample values for these macros:

```
#define CLIENT_GEN_VER            710
#define CLIENT_OS_VER             710
#define CLIENT_SQLI_VER            710
#define CLIENT_GLS_VER            710
```

Tip: The Informix ESQL/C preprocessor automatically includes the **sqlhdr.h** file in all Informix ESQL/C executable files that it generates.

If the API version of the libraries in this executable file are not compatible, Informix ESQL/C returns a runtime error that indicates which library is not compatible. You must recompile your Informix ESQL/C application to link the new release version of the shared library.

If you do not use **esql** to link one of the shared Informix general libraries with your Informix ESQL/C application, you must explicitly link the **checkapi.o** file with your application. Otherwise, Informix ESQL/C might generate an error at link time of the form:

```
undefined ifx_checkAPI()
```

Create thread-safe ESQL/C applications

Informix ESQL/C provides shared and static thread-safe as well as shared and static default versions of the Informix general libraries on both UNIX and Windows operating systems. On Windows operating systems, Informix ESQL/C provides a dll named **isqlt09a.dll** and a static thread-safe library named **isqlt09s.lib**.

A thread-safe Informix ESQL/C application can have one active connection per thread and many threads per application. The thread-safe libraries contain thread-safe (or reentrant) functions. A thread-safe function is one that behaves correctly when several threads call it simultaneously.

In the case of Informix ESQL/C on a UNIX operating system, the thread-safe Informix general libraries use functions from the Distributed Computing Environment (DCE) thread package. The DCE thread library, which the Open Software Foundation (OSF) developed, creates a standard interface for thread-safe applications.

If the DCE thread library is not available on your operating system, ESQL/C can use POSIX thread libraries or Sun Solaris thread libraries.

If your operating system supports the DCE, POSIX, or Solaris thread packages, you must install them on the same client computer as ESQL/C.

In Windows environments, the Informix general libraries use the Windows API to ensure that they are thread safe.

With the thread-safe Informix general libraries, you can develop thread-safe Informix ESQL/C applications. A thread-safe application can have many threads of control. It separates a process into multiple execution threads, each of which runs independently. While a non-threaded Informix ESQL/C application can establish many connections to one or more databases, it can have only one active connection at a time. An active connection is one that is ready to process SQL requests. A thread-safe Informix ESQL/C application can have one active connection per thread and many threads per application.

When you specify the **-thread** command-line option, the **esql** command passes this option to the Informix ESQL/C preprocessor, **esqlc**. With the **-thread** option, the Informix ESQL/C preprocessor generates thread-safe code that different threads can execute concurrently.

Characteristics of thread-safe ESQL/C code

The thread-safe Informix ESQL/C code has the following characteristics that are different from non-thread-safe code:

- The thread-safe code does not define any static data structures.

  For example, Informix ESQL/C allocates **sqlda** structures dynamically and binds host variables to these **sqlda** structures at runtime. For more information about **sqlda** structures to perform dynamic SQL, see Chapter 12, "Working with the database server," on page 12-1.

- The thread-safe code declares cursor blocks dynamically instead of declaring them statically.

- The thread-safe code uses macro definitions for status variables (SQLCODE, SQLSTATE and the **sqlca** structure).

For more information, see "Defining Thread-Safe Variables (UNIX)" on page 13-17.

Because of the preceding differences, the thread-safe C source file (**.c**) that the Informix ESQL/C preprocessor generates is different from the non-threaded C source file. Therefore, you cannot link Informix ESQL/C applications that have been compiled with the **-thread** option with applications that have not already been compiled with **-thread**. To link such applications, you must compile both applications with the **-thread** option.

Programming a Thread-Safe ESQL/C Application

This section provides useful hints for how to create thread-safe Informix ESQL/C applications. It discusses the following programming techniques for a thread-safe environment:

- Establishing concurrent active connections
- Using connections across threads
- Disconnecting all connections
- Using prepared statements across threads
- Using cursors across threads
- Accessing environment variables
- Handling **decimal** values
- Handling DCE restrictions (UNIX)

Concurrent active connections

In a thread-safe Informix ESQL/C application, a database server connection can be in one of the following states:

- An *active* database server connection is ready to process SQL requests.

  The major advantage of a thread-safe Informix ESQL/C application is that each thread can have one active connection to a database server. Use the CONNECT statement to establish a connection and make it active. Use the SET CONNECTION statement (with no DORMANT clause) to make a dormant connection active.

- A *dormant* database server connection was established but is not currently associated with a thread.

  When a thread makes an active connection dormant, that connection becomes available to other threads. Conversely, when a thread makes a dormant connection active, that connection becomes unavailable to other threads. Use the SET CONNECTION...DORMANT statement to explicitly put a connection in a dormant state.

The *current connection* is the active database server connection that is currently sending SQL requests to, and possibly receiving data from, the database server. A single-threaded application has only one current (or active) connection at a time. In a multithreaded application, each thread can have a current connection. Thus a multithreaded application can have multiple active connections simultaneously.

When you switch connections with the SET CONNECTION statement (with no DORMANT clause), SET CONNECTION implicitly puts the current connection in the dormant state. Once in a dormant state, a connection is available to other threads. Any thread can access any dormant connection. However, a thread can only have one active connection at a time.

The following figure shows a thread-safe Informix ESQL/C application that establishes three concurrent connections, each of which is active.
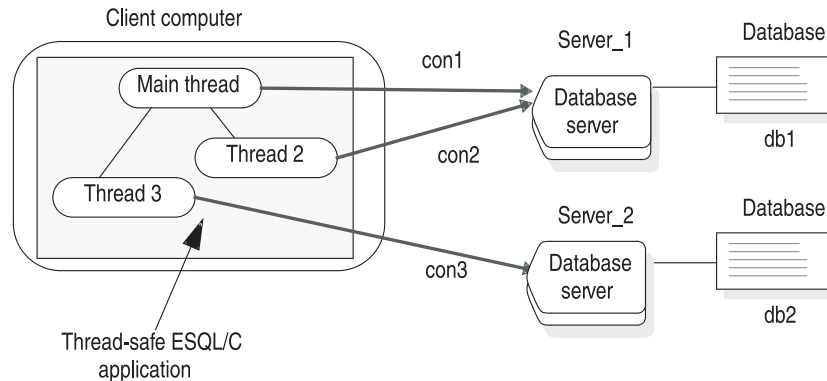
Figure 13-2. Concurrent connections in a thread-safe ESQL/C application

In previous figure, the Informix ESQL/C application consists of the following threads:

- The main thread (main function) starts connection **con1** to database **db1** on **Server_1**.
- The main thread spawns Thread 2. Thread 2 establishes connection **con2** to database **db1** on **Server_1**.
- The main thread spawns Thread 3. Thread 3 establishes connection **con3** to database **db2** on **Server_2**.

All connections in Figure 13-2 are concurrently active and can execute SQL statements. The following code fragment establishes the connections that Figure 13-2 illustrates. It does not show DCE-related calls and code for the **start_threads()** function.

```
main()
{
    EXEC SQL connect to 'db1@Server_1' as 'con1';
    start_threads(); /* start 2 threads */
    EXEC SQL select a into :a from t1; /* table t1 resides in db1 */
    .
    .
    .
}
thread_1()
{
    EXEC SQL connect to 'db1@Server_1' as 'con2';
    EXEC SQL insert into table t2 values (10); /* table t2 is in db1 */
    EXEC SQL disconnect 'con2';
}
thread_2()
{
    EXEC SQL connect to 'db2@Server_2' as 'con3';
    EXEC SQL insert into table t1 values(10); /* table t1 resides in db2
                                                */
    EXEC SQL disconnect 'con3';
}
```

You can use the **ifx_getcur_conn_name()** function to obtain the name of the current connection. For more information, see "Identifying an Explicit Connection" on page 12-20.

Connections across threads

If your application contains threads that need to use the same connection, one thread might be using the connection when another thread needs to access it. To avoid this type of contention, your Informix ESQL/C application must manage access to the connections.

The simplest way to manage a connection that several threads must use is to put the SET CONNECTION statement in a loop. (For more information about the SET CONNECTION statement, see the *IBM Informix Guide to SQL: Syntax*.) The code fragment in the following figure shows a simple SET CONNECTION loop.

```
/* wait for connection: error -1802 indicates that the connection
   is in use
 */
do {
   EXEC SQL set connection  :con_name;
   } while (SQLCODE == -1802);
```

Figure 13-3. A SET CONNECTION loop to handle multithread access to a connection

The preceding algorithm waits for the connection that the host variable **:con_name** names to become available. However, the disadvantage of this method is that it consumes processor cycles.

The following code fragment uses the CONNECT statement to establish connections and SET CONNECTION statements to make dormant connections active within threads. It also uses SET CONNECTION...DORMANT to make active connections dormant. This code fragment establishes the connections that Figure 13-2 on page 13-11 illustrates. It does not show DCE-related calls and code for the **start_threads()** function.

```
main()
{   EXEC SQL BEGIN DECLARE SECTION;
       int a;
    EXEC SQL END DECLARE SECTION;

    start_threads(); /* start 2 threads */
    wait for the threads to finish work.

    /* Use con1 to update table t1; Con1 is dormant at this point.*/
    EXEC SQL set connection 'con1';
    EXEC SQL update table t1 set a = 40 where a = 10;

    /* disconnect all connections */
    EXEC SQL disconnect all;
}
thread_1()
{
    EXEC SQL connect to 'db1' as 'con1';
    EXEC SQL insert into table t1 values (10); /* table t1 is in db1*/

    /* make con1 available to other threads */
    EXEC SQL set connection 'con1' dormant;

    /* Wait for con2 to become available and then update t2 */
    do {
       EXEC SQL set connection 'con2';
       } while ((SQLCODE == -1802) );
    if (SQLCODE != 0)
       return;
    EXEC SQL update t2 set a = 12 where a = 10; /* table t2 is in db1 */
```

```
        EXEC SQL set connection 'con2' dormant;
}

thread_2()
{   /* Make con2 an active connection */
    EXEC SQL connect to 'db2' as 'con2';
    EXEC SQL insert into table t2 values(10); /* table t2 is in db2*/
    /* Make con2 available to other threads */
    EXEC SQL set connection'con2' dormant;
}
```

In this code fragment, **thread_1()** uses a SET CONNECTION statement loop (see Figure 13-5 on page 13-17) to wait for **con2** to become available. Once **thread_2()** makes **con2** dormant, other threads can use this connection. At this time, the SET CONNECTION statement in **thread_1()** is successful and **thread_1()** can use the **con2** connection to update table **t2**.

The DISCONNECT ALL Statement

The DISCONNECT ALL statement serially disconnects all connections in an application. In a thread-safe Informix ESQL/C application, only the thread that issues the DISCONNECT ALL statement can be processing an SQL statement (in this case, the DISCONNECT ALL statement). If any other thread is executing an SQL statement, the DISCONNECT ALL statement fails when it tries to disconnect that connection. This failure might leave the application in an inconsistent state.

For example, suppose a DISCONNECT ALL statement successfully disconnects connection A and connection B but is unable to disconnect connection C because this connection is processing an SQL statement. The DISCONNECT ALL statement fails, with connections A and B disconnected but connection C open. It is recommended that you issue the DISCONNECT ALL statement in the main function of your application after all threads complete their work.

While the DISCONNECT ALL statement is serially disconnecting application connections, Informix ESQL/C blocks other connection requests. If another thread requests a connect while the DISCONNECT ALL statement executes, this thread must wait until the DISCONNECT ALL statement completes before Informix ESQL/C can send this new connection request to the database server.

Prepared Statements Across Threads

PREPARE statements are scoped at the connection level. That is, they are associated with a connection. When a thread makes a connection active, it can access any of the prepared statements that are associated with this connection. If your thread-safe Informix ESQL/C application uses prepared statements, you might want to isolate compilation of PREPARE statements so that they are compiled only once in a program.

One possible way to structure your application is to execute the statements that initialize the connection context as a group. The connection context includes the name of the current user and the information that the database environment associates with this name (including prepared statements).

For each connection, the application would perform the following steps:

1. Use the CONNECT statement to establish the connection that the thread requires.
2. Use the PREPARE statement to compile any SQL statements that are associated with the connection.

3. Use the SET CONNECTION...DORMANT statement to put the connection in the dormant state.

Once the connection is dormant, any thread can access the dormant connection through the SET CONNECTION statement. When the thread makes this connection active, it can send the corresponding prepared statement(s) to the database server for execution.

In Figure 13-4, the code fragment prepares SQL statements during the connection initialization and executes them later in the program.
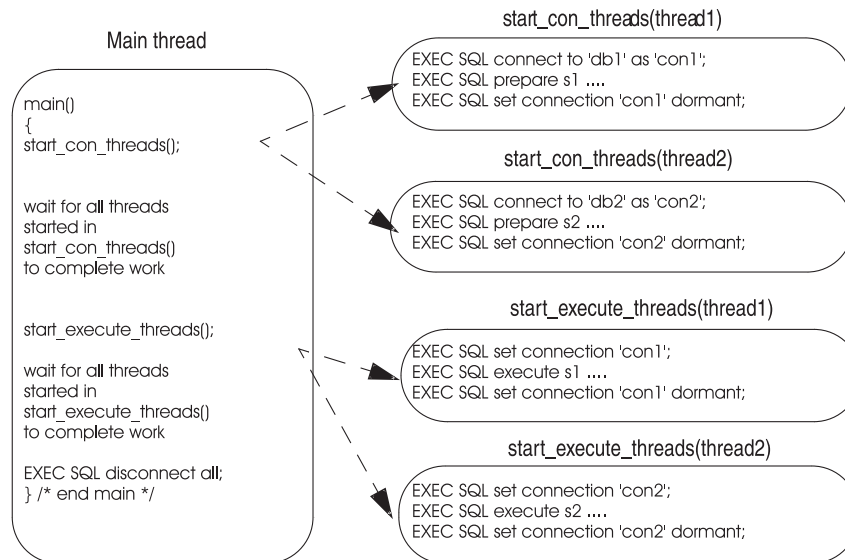
Figure 13-4. Using Prepared SQL Statements Across Threads

The code fragment in Figure 13-4 on page 13-14 performs the following actions:
1. The main thread calls **start_con_threads(), which calls start_con_thread() to start** two threads:
   - For Thread 1, the **start_con_thread()** function establishes connection **con1**, prepares a statement that is called **s1**, and makes connection **con1** dormant.
   - For Thread 2, the **start_con_thread()** function establishes connection **con2**, prepares a statement that is called **s2**, and makes connection **con2** dormant.
2. The main thread calls **start_execute_threads(), which calls start_execute_thread() to** execute the prepared statements for each of the two threads:
   - For Thread 1, the **start_execute_thread()** function makes connection **con1** active, executes the **s1** prepared statement associated with **con1**, and makes connection **con1** dormant.
   - For Thread 2, the **start_execute_thread()** function makes connection **con2** active, executes the **s2** prepared statement associated with **con2**, and makes connection **con2** dormant.
3. The main thread disconnects all connections.

Cursors Across Threads

Like prepared statements, cursors are scoped at the connection level. That is, they are associated with a connection. When a thread makes a connection active, it can access any of the database cursors that are declared for this connection. If your thread-safe Informix ESQL/C application uses database cursors, you might want to

isolate the declaration of cursors in much the same way that you can isolate prepared statements (see "Prepared Statements Across Threads" on page 13-13). The following code fragment shows a modified version of the **start_con_thread()** function (in Figure 13-4 on page 13-14). This version prepares an SQL statement and declares a cursor for that statement:

```
EXEC SQL connect to 'db1' as 'con1';
EXEC SQL prepare s1 ....
EXEC SQL declare cursor cursor1 for s1;
EXEC SQL set connection 'con1' dormant;
```

For a sample thread-safe program that uses cursors across threads, see "A Sample Thread-Safe Program" on page 13-19.

Environment Variables Across Threads

Environment variables are not thread-scoped in a thread-safe Informix ESQL/C application. That is, if a thread changes the value of a particular environment variable, this change is visible in all other threads as well.

Message File Descriptors

By default ESQL/C frees all file descriptors for a message file when it closes the message file. As a performance optimization, however, you can set the environment variable IFX_FREE_FD to cause Informix ESQL/C to *not* free the file descriptor if the message file being closed is open for another thread. If you set IFX_FREE_FD to 1, Informix ESQL/C will free the message file descriptor only for the thread that closes the file.

Decimal Functions

The **dececvt()** and **decfcvt()** functions of the Informix ESQL/C library return a character string that can be overwritten if two threads simultaneously call these functions. For this reason, use the thread-safe versions of these two functions, **ifx_dececvt()** and **ifx_decfcvt()**. For more information, see "Using ESQL/C Thread-Safe Decimal Functions" on page 13-18.

DCE Restrictions (UNIX)

A thread-safe Informix ESQL/C code is also subject to all restrictions that the DCE thread package imposes. DCE requires that all applications that use the DCE thread library be ANSI compliant. This section lists some of the restrictions to keep in mind when you create a thread-safe Informix ESQL/C application. For more information, see your DCE documentation.

Operating-System Calls: You must substitute DCE thread-jacket routines for all operating-system calls within the thread-safe Informix ESQL/C application. Thread-jacket routines take the name of a system call, but they call the DCE **pthread_lock_global_np()** function to lock the global mutual exclusion lock (mutex) before they call the actual system service. (Mutexes serialize thread execution by ensuring that only one thread at a time executes a critical section of code.) The DCE include file, **pthread.h**, defines the jacketed versions of system calls.

The fork() Operating-System Call: In the DCE environment, restrict use of the **fork()** operating-system call. In general, terminate all threads but one before you call **fork()**. An exception to this rule is when a call to the **exec()** system call immediately follows the **fork()** call. If your application uses **fork()**, it is recommended that the child process call **sqldetach()** before it executes any Informix ESQL/C statements. For more information about **sqldetach()**, see B-223.

Resource Allocation: It is recommended that you include the DCE **pthread_yield()** call in tight loops to remind the scheduler to allocate resources as necessary. The call to **pthread_yield()** instructs the DCE scheduler to try to uniformly allocate resources in the event that a thread is caught in a tight loop, waiting for a connection (thus preventing other threads from proceeding). The following code fragment shows a call to the **pthread_yield()** routine:

```
/* loop until the connection is available*/
do
   {
   EXEC SQL set connection  :con_name;
   pthread_yield();
   } while (SQLCODE == -1802);
```

Linking Thread-Safe Libraries

The **esql** command links the thread-safe versions of the static or shared Informix general libraries when you specify the **-thread** command-line option.

Linking Thread-Safe Informix General Libraries to an ESQL/C Module on a UNIX Operating System

Perform the following steps to link thread-safe Informix general libraries to an Informix ESQL/C module on a UNIX operating system:

1. Install the DCE thread package on the same client computer as the Informix ESQL/C product. For more information, see your DCE installation instructions.

    If DCE is not available on your platform, ESQL/C can use POSIX thread libraries or Sun Solaris thread libraries.

2. Set the **THREADLIB** environment variable to indicate which thread package to use when you compile the application.

    The following C-shell command sets **THREADLIB** to the DCE thread package:

    ```
    setenv THREADLIB DCE
    ```

    SOL and POSIX are also valid options for the **THREADLIB** environment variable.

Important: This version of Informix ESQL/C supports only the DCE thread package.

For more information about **THREADLIB**, see the *IBM Informix Guide to SQL: Reference*.

1. Compile your program with the **esql** command, and specify the **-thread** command-line option.

    The **-thread** command-line option tells **esql** to generate thread-safe code and to link in thread-safe libraries. The following command links thread-safe shared libraries with the **file.exe** executable file:

    ```
    esql -thread file.ec -o file.exe
    ```

The **-thread** command-line option instructs the **esql** command to perform the following steps:

1. Pass the **-thread** option to the Informix ESQL/C preprocessor to generate thread-safe code.
2. Call the C compiler with the -DIFX_THREAD command-line option.
3. Link the appropriate thread libraries (shared or static) to the executable file.

Tip: You must set the THREADLIB environment variable before you use the **esql** command with the -thread command-line option.

If you specify the **-thread** option but do not set **THREADLIB**, or if you set **THREADLIB** to some unsupported thread package, the **esql** command issues the following message:

```
esql: When using -thread, the THREADLIB environment variable
   must be set to a supported thread library. Currently
   supporting: DCE, POSIX(Solaris 2.5 and higher only) and
   SOL (Solaris Kernel Threads)
```

Defining Thread-Safe Variables (UNIX)

When you specify the **-thread** command-line option to **esql**, the Informix ESQL/C preprocessor passes the IFX_THREAD definition to the C compiler. The IFX_THREAD definition tells the C compiler to create thread-scoped variables for variables that are global in non-thread-safe Informix ESQL/C code.

For example, when the C compiler includes the **sqlca.h** file with IFX_THREAD set, it defines thread-scoped variables for the Informix ESQL/C status variables: SQLCODE, SQLSTATE, and the **sqlca** structure. The thread-scoped versions of status variables are macros that map the global status variables to thread-safe function calls that obtain thread-specific status information.

Figure 13-5 shows an excerpt from the **sqlca.h** file with the thread-scoped definitions for Informix ESQL/C status variables.

```
    .
    .
    .

extern struct sqlca_s sqlca;

extern int4 SQLCODE;

extern char SQLSTATE[];
#else /* IFX_THREAD */
extern int4 * ifx_sqlcode();
extern struct sqlca_s * ifx_sqlca();
#define SQLCODE (*(ifx_sqlcode()))
#define SQLSTATE ((char *)(ifx_sqlstate()))
#define sqlca (*(ifx_sqlca()))
#endif /* IFX_THREAD */
```

Figure 13-5. Declaration of Thread-Scoped Status Variables

Linking Shared or Static Versions

To tell the **esql** command to link the thread-safe versions of the Informix libraries into your application, use the **-thread** command-line option of **esql**, as follows:

- Thread-safe shared libraries require the **-thread** command-line option only.
- Thread-safe static libraries require the **-thread** and **-static** command-line options.

Linking Thread-Safe Informix General Libraries to an ESQL/C Module in a Windows Environment

To create a thread-safe Informix ESQL/C application, you must perform the following steps:

1. In your Informix ESQL/C source file, include the appropriate thread functions and variables of the Windows API. For more information about threads, consult your Microsoft or Borland programmer documentation.
2. When you compile the Informix ESQL/C source file, specify the **-thread** command-line option of the **esql** command.

The **-thread** option tells the Informix ESQL/C preprocessor to generate thread-safe C code when it translates SQL and Informix ESQL/C statements. This thread-safe code includes calls to thread-safe functions in the Informix DLLs.

If you are not creating an Informix ESQL/C application with threads, omit the **-thread** option. Although the Informix DLLs are thread safe, your non-thread-safe application does not use the thread-safe feature when you omit **-thread**.

Using ESQL/C Thread-Safe Decimal Functions

The **dececvt()** and **decfcvt()** functions of the Informix ESQL/C library return a character string that can be overwritten if two threads simultaneously call these functions. For this reason, use the following thread-safe versions of these two functions.

Function Name	Description	See
ifx_dececvt()	Thread-safe version of the dececvt() ESQL/C function	B-63
ifx_decfcvt()	Thread-safe version of the decfcvt() ESQL/C function	B-63

Both of these functions convert a decimal value to an ASCII string and return it in the user-defined buffer.

When you compile your Informix ESQL/C program with the **-thread** command-line option of the **esql** command, **esql** automatically links these functions to your program. These functions are described in Appendix A, "The ESQL/C example programs," on page A-1.

Context Threaded Optimization

IBM Informix ESQL/C allows developers to specify the runtime context that will be used for a set of statements. A runtime context will hold all the thread-specific data that Informix ESQL/C must maintain including connections and their current states, cursors and their current states.

This feature allows Informix ESQL/C programmers to improve the performance of their MESQL/C applications. By using the SQLCONTEXT definitions and directives, the number of thread-specific data block lookups is reduced.

The following embedded SQL statements support the definition and usage of runtime contexts:

```
SQLCONTEXT context_var;
PARAMETER SQLCONTEXT param_context_var;
BEGIN SQLCONTEXT OPTIMIZATION;
END SQLCONTEXT OPTIMIZATION;
```

The SQLCONTEXT definition and statements are only recognized when the *esql-thread* option is used. If the *-thread* option is not specified, the statements are ignored.

The use of the SQLCONTEXT statements will cause the ESQL/C preprocessor to generate code in the **.c** file that differs from the generated code when no SQLCONTEXT statements are present.

The following SQLCONTEXT definition will generate code to define and set the value of the SQLCONTEXT to the handle of the runtime context:

```
SQLCONTEXT context_var;
```

The following SQLCONTEXT is used to generate code to define a parameter that contains the handle of the runtime context:

```
PARAMETER SQLCONTEXT param_context_var;
```

The following BEGIN SQLCONTEXT directive will cause all statements positionally following it in the source file to use the indicated runtime context until the END CONTEXT directive is seen:

```
BEGIN SQLCONTEXT OPTIMIZATION;
...
END SQLCONTEXT OPTIMIZATION;
```

The END SQLCONTEXT directive should appear before the end of the scope of the SQLTCONTEXT definition currently used, or compile time errors will occur for "undefined symbol sql_context_var."

A Sample Thread-Safe Program

The following sample program, **thread_safe**, shows how you can use a cursor across threads. Sample output for this program follows the source listing.

Source Listing

The main thread starts a connection that is named **con1** and declares a cursor on table **t**. It then opens the cursor and makes connection **con1** dormant. The main thread then starts six threads (six instances of the **threads_all()** function) and waits for the threads to complete their work with the **pthread_join()** DCE call.

Each thread uses the connection **con1** and the opened cursor to perform a fetch operation. After the fetch operation, the program makes the connection dormant. Threads use connection **con1** in a sequential manner because only one thread can use the connection at a time. Each thread reads the next record from the **t** table.

```
/* **************************************************************
* Program Name: thread_safe()
*
* purpose      : If a server connection is initiated with the WITH
*                CONCURRENT TRANSACTION clause, an ongoing transaction
*                can be shared across threads that subsequently
*                connect to that server.
*                In addition, if an open cursor is associated with such
*                connection, the cursor will remain open when the
*                connection
*                 is made dormant. Therefore, multiple threads can share a
*                cursor.
*
* Methods      : - Create database db_con221 and table t1.
*                - Insert 6 rows into table t1, i.e. values 1 through 6.
*                - Connect to db_con221 as con1 with CONCURRENT
*                   TRANSACTION.
*                   The CONCURRENT TRANSACTION option is required since
*                   all
*                   threads use the cursor throughout the same
*                   connection.
*                - Declare c1 cursor for "select a from t1 order by a".
*                - Open the cursor.
*                - Start 6 threads. Use DCE pthread_join() to determine if
```

```
*                    all threads complete & all threads do same thing as
*                    follows.
*                    For thread_1, thread_2, ..., thread_6:
*                            o SET CONNECTION con1
*                            o FETCH a record and display it.
*                            o SET CONNECTION con1 DORMANT
*              - Disconnect all connections.
**************************************************************
*/

#include <pthread.h>
#include <dce/dce_error.h>

/* global definitions */
#define num_threads     6

/* Function definitions */
static  void thread_all();
static  long  dr_dbs();
static int checksql(char *str, long expected_err, char *con_name);
static void dce_err();

/* Host variables declaration */
EXEC SQL BEGIN DECLARE SECTION;
   char   con1[] = "con1";
EXEC SQL END DECLARE SECTION;

/* ****************************************************
 * Main Thread
 ****************************************************/
main()
{
/* create database */

EXEC SQL create database db_con221 with log;
if (! checksql("create database", 0, EMPTYSTR))
   {
   printf("MAIN:: create database returned status {%d}\n", SQLCODE);
   exit(1);
   }

EXEC SQL create table t1( sales int);
if (! checksql( "create_table", 0, EMPTYSTR))
   {
   dr_dbs("db_con221");
   printf("MAIN:: create table returned status {%d}\n", SQLCODE);
   exit(1);
   }

if ( populate_tab() != FUNCSUCC)
   {
   dr_dbs("db_con221");
   printf("MAIN:: returned status {%d}\n", SQLCODE);
   exit(1);
   }

EXEC SQL close database;
checksql("[main] <close database>", 0, EMPTYSTR);

/* Establish connection 'con1' */
EXEC SQL connect to 'db_con221' as 'con1' WITH CONCURRENT TRANSACTION;
if (! checksql("MAIN:: <close database>", 0, EMPTYSTR))
   {
   dr_dbs("db_con221");
```

```
        exit(1);
        }

    /* Declare cursor c1 associated with the connection con1 */
    EXEC SQL prepare tabid from "select sales from t1 order by sales";
    checksql("MAIN:: <prepare>", 0, EMPTYSTR);

    EXEC SQL declare c1 cursor for tabid;
    checksql("MAIN:: <declare c1 cursor for>", 0, EMPTYSTR);

    /* Open cursor c1 and make the connection dormant */
    EXEC SQL open c1;
    checksql("MAIN:: <open c1>", 0, EMPTYSTR);
    EXEC SQL set connection :con1 dormant;
    checksql("MAIN:: <set connection con1 dormant>", 0, EMPTYSTR);

    /* Start threads  */
    start_threads();

    /* Close cursor and drop database */
    EXEC SQL set connection :con1;
    checksql("MAIN:: set connection", 0, EMPTYSTR);
    EXEC SQL close c1;
    checksql("MAIN:: <close cursor>", 0, EMPTYSTR);
    EXEC SQL free c1;
    checksql("MAIN:: <free cursor>", 0, EMPTYSTR);

    EXEC SQL disconnect all;
    checksql("MAIN:: disconnect all", 0, EMPTYSTR);
    dr_dbs("db_con221");
    } /* end of Main Thread */

    /***********************************************************************
    * Function: thread_all()
     * Purpose : Uses connection con1 and fetches a row from table t1 using *
          cursor c1.
     * Returns : Nothing
    ***********************************************************************/
    static void thread_all(thread_num)
    int *thread_num;
    {
    EXEC SQL BEGIN DECLARE SECTION;
       int    val;
    EXEC SQL END DECLARE SECTION;

    /* Wait for the connection to become available */
    do {
       EXEC SQL set connection :con1;
       } while (SQLCODE == -1802);

    checksql("thread_all: set connection", 0, con1);

    /* Fetch a row */
    EXEC SQL fetch c1 into :val;
    checksql("thread_all: fetch c1 into :val", 0, con1);

    /* Free connection con1 */
    EXEC SQL set connection :con1 dormant;
    checksql("thread_all: set connection con1 dormant", 0, EMPTYSTR);
    printf("Thread id %d fetched value %d from t1\n", *thread_num, val);
    } /* thread_all() */

    /***********************************************************************
    * Function: start_threads()
```

```
   * purpose : Create num_threads and passes a thread id number to each
   *               thread
   *********************************************************************/

   start_threads()
   {

      int          thread_num[num_threads];
      pthread_t         thread_id[num_threads];
      int          i, ret, return_value;

   for(i=0; i< num_threads; i++)
      {
      thread_num[i] = i;
      if ((pthread_create(&thread_id[i], pthread_attr_default
           (pthread_startroutine_t ) thread_all, &thread_num[i])) == -1)
         {
         dce_err(__FILE__, "pthread_create failed", (unsigned long)-1);
         dr_dbs("db_con221");
         exit(1);
         }
      }

    /* Wait for all the threads to complete their work */
   for(i=0; i< num_threads; i++)
      {
      ret = pthread_join(thread_id[i], (pthread_addr_t *) &return_value);
      if(ret == -1)
         {
         dce_err(__FILE__, "pthread_join", (unsigned long)-1);
         dr_dbs("db_con221");
         exit(1);
         }
      }
   } /* start_threads() */


   /*********************************************************************
   * Function: populate_tab()
    * Purpose : insert values in table t1.
    * Returns : FUNCSUCC on success and FUNCFAIL when it fails.
    *********************************************************************/
   static int
   populate_tab()
   {
   EXEC SQL BEGIN DECLARE SECTION;
      int i;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL begin work;
   if (!checksql("begin work", 0,EMPTYSTR))
      return FUNCFAIL;
   for (i=1; i<=num_threads; i++)
      {
      EXEC SQL insert into t1 values (:i);
       if(!checksql("insert", 0,EMPTYSTR))
         return FUNCFAIL;
      }
   EXEC SQL commit work;
   if (!checksql("commit work", 0,EMPTYSTR))
          return FUNCFAIL;

   return FUNCSUCC;
   } /* populate_tab() */
```

```c
/***********************************************************************
 * Function: dr_dbs()
 * Purpose : drops the database.
 ***********************************************************************/
long dr_dbs(db_name)
EXEC SQL BEGIN DECLARE SECTION;
   char   *db_name;
EXEC SQL END DECLARE SECTION;

{
EXEC SQL connect to DEFAULT;
checksql("dr_dbs: connect",  0, "DEFAULT");

EXEC SQL drop database :db_name;
checksql("dr_dbs: drop database", 0, EMPTYSTR);

EXEC SQL disconnect all;
checksql("dr_dbs: disconnect all", 0, EMPTYSTR);
} /*dr_dbs() */

/***********************************************************************
 * Function: checksql()
 * Purpose : To check the SQLCODE against the expected error
 *                     (or the expected SQLCODE) and issue a proper message.
 * Returns : FUNCSUCC on success & FUNCFAIL on FAILURE.
 ***********************************************************************/
int checksql(str, expected_err, con_name)
char *str;
long expected_err;
char *con_name;
{
if (SQLCODE != expected_err)
   {
    printf( "%s %s Returned {%d}, Expected {%d}\n", str, con_name,
SQLCODE,
      expected_err);
   return(FUNCFAIL);
   }
return (FUNCSUCC);
} /* checksql() */

/***********************************************************************
 * Function: dce_err()
 * purpose : prints error from dce lib routines
 * return  : nothing
 ***********************************************************************/

void dce_err(program, routine, status)
char *program, *routine;
unsigned long status;
{
int dce_err_status;
char dce_err_string[dce_c_error_string_len+1];

if(status == (unsigned long)-1)
   {
   dce_err_status = 0;
   sprintf(dce_err_string, "returned FAILURE (errno is %d)", errno);
   }
else
   dce_error_inq_text(status, (unsigned char *)dce_err_string,
&dce_err_status);

if(!dce_err_status)
```

```
        {
         fprintf(stderr, "%s: error in %s:\n   ==> %s (%lu) <==\n",
         program, routine, dce_err_string, status);
         }
    else
         fprintf(stderr, "%s: error in %s: %lu\n", program, routine, status);
} /* dce_err() */
```

Output

The sample output might appear different each time the sample program executes
because it depends on the execution order of the threads.

```
Thread id 0 fetched value 1 from t1
Thread id 2 fetched value 2 from t1
Thread id 3 fetched value 3 from t1
Thread id 4 fetched value 4 from t1
Thread id 5 fetched value 5 from t1
Thread id 1 fetched value 6 from t1
```

In this output, Thread 1 fetches the last record in the table.

Creating a Dynamic Thread Library on UNIX Operating Systems

To create a dynamic thread library, you must define routines for every threaded
operation that Informix ESQL/C performs and you must register those functions
with Informix ESQL/C. The following list shows all of the functions that a
multithreaded Informix ESQL/C application requires and describes what each
function must do.

- **mint ifxOS_th_once(ifxOS_th_once_t *pblock, ifxOS_th_initroutine_t pfn, int *init_data)**

   This routine executes the initialization routine **pfn()**. Execute the **pfn()** functions
   only once, even if they are called simultaneously by multiple threads or multiple
   times in the same thread. The **pfn()** routine is equivalent to the DCE
   pthread_once(), or the POSIX **pthread_once()** routines.

   The **init_data** variable is used for thread packages that do not have a
   pthread_once() type routine, such as Solaris Kernel Threads. The routine can be
   simulated as follows using **init_data** as a global variable initialized to 0.

   ```
   if (!*init_data)
   {
        mutex_lock(pblock);
        if (!*init_data)
        {
           (*pfn)();
           *init_data = 1;
        }
        mutex_unlock(pblock);
   }
   return(0);
   ```

- **mint ifxOS_th_mutexattr_create(ifxOS_th_mutexattr_t *mutex_attr)**

   This function creates a mutex attributes object that specifies the attributes of
   mutexes when they are created. The mutex attributes object is initialized with
   the default value for all of the attributes defined by the user's implementation.
   This routine is equivalent to the DCE **pthread_mutexattr_create()**, or the POSIX
   pthread_mutexattr_init() routines. If a thread package does not support mutex
   attribute objects, the mutex attribute routines can be no-ops.

- **mint ifxOS_th_mutexattr_setkind_np(ifxOS_th_mutexattr_t *mutex_attr, int kind)**

This routine sets the mutex type attribute that is used when a mutex is created. The mutex attribute mutex_attr is set to type **kind**. For DCE, this routine is **pthread_mutexattr_setkind_np()**.

- **mint ifxOS_th_mutexattr_delete(ifxOS_th_mutexattr_t *mutex_attr)**

  This routine deletes the mutex attribute object mutex_attr. This routine should have the same functionality as the DCE **pthread_mutexattr_delete()**, or the POSIX **pthread_mutexattr_destroy()** routines.

- **mint ifxOS_th_mutex_init(ifxOS_th_mutex_t *mutexp, ifxOS_th_mutexattr_t mutex_attr)**

  This routine creates a mutex and initializes it to the unlocked state. This routine should have the same functionality as the DCE **pthread_mutex_init()**, or the POSIX **pthread_mutex_init()** routines.

- **mint ifxOS_th_mutex_destroy(ifxOS_th_mutex_t *mutexp)**

  This routine deletes a mutex. The mutex must be unlocked before it is deleted. This routine should have the same functionality as the DCE **pthread_mutex_destroy()**, or the POSIX **pthread_mutex_destroy()** routines.

- **mint ifxOS_th_mutex_lock(ifxOS_th_mutex_t *mutexp)**

  This routine locks an unlocked mutex. If the mutex is already locked, the calling thread should wait until the mutex becomes unlocked. This routine should have the same functionality as the DCE **pthread_mutex_lock()**, or the POSIX **pthread_mutex_lock()** routines.

- **mint ifxOS_th_mutex_trylock(ifxOS_th_mutex_t *mutexp)**

  If the mutex is successfully locked, it returns the value 1, if the mutex is locked by another thread, it returns the value 0.

  This routine should have the same functionality as the DCE **pthread_mutex_trylock()** routine.

- **mint ifxOS_th_mutex_unlock(ifxOS_th_mutex_t *mutexp)**

  This routine unlocks the mutex mutexp. If threads are waiting to lock this mutex, the implementation defines which thread receives the mutex. This routine should have the same functionality as the DCE **pthread_mutex_unlock()**, or the POSIX **pthread_mutex_unlock()** routines.

- **mint ifxOS_th_condattr_create(ifxOS_th_condattr_t *cond_attr)**

  This routine creates an object that is used to specify the attributes of condition variables when they are created. Initialize the object with the default value for all of the attributes defined by the user's implementation. This routine should have the same functionality as the DCE **pthread_condattr_create()**, or the POSIX **pthread_condattr_init()** routines.

- **mint ifxOS_th_cond_init(ifxOS_th_cond_t *condp, ifxOS_th_condattr_t cond_attr)**

  This routine creates and initializes a condition variable. It should have the same functionality as the DCE **pthread_cond_init()**, or the POSIX **pthread_cond_init()** routines.

- **mint ifxOS_th_condattr_delete(ifxOS_th_condattr_t *cond_attr)**

  This routine deletes the condition variable attribute object cond_attr. This routine should have the same functionality as the DCE **pthread_condattr_delete()**, or POSIX **pthread_condattr_destroy()** routines.

- **mint ifxOS_th_cond_destroy(ifxOS_th_cond_t *condp)**

  This routine deletes the condition variable condp. The routine should have the same functionality as the DCE **pthread_cond_destroy()**, or the POSIX **pthread_cond_destroy()** routines.

- **mint ifxOS_th_cond_timedwait(ifxOS_th_cond_t *sleep_cond, ifxOS_th_mutex_t *sleep_mutex, ifxOS_th_timespec_t *t)**

    This routine causes a thread to wait until either the condition variable sleep_cond is signaled or broadcast, or the current system clock time becomes greater than or equal to the time specified in **t**. The routine should have the same functionality as the DCE **pthread_cond_timedwait()**, or the POSIX **pthread_cond_timedwait()** routines.

- **mint ifxOS_th_keycreate(ifxOS_th_key_t *allkey, ifxOS_th_destructor_t AllDestructor)**

    This routine generates a unique value that identifies a thread-specific data value. This routine should have the same functionality as the DCE **pthread_keycreate()**, or the POSIX **pthread_key_create()** routines.

- **mint ifxOS_th_getspecific(ifxOS_th_key_t key, ifxOS_th_addr_t *tcb)**

    This routine obtains the thread-specific data associated with the key. This routine should have the same functionality as the DCE **pthread_getspecific()**, or the POSIX **pthread_getspecific()** routines.

- **mint ifxOS_th_setspecific(ifxOS_th_key_t key, ifxOS_th_addr_t tcb)**

    This routine sets the thread-specific data in the tcb associated with the **key** for the current thread. If a value is already defined for **key** in the current thread, the new value is substituted for the existing value. This routine should have the same functionality as the DCE **pthread_setspecific()**, or the POSIX **pthread_setspecific()** routines.

Data Types

You can create typedefs for the data types in the preceding functions to the equivalent data types in your thread package, or you can use the appropriate data type from the thread package instead of the **ifxOS_** version. The following list includes all the data types that the preceding functions use:

- **ifxOS_th_mutex_t**

    This structure defines a mutex object: pthread_mutex_t in DCE and POSIX.

- **ifxOS_th_mutexattr_t**

    This structure defines a mutex attributes object called pthread_mutexattr_t in DCE and POSIX. If mutex attribute objects are unsupported in your thread package (for instance, Solaris Kernel Threads), you can assign them a data type of **mint**.

- **ifxOS_th_once_t**

    This structure allows client initialization operations to guarantee mutually exclusive access to the initialization routine, and to guarantee that each initialization is executed only once. This routine should have the same functionality as the pthread_once_t structure in DCE and POSIX.

- **ifxOS_th_condattr_t**

    This structure defines an object that specifies the attributes of a condition variable: pthread_condattr_t in DCE and POSIX. If this object is unsupported in your thread package (for instance, Solaris Kernel Threads), you can assign it a data type of **mint**.

- **ifxOS_th_cond_t**

    This structure defines a condition variable called pthread_cond_t in DCE and POSIX.

- **ifxOS_th_timespec_t**

This structure defines an absolute time at which the ifxOS_th_cond_timedwait() function should time out if a condition variable has not been signaled or broadcast. This structure is **timespec_t** in DCE and POSIX.

- **ifxOS_th_key_t**

  This structure defines a thread-specific data key used in the ifxOS_th_keycreate(), **ifxOS_th_setspecific()** and **ifxOS_getspecific()** routines. This structure is **pthread_key_t** in DCE and POSIX.

- **ifxOS_th_addr_t**

  This structure defines an address that contains data to be associated with a thread-specific data key of type ifxOS_th_key_t. The **ifxOS_th_addr_t** structure is equivalent to **pthread_addr_t** in DCE. You can specify **void *** as an alternative that can be used for thread packages (such as POSIX) that do not define such a structure.

The following example uses the Solaris Kernel Threads package to demonstrate how to set up a dynamic-thread library. The first task is to define the 17 dynamic-thread functions that the shared and/or static library needs. In this example, the file is called **dynthr.c**:

```
/* Prototypes for the dynamic thread functions */

mint ifx_th_once(mutex_t *pblock, void (*pfn)(void), mint *init_data);
mint ifx_th_mutexattr_create(mint *mutex_attr);
mint ifx_th_mutexattr_setkind_np(mint *mutex_attr, mint kind);
mint ifx_th_mutexattr_delete(mint *mutex_attr);
mint ifx_th_mutex_init(mutex_t *mutexp, mint mutex_attr);
mint ifx_th_mutex_destroy(mutex_t *mutexp);
mint ifx_th_mutex_lock(mutex_t *mutexp);
mint ifx_th_mutex_trylock(mutex_t *mutexp);
mint ifx_th_mutex_unlock(mutex_t *mutexp);
mint ifx_th_condattr_create(mint *cond_attr);
mint ifx_th_cond_init(cond_t *condp, mint cond_attr);
mint ifx_th_condattr_delete(mint *cond_attr);
mint ifx_th_cond_destroy(cond_t *condp);
mint ifx_th_cond_timedwait(cond_t *sleep_cond, mutex_t *sleep_mutex,
    timestruc_t *t);
mint ifx_th_keycreate(thread_key_t *allkey, void (*AllDestructor)
    (void *));
mint ifx_th_getspecific(thread_key_t key, void **tcb);
mint ifx_th_setspecific(thread_key_t key, void *tcb);

/*
 * The functions . . . *
 *                    */

mint ifx_th_once(mutex_t *pblock, void (*pfn)(void), mint *init_data)
{
   if (!*init_data)
   {
      mutex_lock(pblock);
      if (!*init_data)
      {
         (*pfn)();
         *init_data = 1;
      }
      mutex_unlock(pblock);
   }
   return(0);
}

/* Mutex attributes are not supported in solaris kernel threads *
 * The functions must be defined anyway, to avoid accessing     *
 * a NULL function pointer.                                      */
```

```
mint ifx_th_mutexattr_create(mint *mutex_attr)
{
   *mutex_attr = 0;
   return(0);
}

/* Mutex attributes are not supported in solaris kernel threads */
mint ifx_th_mutexattr_setkind_np(mint *mutex_attr, mint kind)
{
   *mutex_attr = 0;
   return(0);
}

/* Mutex attributes are not supported in solaris kernel threads */
mint ifx_th_mutexattr_delete(mint *mutex_attr)
{
   return(0);
}

mint ifx_th_mutex_init(mutex_t *mutexp, mint mutex_attr)
{
   return(mutex_init(mutexp, USYNC_THREAD, (void *)NULL));
}

mint ifx_th_mutex_destroy(mutex_t *mutexp)
{
   return(mutex_destroy(mutexp));
}

mint ifx_th_mutex_lock(mutex_t *mutexp)
{
   return(mutex_lock(mutexp));
}
/* Simulate mutex_trylock using mutex_lock */
mint ifx_th_mutex_trylock(mutex_t *mutexp)
{
   mint ret;

   ret = mutex_trylock(mutexp);
   if (ret == 0)
      return(1); /* as per the DCE call */
   if (ret == EBUSY)
      return(0); /* as per the DCE call */
   return(ret);
}

mint ifx_th_mutex_unlock(mutex_t *mutexp)
{
   return(mutex_unlock(mutexp));
}

/* Condition attributes are not supported in solaris kernel threads */
mint ifx_th_condattr_create(mint *cond_attr)
{
   *cond_attr = 0;
   return(0);
}

mint ifx_th_cond_init(cond_t *condp, mint cond_attr)
{
   return(cond_init(condp, USYNC_THREAD, (void *)NULL));
}

mint ifx_th_condattr_delete(int *cond_attr)
{
   return(0);
```

```
}

mint ifx_th_cond_destroy(cond_t *condp)
{
   return(cond_destroy(condp));
}

mint ifx_th_cond_timedwait(cond_t *sleep_cond, mutex_t
   *sleep_mutex,timestruc_t
   *t)
{
   return(cond_timedwait(sleep_cond, sleep_mutex, t));
}

mint ifx_th_keycreate(thread_key_t *allkey, void (*AllDestructor)
   (void *))
{
   return(thr_keycreate(allkey, AllDestructor));
}

mint ifx_th_getspecific(thread_key_t key, void **tcb)
{
   return(thr_getspecific(key, tcb));
}

mint ifx_th_setspecific(thread_key_t key, void *tcb)
{
   return(thr_setspecific(key, tcb));
```

Registering the Dynamic Thread Functions

Your Informix ESQL/C application must use the **ifxOS_set_thrfunc()** function to
register the dynamic thread functions with Informix ESQL/C. The following
declaration describes the **ifxOS_set_thrfunc()** function.

```
mint ifxOS_set_thrfunc(mint func, mulong (*funcptr)())
```

The first parameter, **func**, is a **mint** that indexes the function being registered. The
second parameter is the name of the function that is being registered.

You must call **ifxOS_set_thrfunc()** once for each of the 17 **ifxOS** functions listed in
"Creating a Dynamic Thread Library on UNIX Operating Systems" on page 13-24.

The **ifxOS_set_thrfunc()** function returns 0 if it successfully registers the function
and **-1** if it fails to register the function. For example, to register the user-defined
function **my_mutex_lock()** as the **ifxOS_th_mutex_lock routine**, you use the
following call:

```
if (ifxOS_set_thrfunc(TH_MUTEX_LOCK, (mulong (*)())my_mutex_lock)
== -1)
```

TH_MUTEX_LOCK is defined in **sqlhdr.h** and tells the client to call
my_mutex_lock() whenever it needs to lock a mutex.

The following list shows the indexes and the functions they register.

Index	Function
TH_ONCE	**ifxOS_th_once**
TH_MUTEXATTR_CREATE	**ifxOS_th_mutexattr_create()**
TH_MUTEXATTR_SETKIND	**ifxOS_th_mutexattr_setkind_np()**
TH_MUTEXATTR_DELETE	**ifxOS_th_mutexattr_delete()**

TH_MUTEX_INIT	**ifxOS_th_mutex_init()**
TH_MUTEX_DESTROY	**ifxOS_th_mutex_destroy()**
TH_MUTEX_LOCK	**ifxOS_th_mutex_lock()**
TH_MUTEX_UNLOCK	**ifxOS_th_mutex_unlock()**
TH_MUTEX_TRYLOCK	**ifxOS_th_mutex_trylock()**
TH_CONDATTR_CREATE	**ifxOS_th_condattr_create()**
TH_CONDATTR_DELETE	**ifxOS_th_condattr_delete()**
TH_COND_INIT	**ifxOS_th_cond_init()**
TH_COND_DESTROY	**ifxOS_th_cond_destroy()**
TH_COND_TIMEDWAIT	**ifxOS_th_cond_timedwait()**
TH_KEYCREATE	**ifxOS_th_keycreate()**
TH_GETSPECIFIC	**ifxOS_th_getspecific()**
TH_SETSPECIFIC	**ifxOS_th_setspecific()**

The following function, **dynthr_init()**, which is also defined in **dynthr.c**, registers
the 17 functions defined in "Creating a Dynamic Thread Library on UNIX
Operating Systems" on page 13-24. FUNCFAIL is defined to be -1.

```
dynthr_init()
{
   if (ifxOS_set_thrfunc(TH_ONCE, (mulong (*)())ifx_th_once)
== FUNCFAIL)
      return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEXATTR_CREATE,
      (mulong (*)())ifx_th_mutexattr_create) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEXATTR_SETKIND,
      (mulong (*)())ifx_th_mutexattr_setkind_np) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEXATTR_DELETE,
      (mulong (*)())ifx_th_mutexattr_delete) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEX_INIT,
      (mulong (*)())ifx_th_mutex_init) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEX_DESTROY,
      (mulong (*)()) ifx_th_mutex_destroy) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEX_LOCK,
      (mulong (*)()) ifx_th_mutex_lock) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEX_UNLOCK,
      (mulong (*)())ifx_th_mutex_unlock) == FUNCFAIL)
         return FUNCFAIL;

   if (ifxOS_set_thrfunc(TH_MUTEX_TRYLOCK,
      (mulong (*)())ifx_th_mutex_trylock) == FUNCFAIL)
         return FUNCFAIL;
   if (ifxOS_set_thrfunc(TH_CONDATTR_CREATE,
      (mulong (*)())ifx_th_condattr_create) == FUNCFAIL)
```

```
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_CONDATTR_DELETE,
        (mulong (*)())ifx_th_condattr_delete) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_COND_INIT,
        (mulong (*)())ifx_th_cond_init) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_COND_DESTROY,
        (mulong (*)())ifx_th_cond_destroy) == FUNCFAIL)
            return FUNCFAIL;
    if (ifxOS_set_thrfunc(TH_COND_TIMEDWAIT,
        (mulong (*)())ifx_th_cond_timedwait) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_KEYCREATE,
        (mulong (*)())ifx_th_keycreate) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_GETSPECIFIC,
        (mulong (*)())ifx_th_getspecific) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_SETSPECIFIC,
        (mulong (*)())ifx_th_setspecific) == FUNCFAIL)
            return FUNCFAIL;
    return 0;

}
```

Setting the $THREADLIB Environment Variable

The following C-shell command sets THREADLIB to specify a user-defined thread package:

```
setenv THREADLIB DYNAMIC
```

Creating the Shared Library

You must compile **dynthr.c** into a shared or static library. The following example illustrates how to compile a shared or static library on a workstation running the Solaris operating system:

```
% cc -c -DIFX_THREAD -I$INFORMIXDIR/incl/esql -D_REENTRANT -K pic
dynthr.c
% ld -G -o libdynthr.so dynthr.o
% cp libdynthr.so /usr/lib            <== as root
```

You can also use the $LD_LIBRARY_PATH environment variable:

```
% cc -c -DIFX_THREAD -I$INFORMIXDIR/incl/esql -D_REENTRANT -K pic
dynthr.c
% cp dynthr.so <some directory>
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:<some directory>
```

To compile **dynthr.c** into a static library, perform the following tasks (on Solaris):

```
% cc -c -DIFX_THREAD -I$INFORMIXDIR/incl/esql -D_REENTRANT dynthr.c

% ar -cr dynthr.a dynthr.o
```

You must update your application, **test.ec**, to call the **dynthr_init()** routine first, or none of the thread functions will be registered.

```
void main(argc , argv )
int argc;
char *argv[] ;
{    /* begin main */

   /* First, set up the dynamic thread library */

   dynthr_init();

   /* Rest of program */

   EXEC SQL database stores7;

.
.
.

}
```

Compiling with the -thread and -l Preprocessor Options

You must compile the application using the **-thread** and the **-l** preprocessor
options. The **-thread** option indicates that you are linking thread-safe libraries
instead of the default Informix shared libraries. The **-l** option allows you to specify
system libraries that you want to link. Finally, you compile your application, link
libdynthr.so and run it, as shown in the following example:

```
% setenv THREADLIB "dynamic"
% esql -thread -ldynthr test.ec -o test.exe
% test.exe
```

Part 3. Dynamic SQL

Chapter 14. Using Dynamic SQL

A *static* SQL statement is one for which all the information is known at compile time. For example, the following SELECT statement is a static SQL statement because all information needed for its execution is present at compile time.

```
EXEC SQL select company into :cmp_name from customer where customer_num
= 101;
```

However, in some applications the programmer does not know the contents, or possibly even the types, of SQL statements that the program needs to execute. For example, a program may prompt the user to enter a select statement, so that the programmer has no idea what columns will be accessed when the program is run. Such applications require *dynamic SQL*. Dynamic SQL allows an IBM Informix ESQL/C program to build an SQL statement at runtime, so that the contents of the statement can be determined by user input.

These topics describe the following dynamic SQL information:
- How to execute a dynamic SQL statement, the SQL statements to use, and the types of statements that you can execute dynamically
- How to execute SQL statements when you know most of the information about the statement at compile time

For information about how to execute SQL statements when you do not know all the information at compile time, see Chapter 15, "Determining SQL statements," on page 15-1.

Using Dynamic SQL

To execute an SQL statement, the database server must have the following information about the statement:
- The type of statement, such as SELECT, DELETE, EXECUTE PROCEDURE, or GRANT
- The names of any database objects, such as tables, columns, and indexes
- Any WHERE-clause conditions, such as column names and matching criteria
- Where to put any returned values, such as the column values from the select list of a SELECT statement
- Values that need to be sent to the database server, such as the column values for a new row for an INSERT statement

If information in an SQL statement varies according to some conditions in the application, your Informix ESQL/C program can use dynamic SQL to build the SQL statement at runtime. The basic process to dynamically execute SQL statements consists of the following steps:

1. Assemble the text of an SQL statement in a character-string variable.
2. Use a PREPARE statement to have the database server examine the statement text and prepare it for execution.
3. Execute the prepared statement with the EXECUTE or OPEN statement.
4. Free dynamic resources that are used to execute the prepared statement.

Assembling and Preparing the SQL Statement

Dynamic SQL allows you to assemble an SQL statement in a character string as the user interacts with your program. A dynamic SQL statement is like any other SQL statement that is embedded into a program, except that the statement string cannot contain the names of any host variables. The PREPARE statement sends the contents of an SQL statement string to the database server, which parses it and creates a *statement identifier structure* (statement identifier).

Assembling the Statement
Assign the text for the SQL statement to a single host variable, which appears in the PREPARE statement. The key to dynamically execute an SQL statement is to

assemble the text of the statement into a character string. You can assemble this statement string in the following two ways:

- As a fixed string, if you know all the information at compile time
- As a series of string operations, if you do not have all the information at compile time

If you know the whole statement structure, you can list it after the FROM keyword of the PREPARE statement. Single quotes or double quotes around the statement text are valid, although the ANSI SQL standard specifies single quotes.

For example:
```
EXEC SQL prepare slct_id from
   'select company from customer where customer_num = 101';
```

Tip: Although Informix ESQL/C does not allow newline characters in quoted strings, you can include newline characters in the quoted string of a PREPARE statement. The quoted string is passed to the database server with the PREPARE statement and, if you specify that it should, the database server will allow newline characters in quoted strings. Therefore, you can allow a user to enter the preceding SQL statement from the command line as follows:
```
select lname from customer
where customer_num = 101
```

For more information about allowing newlines in quoted strings, see "Including Newline in Quoted Strings" on page 1-7.

Alternatively, you can copy the statement into a **char** variable as shown in Figure 14-1.

```
stcopy("select company from customer where customer_num = 101", stmt_txt);
EXEC SQL prepare slct_id from :stmt_txt;
```

Figure 14-1. Preparing a SELECT Statement from a Character String

Both of these methods have the same restriction as a static SQL statement. They assume that you know the entire statement structure at compile time. The disadvantage of these dynamic forms over the static one is that any syntax errors encountered in the statement will not be discovered until runtime (by the PREPARE statement). If you statically execute the statement, the Informix ESQL/C preprocessor can uncover syntactic errors at compile time (semantic errors might remain undiagnosed until runtime). You can improve performance when you dynamically execute an SQL statement that is to be executed more than once. The statement is parsed only once.

In Figure 14-1, the **stmt_txt** variable is a host variable because it is used in an embedded SQL statement (the PREPARE statement). Also the INTO clause of the SELECT statement was removed because host variables cannot appear in a statement string. Instead, you specify the host variables in the INTO clause of an EXECUTE or FETCH statement (see 14-6). Other SQL statements like DESCRIBE, EXECUTE, and FREE can access the prepared statement when they specify the **slct_id** statement identifier.

Important: By default, the scope of a statement identifier is global. If you create a multifile application and you want to restrict the scope of a statement identifier to a single file, preprocess the file with the **-local** preprocessor option.

For more information about **-local**, see Chapter 1, "Programming with IBM Informix ESQL/C," on page 1-1.

If you do not know all the information about the statement at compile time, you can use the following features to assemble the statement string:

- The **char** host variables can hold the identifiers in the SQL statement (column names or table names) or parts of the statement like the WHERE clause. They can also contain keywords of the statement.
- If you know what column values the statement specifies, you can declare host variables to provide column values that are needed in a WHERE clause or to hold column values that are returned by the database server.
- Input-parameter placeholders, represented by a question mark (?), in a WHERE clause indicate a column value to be provided, usually in a host variable at time of execution. Host variables used in this way are called *input parameters*. For more information, see "Execute statements with input parameters" on page 14-33.
- You can use Informix ESQL/C string library functions like **stcopy()** and **stcat()**. For more information, see Chapter 4, "Working with character and string data types," on page 4-1.

Figure 14-2 shows the SELECT statement of Figure 14-1 changed so that it uses a host variable to determine the customer number dynamically.

```
stcopy("select company from customer where customer_num = ", stmt_txt);
stcat(cust_num, stmt_txt);
EXEC SQL prepare slct_id from :stmt_txt;
```

Figure 14-2. Using a Host Variable to Assemble a SELECT Statement

Figure 14-3 shows how you can use an input parameter to program this same SELECT statement so that the user can enter the customer number.

```
EXEC SQL prepare slct_id from
     'select company from customer where customer_num = ?';
```

Figure 14-3. Using an Input Parameter to Assemble a SELECT Statement

You can prepare almost any SQL statement dynamically. The only statements that you cannot prepare dynamically are those directly concerned with dynamic SQL and cursor management (such as FETCH and OPEN), and the SQL connection statements. For a complete list of statements, see the PREPARE statement in the *IBM Informix Guide to SQL: Syntax*.

Tip: You can use the Deferred-PREPARE feature to defer execution of a prepared SELECT, INSERT, or EXECUTE FUNCTION statement until the OPEN statement.

For more information, see "Deferring Execution of the PREPARE Statement" on page 14-17.

Preparing Statements That Have Collection Variables

You use the Collection Derived Table clause with an INSERT or SELECT statement to access an Informix ESQL/C **collection** variable. (For more information about how to use the Collection Derived Table clause and **collection** variables, see Chapter 9 of this publication.)

When you prepare a statement that manipulates an Informix ESQL/C **collection** variable, the following restrictions apply:

- You must specify the statement text as a quoted string in the PREPARE statement.

  For **collection** variables, Informix ESQL/C does *not* support statement text that is stored in a program variable.

- The quoted string for the statement text *cannot* contain any **collection** host variables.

  To manipulate a **collection** variable, you must use the question mark (?) symbol to indicate an input parameter and then provide the **collection** variable when you execute the statement.

- You cannot perform multi-statement prepares if a statement contains a **collection** variable.

For example, the following Informix ESQL/C code fragment prepares an INSERT on the **a_set** client **collection** variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare coll_stmt from
   'insert into table values (1, 2, 3)';
EXEC SQL execute coll_stmt using :a_set;
```

Important: You must declare an Informix ESQL/C collection variable as a client **collection** variable (a collection variable that is stored on the client computer).

Checking the Prepared Statement

When PREPARE sends the statement string to the database server, the database server parses it to analyze it for errors. The database server indicates the success of the parse in the **sqlca** structure, as follows:

- If the syntax is correct, the database server sets the following **sqlca** fields:
  - The **sqlca.sqlcode** field (SQLCODE) contains zero (0).
  - The **sqlca.sqlerrd[0]** field contains an estimate of the number of rows affected if the parsed statement was a SELECT, UPDATE, INSERT, or DELETE.
  - The **sqlca.sqlerrd[3]** field contains an estimated cost of execution if the parsed statement was a SELECT, UPDATE, INSERT, or DELETE. This execution cost is a weighted sum of disk accesses and total number of rows processed.
- If the statement string contains a syntax error, or if some other error was encountered during the PREPARE, the database server sets the following **sqlca** fields:
  - The **sqlca.sqlcode** field (SQLCODE) is set to a negative number (<0). The database server also sets the SQLSTATE variable to an error code.
  - The **sqlca.sqlerrd[4]** field contains the offset into the statement text where the error was detected.

Executing the SQL Statement

Once an SQL statement was prepared, the database server can execute it. The way to execute a prepared statement depends on:

- How many rows (groups of values) the SQL statement returns:
  - Statements that return *one row of data* include a singleton SELECT and an EXECUTE FUNCTION statement.
  - Statements that can return *more than one row of data* require a cursor to execute; they include a non-*singleton* SELECT and an EXECUTE FUNCTION statement.
  - All other SQL statements, including EXECUTE PROCEDURE, return *no rows of data*.

  For more information about how to execute statements that require cursors, see "Using a Database Cursor" on page 14-8.

- Whether the statement has input parameters

  If so, the statement must be executed with the USING clause:
  - For SELECT and INSERT statements, use the OPEN...USING statement.
  - For non-SELECT statements, use the EXECUTE...USING statement.

- Whether you know the data types of statement columns at compile time:
  - When you know the number and data types of the columns at compile time, you can use host variables to hold the column values.

    For more information, see "SQL Statements That Are Known at Compile Time" on page 14-25.
  - When you do *not* know the number and data types of columns at compile time, you must use the DESCRIBE statement to define the column and a dynamic-management structure to hold the column values.

    For more information, see Chapter 15, "Determining SQL statements," on page 15-1.

Table 14-1 summarizes how to execute the different types of prepared SQL statements.

Table 14-1. Executing a Prepared SQL Statement

Type of SQL Statement	Input Parameters	Statement to Execute	See
Statements that do not return rows (except INSERT that is associated with a cursor)			
With no input parameters	No	EXECUTE	14-25
When number and data types of input parameters are known	Yes	EXECUTE...USING	14-34
When number and data types of input parameters are *not* known	Yes	EXECUTE...USING SQL DESCRIPTOR EXECUTE...USING DESCRIPTOR	16-28 17-27
INSERT that is associated with a cursor			
With no input parameters	No	OPEN	14-28
When number and data types of input parameters (insert columns) are known	Yes	OPEN...USING	14-35, 15-17

Table 14-1. Executing a Prepared SQL Statement (continued)

Type of SQL Statement	Input Parameters	Statement to Execute	See
When number and data types of input parameters are *not* known	Yes	OPEN...USING SQL DESCRIPTOR OPEN...USING DESCRIPTOR	16-18 17-18
Statements that can return *more than one row*: non-singleton SELECT, SPL function			
With no input parameters	No	OPEN	14-28
When number and data types of select-list columns are *not* known	No	OPEN	16-9, 17-11
When number and data types of return values are *not* known	No	OPEN	16-17, 17-17
When number and data types of input parameters are known	Yes	OPEN...USING	14-35
When number and data types of input parameters are *not* known	Yes	OPEN...USING SQL DESCRIPTOR OPEN...USING DESCRIPTOR	16-23 17-21
Statements that return *only one row*: singleton SELECT, any external function or an SPL function that returns only one group of values			
With no input parameters	No	EXECUTE...INTO	14-27
When number and data types of returned values are *not* known	No	EXECUTE...INTO DESCRIPTOR EXECUTE...INTO SQL DESCRIPTOR	16-9, 16-1417-11, 17-17
When number and data types of input parameters are known	Yes	EXECUTE...INTO ...USING	14-34
When number and data types of input parameters are *not* known	Yes	EXECUTE...INTO ...USING SQL DESCRIPTOR EXECUTE...INTO ...USING DESCRIPTOR	16-27 17-25

Freeing Resources

Sometimes you can ignore the cost of resources allocated to prepared statements and cursors. However, the number of prepared objects that the application can create is limited. Free resources that IBM Informix ESQL/C uses to execute a prepared statement, as follows:

- If the statement is associated with a cursor, use CLOSE to close the cursor after all the rows are fetched (or inserted).
- Use the FREE statement to release the resources allocated for the prepared statement and any associated cursor. Once you have freed a prepared statement, you can no longer use it in your program until you reprepare or redeclare it. However, once you declare the cursor, you can free the associated statement identifier but not affect the cursor.

You can use the AUTOFREE feature to have the database server automatically free resources for a cursor and its prepared statement. For more information, see "Automatically Freeing a Cursor" on page 14-13.

If your program uses a dynamic-management structure to describe an SQL statement at runtime, also deallocate the resources of this structure once the structure is no longer needed. For information about how to deallocate a system-descriptor area, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8. For information about how to deallocate an **sqlda** structure, see "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

Using a Database Cursor

A *database cursor* is an identifier associated with a group of rows. It is, in a sense, a pointer to the current row in a buffer. You *must* use a cursor in the following cases:

- Statements that *return* more than one row of data from the database server:
  - A SELECT statement requires a select cursor.
  - An EXECUTE FUNCTION statement requires a function cursor.
- An INSERT statement that *sends* more than one row of data to the database server requires an insert cursor.

The following sections summarize how to use cursors in an Informix ESQL/C application. For more information about how to use cursors, see the *IBM Informix Guide to SQL: Tutorial*.

Receiving More Than One Row

Statements that return *one row of data* include a singleton SELECT and an EXECUTE FUNCTION statement whose user-defined function returns only one row of data. Statements that can return *more than one row of data* include:

- a non-*singleton* SELECT.

  When a SELECT statement returns more than one row, define a *select cursor* with the DECLARE statement.

- an EXECUTE FUNCTION statement whose user-defined function returns more than one row.

  When an EXECUTE FUNCTION statement executes a user-defined function that returns more than one row, define a *function cursor* with the DECLARE statement.

For the select or function cursor, you can use a sequential, scroll, hold, or update cursor. Table 14-2 summarizes the SQL statements that manage a select or function cursor.

Table 14-2. SQL Statements That Manage a Select or Function Cursor

Task	Select Cursor	Function Cursor
Declare the cursor identifier	DECLARE associated with a SELECT statement	DECLARE associated with an EXECUTE FUNCTION statement
Execute the statement	OPEN	OPEN
Access a single row from the fetch buffer into the program	FETCH	FETCH
Close the cursor	CLOSE	CLOSE
Free cursor resources	FREE	FREE

For more information about any of these statements, see their entries in the *IBM Informix Guide to SQL: Syntax*. You can change the size of the select or fetch buffer with the Fetch-Buffer-Size feature. For more information, see "Sizing the Cursor Buffer" on page 14-11.

A Select Cursor

A *select cursor* enables you to scan multiple rows of data that a SELECT statement returns. The DECLARE statement associates the SELECT statement with the select cursor. In the DECLARE statement, the SELECT statement can be in either of the following formats:

- A literal SELECT statement in the DECLARE statement

  The following DECLARE statement associates a literal SELECT statement with the **slct1_curs** cursor:

  ```
  EXEC SQL declare slct1_curs cursor for select * from customer;
  ```

- A prepared SELECT statement in the DECLARE statement

  The following DECLARE statement associates a prepared SELECT statement with the **slct2_curs** cursor:

  ```
  EXEC SQL prepare slct_stmt cursor from
      'select * from customer';
  EXEC SQL declare slct2_curs for slct_stmt;
  ```

If the SELECT returns only one row, it is called a singleton SELECT and it does not require a select cursor to execute.

A Function Cursor

A *function cursor* enables you to scan multiple rows of data that the user-defined function returns. The following user-defined functions can return more than one row:

- An SPL function that contains the WITH RESUME keywords in its RETURN statement

  For information about how to write this type of SPL function, see the chapter on SPL in the *IBM Informix Guide to SQL: Tutorial*.

- An external function that is an iterator function

  For information about how to write an iterator function, see the *IBM Informix DataBlade® API Programmer's Guide*.

You execute a user-defined function with the EXECUTE FUNCTION statement. The DECLARE statement associates the EXECUTE FUNCTION with the function cursor. In the DECLARE statement, the EXECUTE FUNCTION statement can be in either of the following formats:

- A literal EXECUTE FUNCTION statement in the DECLARE statement

  The following DECLARE statement associates a literal EXECUTE FUNCTION statement with the **func1_curs** cursor:

  ```
  EXEC SQL declare func1_curs cursor for execute function
      func1();
  ```

- A prepared EXECUTE FUNCTION statement in the DECLARE statement

  The following DECLARE statement associates a prepared EXECUTE FUNCTION statement with the **func2_curs** cursor:

  ```
  EXEC SQL prepare func_stmt from
      'execute function func1()';
  EXEC SQL declare func2_curs cursor for func_stmt;
  ```

If the external or SPL function returns only one row, it does not require a function cursor to execute.

Sending More Than One Row

When you execute the INSERT statement, the statement sends one row of data to the database server. When an INSERT statement sends more than one row, define an *insert cursor* with the DECLARE statement. An insert cursor enables you to buffer multiple rows of data for insertion at one time. The DECLARE statement associates the INSERT statement with the insert cursor. In the DECLARE statement, the INSERT statement can be in either of the following formats:

- A literal INSERT statement in the DECLARE statement

  The following DECLARE statement associates a literal INSERT statement with the **ins1_curs** cursor:

  ```
  EXEC SQL declare ins1_curs cursor for
        insert into customer values;
  ```

- A prepared INSERT statement in the DECLARE statement

  The following DECLARE statement associates a prepared INSERT statement with the **ins2_curs** cursor:

  ```
  EXEC SQL prepare ins_stmt from
        'insert into customer values';
  EXEC SQL declare ins2_curs cursor for ins_stmt;
  ```

If you use an insert cursor it can be much more efficient than if you insert rows one at a time, because the application process does not need to send new rows to the database as often. You can use a sequential or hold cursor for the insert cursor. Table 14-3 summarizes the SQL statements that manage an insert cursor.

Table 14-3. SQL Statements That Manage an Insert Cursor

Task	Insert Cursor
Declare the cursor ID	DECLARE associated with an INSERT statement
Execute the statement	OPEN
Send a single row from the program into the insert buffer	PUT
Clear the insert buffer and send the contents to the database server	FLUSH
Close the cursor	CLOSE
Free cursor resources	FREE

For more information about any of these statements, see their entries in the *IBM Informix Guide to SQL: Syntax*. You can change the size of the insert buffer with the Fetch-Buffer-Size feature. For more information, see "Sizing the Cursor Buffer" on page 14-11.

Naming the Cursor

In an Informix ESQL/C program, you can specify a cursor name with any of the following items:

- A literal name must follow the rules for identifier names. See the Identifier segment in the *IBM Informix Guide to SQL: Syntax*.

- A *delimited identifier* is an identifier name that contains characters that do not conform to identifier-naming rules. For more information, see "SQL Identifiers" on page 1-12.
- A *dynamic cursor* is a character host variable that contains the name of the cursor. This type of cursor specification means that the cursor name is specified dynamically by the value of the host variable. You can use a dynamic cursor in any SQL statement that allows a cursor name *except* the WHERE CURRENT OF clause of the DELETE or UPDATE statement.

  Dynamic cursors are particularly useful to create generic functions to perform cursor-management tasks. You can pass in the name of the cursor as an argument to the function. If the cursor name is to be used in an Informix ESQL/C statement within the function, make sure you declare the argument as a host variable with the PARAMETER keyword. Figure 14-4 on page 14-11 shows a generic cursor deallocation function called **remove_curs()**.

```
void remove_curs(cursname)
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER char *cursname;
EXEC SQL END DECLARE SECTION;
{
   EXEC SQL close :cursname;
   EXEC SQL free :cursname;
}
```

Figure 14-4. A Sample ESQL/C Function That Uses a Dynamic Cursor

Optimizing Cursor Execution

IBM Informix ESQL/C supports the following features that allow you to minimize network traffic when an Informix ESQL/C application fetches rows from a database server:

- Change the size of the fetch and insert buffers
- Automatically free the cursor
- Defer the PREPARE statement until the OPEN statement

Sizing the Cursor Buffer

The *cursor buffer* is the buffer that an Informix ESQL/C application uses to hold the data (except simple large-object data) in a cursor. Informix ESQL/C has the following uses for the cursor buffer:

- The *fetch buffer* holds data from a select or function cursor.

  When the database server returns rows from the active set of a query, Informix ESQL/C stores these rows in the fetch buffer.
- The *insert buffer* holds data for an insert cursor.

  Informix ESQL/C stores the rows to be inserted in the insert buffer then sends this buffer as a whole to the database server for insertion.

With a fetch buffer, the client application performs the following tasks:

1. Sends the size of the buffer to the database server and requests rows when it executes the first FETCH statement.

   The database server sends *as many rows as will fit* in the fetch buffer to the application.
2. Retrieves the rows from the database server and puts them in the fetch buffer.

3. Takes the first row out of the fetch buffer and puts the data in the host variables that the user has provided.

For subsequent FETCH statements, the application checks whether more rows exist in the fetch buffer. If they do, it takes the next row out of the fetch buffer. If no more rows are in the fetch buffer, the application requests more rows from the database server, sending the fetch-buffer size.

The client application uses an insert buffer to perform the following tasks:
1. Put the data from the first PUT statement into the insert buffer.
2. Check whether more room exists in the insert buffer for subsequent PUT statements.

   If more rows can fit, the application puts the next row into the insert buffer. If no more rows can fit into the insert buffer, the application sends the contents of the insert buffer to the database server.

The application continues this procedure until no more rows are put into the insert buffer. It sends the contents of the insert buffer to the database server when:
- The insert buffer is full
- It executes the FLUSH statement on the insert cursor
- It executes the CLOSE statement on the insert cursor

Default Buffer Size: The client application sends the prepared statement that is associated with the cursor to the database server and requests DESCRIBE information about the statement. If the cursor has an associated prepared statement, Informix ESQL/C makes this request when the PREPARE statement executes. If the cursor does not have an associated statement, Informix ESQL/C makes the request when the DECLARE statement executes.

When it receives this request, the database server sends the DESCRIBE information about each column in the projection list to the application. With this information, Informix ESQL/C can determine the size of a row of data. By default, Informix ESQL/C sizes this cursor buffer to hold one row of data. It uses the following algorithm to determine the default size of the cursor buffer:
1. If one row fits in a 4096-byte buffer, the default buffer size is 4096 bytes (4 kilobytes).
2. If the size of one row exceeds 4096 bytes, the default buffer size is the size of that row.

Once it has the buffer size, Informix ESQL/C allocates the cursor buffer.

Changing Size of Cursor Buffer: If your application routinely sends or receives large amounts of data then you can increase performance by increasing the size of the cursor buffer. The cursor buffer is used both as a fetch buffer and as an insert buffer.

Note: Larger cursor buffers are currently only supported in cursors for which the database server is IBM Informix Extended Parallel Server Version 8.40 or higher.

Setting the size of the cursor buffer is an Informix extension to ANSI-standard syntax.

To increase the size of the cursor buffer, you can set either of the following variables in the client environment:

- You can set the BIG_FET_BUF_SIZE environment variable in the environment your client application runs in. Set it to the size you want all cursor buffers in the application to be.

  For example, the following command sets the **BIG_FET_BUF_SIZE** environment variable in the C-shell environment to 40,000 bytes:

  ```
  setenv BIG_FET_BUF_SIZE 40000
  ```

- Inside your application you can set the **BigFetBufSize** global variable to change the size of the cursor buffer *from cursor to cursor*.

  Each time Informix ESQL/C creates a cursor buffer the buffer will be the size currently given in **BigFetBufSize** if it is set.

  The following C code, for example, sets **BigFetBufSize** to 40000 bytes:

  ```
  FetBufSize = 40000;
  ```

  After this line, every cursor that is created will have a size of 40000 bytes until the value of **BigFetBufSize** is changed.

  The **BigFetBufSize** global variable overrides any value in the **BIG_FET_BUF_SIZE** environment variable.

  The **BigFetBufSize** global variable is defined in the Informix ESQL/C **sqlhdr.h** header file which is included automatically in all Informix ESQL/C programs.

The maximum value to which you can set **BIG_FET_BUF_SIZE** or **BigFetBufSize** is 4194303. If you specify a buffer size that is less than the default size or greater than the system maximum, the new size is ignored. If you do not specify a buffer size, the database server uses the default size.

Note: The **BIG_FET_BUF_SIZE** environment variable and the **BigFetBufSize** global variable replace the older **FET_BUF_SIZE** and **FetBufSize** variables. The old variables are still supported for compatibility with earlier versions but only allow a maximum value of 32767. If both the old and new version of a variable are set then whichever is set to the largest value is used.

Automatically Freeing a Cursor

When an IBM Informix ESQL/C application uses a cursor, it usually sends a FREE statement to the database server to deallocate memory assigned to a select cursor once it no longer needs that cursor. Execution of this statement involves a round trip of message requests between the application and the database server. The Automatic-FREE feature (AUTOFREE) reduces the number of round trips by one.

When the AUTOFREE feature is enabled, Informix ESQL/C saves a round trip of message requests because it does not need to execute the FREE statement. When the database server closes a select cursor, it automatically frees the memory that it has allocated for it. Suppose you enable the AUTOFREE feature for the following select cursor:

```
/* Select cursor associated with a SELECT statement */
EXEC SQL declare sel_curs cursor for
   select * from customer;
```

When the database server closes the **sel_curs** cursor, it automatically performs the equivalent of the following FREE statement:

```
FREE sel_curs
```

If the cursor had an associated prepared statement, the database server also frees memory allocated to the prepared statement. Suppose you enable the AUTOFREE feature for the following select cursor:

```
/* Select cursor associated with a prepared statement */
EXEC SQL prepare sel_stmt 'select * from customer';
EXEC SQL declare sel_curs2 cursor for sel_stmt;
```

When the database server closes the **sel_curs2** cursor, it automatically performs the equivalent of the following FREE statements:

```
FREE sel_curs2;
FREE sel_stmt
```

You must enable the AUTOFREE feature *before* you open or reopen the cursor.

Enabling the AUTOFREE Feature: You can enable the AUTOFREE feature for an Informix ESQL/C application in either of the following ways:

- Set the **IFX_AUTOFREE** environment variable to 1 (one).

   When you use the **IFX_AUTOFREE** environment variable to enable the AUTOFREE feature, you automatically free cursor memory when *cursors in any thread* of the program are closed.

- Execute the SQL statement, SET AUTOFREE.

   With the SET AUTOFREE statement, you can enable the AUTOFREE feature for *a particular cursor*. You can also enable or disable the feature in a particular connection or thread.

Warning: Be careful when you enable the AUTOFREE feature in legacy Informix ESQL/C applications. If a legacy application uses the same cursor twice, it generates an error when it tries to open the cursor for the second time. When the AUTOFREE feature is enabled, the database server automatically frees the cursor when it closes it. Therefore, the cursor does not exist when the legacy application attempts to open it a second time, even though the application does not explicitly execute the FREE statement.

For more information about the syntax and use of the SET AUTOFREE statement, see "Using the SET AUTOFREE Statement" on page 14-14. For more information about the **IFX_AUTOFREE** environment variable, see the *IBM Informix Guide to SQL: Reference*.

Using the SET AUTOFREE Statement: You can use the SQL statement, SET AUTOFREE, to enable and disable the AUTOFREE feature. The SET AUTOFREE statement allows you to take the following actions in an Informix ESQL/C program:

- Enable the AUTOFREE feature for *all* cursors:

   ```
   EXEC SQL set autofree;
   EXEC SQL set autofree enabled;
   ```

   These statements are equivalent because the default action of the SET AUTOFREE statement is to enable all cursors.

- Disable the AUTOFREE feature for *all* cursors:

   ```
   EXEC SQL set autofree disabled;
   ```

- Enable the AUTOFREE feature for *a specified* cursor identifier or cursor variable:

   ```
   EXEC SQL set autofree for cursor_id;
   EXEC SQL set autofree for :cursor_var;
   ```

The SET AUTOFREE statement overrides any value of the **IFX_AUTOFREE** environment variable.

The following code fragment uses the FOR clause of the SET AUTOFREE statement to enable the AUTOFREE feature for the **curs1** cursor only. After the database server executes the CLOSE statement for **curs1**, it automatically frees the cursor and the prepared statement. The **curs2** cursor and its prepared statement are not automatically freed.

```
EXEC SQL BEGIN DECLARE SECTION;
   int a_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL create database tst_autofree;
EXEC SQL connect to 'tst_autofree';
EXEC SQL create table tab1 (a_col int);
EXEC SQL insert into tab1 values (1);

/* Declare the curs1 cursor for the slct1 prepared
 * statement */
EXEC SQL prepare slct1 from 'select a_col from tab1';
EXEC SQL declare curs1 cursor for slct1;

/* Enable AUTOFREE for cursor curs1 */
EXEC SQL set autofree for curs1;

/* Open the curs1 cursor and fetch the contents */
EXEC SQL open curs1;
while (SQLCODE == 0)
   {
   EXEC SQL fetch curs1 into :a_value;
   printf("Value is: %d\n", a_value);
   }

/* Once the CLOSE completes, the curs1 cursor is freed and
 * cannot be used again. */
EXEC SQL close curs1;

/* Declare the curs2 cursor for the slct2 prepared
 * statement */
EXEC SQL prepare slct2 from 'select a_col from tab1';
EXEC SQL declare curs2 cursor for slct2;

/* Open the curs2 cursor and fetch the contents */
EXEC SQL open curs2;
while (SQLCODE == 0)
   {
   EXEC SQL fetch curs2 into :a_value;
   printf("Value is: %d\n", a_value);
   }

/* Once this CLOSE completes, the curs2 cursor is still
 * available for use. It has not been automatically freed. */
EXEC SQL close curs2;

/* You must explicitly free the curs2 cursor and slct2
 * prepared statement.  */
EXEC SQL free curs2;
EXEC SQL free slct2;
```

When you use the AUTOFREE feature, make sure you do not cause a prepared statement to become *detached*. This situation can occur if you declare more than one cursor on the same prepared statement. A prepared statement is associated or *attached* to the first cursor that specifies it in a DECLARE statement. If the

AUTOFREE feature is enabled for this cursor, then the database server frees the cursor and its associated prepared statement when it executes the CLOSE statement on the cursor.

A prepared statement becomes detached when either of the following events occur:
- If the prepared statement was not associated with any declared cursor
- If the cursor with the prepared statement was freed but the prepared statement was not.

This second condition can occur if the AUTOFREE feature is not enabled for a cursor and you free only the cursor, not the prepared statement. The prepared statement becomes detached. To reattach the prepared statement, declare a new cursor for the prepared statement. Once a prepared statement was freed, it cannot be used to declare any new cursor.

The following code fragment declares the following cursors on the **slct1** prepared statement:
- The **curs1** cursor, with which the **slct1** prepared statement is first associated
- The **curs2** cursor, which executes **slct1** but with which slct1 is not associated
- The **curs3** cursor, with which **slct1** is associated

The following code fragment shows how a detached prepared statement can occur:

```
/******************************************************************
 * Declare curs1 and curs2. The slct1 prepared statement is       *
 * associated curs1 because curs1 is declared first.             */
EXEC SQL prepare slct1 'select a_col from tab1';
EXEC SQL declare curs1 cursor for slct1;
EXEC SQL declare curs2 cursor for slct1;

/******************************************************************
 * Enable the AUTOFREE feature for curs2                         */
EXEC SQL set autofree for curs2;

/******************************************************************
 * Open the curs1 cursor and fetch the contents                 */
EXEC SQL open curs1;
   {
   EXEC SQL fetch curs1 into :a_value;
   printf("Value is: %d\n", a_value);
   }

EXEC SQL close curs1;

/* Because AUTOFREE is enabled only for the curs2 cursor, this   *
 * CLOSE statement frees neither the curs1 cursor nor the slct1  *
 * prepared statement. The curs1 cursor is still defined so the  *
 * slct1 prepared statement does not become detached.            *
 ******************************************************************/

/******************************************************************
 * Open the curs2 cursor and fetch the contents                 */
EXEC SQL open curs2;
while (SQLCODE == 0)
   {
   EXEC SQL fetch curs2 into :a_value;
   printf("Value is: %d\n", a_value);
   }

EXEC SQL close curs2;
```

```
/* This CLOSE statement frees the curs2 cursor but does not free *
 * slct1 prepared statement because the prepared statement is not*
 * associated with curs2.                                        *
 ***************************************************************/

/***************************************************************
 * Reopen the curs1 cursor. This open is possible because the   *
 * AUTOFREE feature has not been enabled on curs1. Therefore, the*
 * database server did not automatically free curs1 when it closed it.*/
EXEC SQL open curs1;
while (SQLCODE == 0)
    {
    EXEC SQL fetch curs1 into :a_value;
    printf("Value is: %d\n", a_value);
    }

EXEC SQL close curs1;
EXEC SQL free curs1;

/* Explicitly freeing the curs1 cursor, with which the slct1     *
 * statement is associated, causes slct1 to become detached. It  *
 * is no longer associated with a cursor.                        *
 ***************************************************************/

/***************************************************************
 * This DECLARE statement causes the slct1 prepared statement   *
 * to become reassociated with a cursor. Therefore, the slct1   *
 * statement is no longer detached.                             */
EXEC SQL declare curs3 cursor for slct1;
EXEC SQL open curs3;

/* Enable the AUTOFREE feature for curs                         */
EXEC SQL set autofree for curs3;

/* Open the curs3 cursor and fetch the content                  */
EXEC SQL open curs3;
while (SQLCODE == 0)
    {
    EXEC SQL fetch curs3 into :a_value;
    printf("Value is: %d\n", a_value);
    }

EXEC SQL close curs3;

/* Because AUTOFREE is enabled for the curs3 cursor, this CLOSE*
 * statement frees the curs3 cursor and the slct1 PREPARE stmt.*
 ***************************************************************/

/***************************************************************
 * This DECLARE statement would generate a run time error       *
 * because the slct1 prepared statement has been freed.         */
EXEC SQL declare x4 cursor for slct1;
 /***************************************************************/
```

For more information about the syntax and use of the SET AUTOFREE statement, see the *IBM Informix Guide to SQL: Syntax*.

Deferring Execution of the PREPARE Statement

When an IBM Informix ESQL/C application uses a PREPARE/DECLARE/OPEN statement block to execute a cursor, each statement involves a round trip of message requests between the application and the database server. The Deferred-PREPARE feature reduces the number of round trips by one. When the

Deferred-PREPARE feature is enabled, Informix ESQL/C saves a round trip of message requests because it does not need to send a separate command to execute the PREPARE statement. Instead, the database server automatically executes the PREPARE statement when it receives the OPEN statement.

Suppose you enable the Deferred-PREPARE feature for the following select cursor:

```
/* Select cursor associated with a SELECT statement */
EXEC SQL prepare slct_stmt FOR
   'select * from customer';
EXEC SQL declare sel_curs cursor for slct_stmt;
EXEC SQL open sel_curs;
```

The Informix ESQL/C application does not send the PREPARE statement to the database server when it encounters the PREPARE before the DECLARE statement. Instead, it sends the PREPARE and the OPEN to the database server together when it executes the OPEN statement.

You can use the Deferred-PREPARE feature in Informix ESQL/C applications that contain dynamic SQL statements that use statement blocks of PREPARE, DECLARE, and OPEN to execute the following statements:

- SELECT statements (select cursors)
- EXECUTE FUNCTION statements (function cursors)
- INSERT statement (insert cursors)

For example, the Deferred-PREPARE feature reduces network round trips for the following select cursor:

```
/* Valid select cursor for Deferred-PREPARE optimization */
EXEC SQL prepare sel_stmt 'select * from customer';
EXEC SQL declare sel_curs cursor for sel_stmt;
EXEC SQL open sel_curs;
```

Restrictions on Deferred-PREPARE: When you enable the deferred-PREPARE feature, the client application does not send PREPARE statements to the database server when it encounters them. The database server receives a description of the prepared statement when it executes the OPEN statement.

The database server generates an error if you execute a DESCRIBE statement on a prepared statement before the first OPEN of the cursor. The error occurs because the database server has not executed the PREPARE statement that the DESCRIBE statement specifies. When the deferred-PREPARE feature is enabled, you must execute the DESCRIBE statement after the first OPEN of a cursor.

Important: The deferred-PREPARE feature eliminates execution of the PREPARE statement as a separate step. Therefore, the application does not receive any error conditions that might exist in the prepared statement until after the initial OPEN.

For more information, see "Using the SET DEFERRED_PREPARE Statement" on page 14-19.

Enabling the Deferred-PREPARE Feature: You can enable the Deferred-PREPARE feature for an Informix ESQL/C application in either of the following ways:

- Set the **IFX_DEFERRED_PREPARE** environment variable to 1 (one).

   When you use the **IFX_DEFERRED_PREPARE** environment variable to enable the Deferred-PREPARE feature, you automatically defer execution of the

PREPARE statement until just before the OPEN statement executes for *every* PREPARE *statement* in any thread of the application.

The default value of the **IFX_DEFERRED_PREPARE** environment variable is 0 (zero). If you set this environment variable from the shell, make sure you set it before you start the Informix ESQL/C application.

- Execute the SQL statement, SET DEFERRED_PREPARE.

  With the SET DEFERRED_PREPARE statement, you can enable the Deferred-PREPARE feature for a particular PREPARE statement. You can also enable or disable the feature in a particular connection or thread.

For more information about the syntax and use of the SET DEFERRED PREPARE statement, see "Using the SET DEFERRED_PREPARE Statement." For more information about the **IFX_DEFERRED_PREPARE** environment variable, see the *IBM Informix Guide to SQL: Reference*.

Using the SET DEFERRED_PREPARE Statement: In an Informix ESQL/C application you can use the SQL statement, SET DEFERRED_PREPARE, to enable and disable the Deferred-PREPARE feature. The SET DEFERRED_PREPARE statement allows you to take the following actions in an Informix ESQL/C program:

- Enable the Deferred-PREPARE feature:

  ```
  EXEC SQL set deferred_prepare;
  EXEC SQL set deferred_prepare enabled;
  ```

- Disable the Deferred-PREPARE feature:

  ```
  EXEC SQL set deferred_prepare disabled;
  ```

The SET DEFERRED_PREPARE statement overrides any value of the **IFX_DEFERRED_PREPARE** environment variable.

The following code fragment shows how to enable the Deferred-PREPARE feature for the **ins_curs** insert cursor:

```
EXEC SQL BEGIN DECLARE SECTION;
   int a;
EXEC SQL END DECLARE SECTION;

EXEC SQL create database test;
EXEC SQL create table table_x (col1 integer);

/************************************
 * Enable Deferred-Prepare feature
 ************************************/
EXEC SQL set deferred_prepare enabled;

/************************************
 * Prepare an INSERT statement
 ************************************/
EXEC SQL prepare ins_stmt from
   'insert into table_x values(?)';

/************************************
 * Declare the insert cursor for the
 * prepared INSERT.
 ************************************/
EXEC SQL declare ins_curs cursor for ins_stmt;
/**********************************************************
 * OPEN the insert cursor. Because the Deferred-PREPARE feature
 * is enabled, the PREPARE is executed at this time
 **********************************************************/
EXEC SQL open ins_curs;
```

```
a = 2;
while (a<100)
    {
    EXEC SQL put ins_curs from :a;
    a++;
    }
```

To execute a DESCRIBE statement on a prepared statement, you must execute the
DESCRIBE *after* the initial OPEN statement for the cursor. In the following code
fragment the first DESCRIBE statement fails because it executes before the first
OPEN statement on the cursor. The second DESCRIBE statement succeeds because
it follows an OPEN statement.

```
EXEC SQL BEGIN DECLARE SECTION;
    int a, a_type;
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate descriptor 'desc';
EXEC SQL create database test;
EXEC SQL create table table_x (col1 integer);

/**********************************************
 * Enable Deferred-Prepare feature
 ***************************************************/
EXEC SQL set deferred_prepare enabled;

/**********************************************
 * Prepare an INSERT statement
 ***************************************************/
EXEC SQL prepare ins_stmt from 'insert into table_x values (?)';

/********************************************************************
 * The DESCRIBE results in an error, because the description of the
 * statement is not determined until after the OPEN. The OPEN is what
 * actually sends the PREPARE statement to the database server and
 * requests a description for it.
 ********************************************************************/
EXEC SQL describe ins_stmt using sql descriptor 'desc'; /* fails */
if (SQLCODE)
    printf("DESCRIBE : SQLCODE is %d\n", SQLCODE);

/********************************************************************
 * Now DECLARE a cursor for the PREPARE statement and OPEN it.
 ********************************************************************/
EXEC SQL declare ins_cursor cursor for ins_stmt;
EXEC SQL open ins_cursor;

/********************************************************************
 * Now the DESCRIBE returns the information about the columns to the
 * system-descriptor area.
 ********************************************************************/
EXEC SQL describe ins_stmt using sql descriptor 'desc'; /* succeeds */
if (SQLCODE)
    printf("DESCRIBE : SQLCODE is %d\n", SQLCODE);
a = 2;
a_type = SQLINT;
while (a<100)
    {
    EXEC SQL set descriptor 'desc' values 1
        type = :a_type, data = :a;
    EXEC SQL put ins_curs using sql descriptor 'desc';
    a++;
    }
```

The collect.ec Program

The **collect.ec** example program, which follows, illustrates the use of collection variables to access LIST, SET, and MULTISET columns. The SELECT statement is considered *static* because the columns that it accesses are determined when the program is written.

```
/*
**
** Sample use of collections in ESQL/C.
**
** Statically determined LIST, SET, and MULTISET collection types.
*/

#include <stdio.h>

static void print_collection(
const char *tag,
EXEC SQL BEGIN DECLARE SECTION;
parameter client collection c
EXEC SQL END DECLARE SECTION;
)
{
    EXEC SQL BEGIN DECLARE SECTION;
    int4 value;
    EXEC SQL END DECLARE SECTION;
    mint item = 0;

    EXEC SQL WHENEVER ERROR STOP;
    printf("COLLECTION: %s\n", tag);
    EXEC SQL DECLARE c_collection CURSOR FOR
        SELECT * FROM TABLE(:c);
    EXEC SQL OPEN c_collection;
    while (sqlca.sqlcode == 0)
    {
        EXEC SQL FETCH c_collection INTO :value;
        if (sqlca.sqlcode != 0)
            break;
        printf("\tItem %d, value = %d\n", ++item, value);
    }
    EXEC SQL CLOSE c_collection;
    EXEC SQL FREE c_collection;
}

mint main(int argc, char **argv)
{
    EXEC SQL BEGIN DECLARE SECTION;
    client collection list     (integer not null) lc1;
    client collection set      (integer not null) sc1;
    client collection multiset (integer not null) mc1;
    char *dbase = "stores7";
    mint seq;
    char *stmt1 =
        "INSERT INTO t_collections VALUES(0, "
        "'LIST{-1,0,-2,3,0,0,32767,249}', 'SET{-1,0,-2,3}', "
        "'MULTISET{-1,0,0,-2,3,0}') ";
    EXEC SQL END DECLARE SECTION;

    if (argc > 1)
        dbase = argv[1];
    EXEC SQL WHENEVER ERROR STOP;
    printf("Connect to %s\n", dbase);
    EXEC SQL connect to :dbase;

    EXEC SQL CREATE TEMP TABLE t_collections
    (
        seq serial not null,
```

```
        l1 list    (integer  not null),
        s1 set     (integer  not null),
        m1 multiset(integer  not null)
    );
EXEC SQL EXECUTE IMMEDIATE :stmt1;

EXEC SQL ALLOCATE COLLECTION :lc1;
EXEC SQL ALLOCATE COLLECTION :mc1;
EXEC SQL ALLOCATE COLLECTION :sc1;

EXEC SQL DECLARE c_collect CURSOR FOR
    SELECT seq, l1, s1, m1 FROM t_collections;
EXEC SQL OPEN c_collect;

EXEC SQL FETCH c_collect INTO :seq, :lc1, :sc1, :mc1;
EXEC SQL CLOSE c_collect;
EXEC SQL FREE c_collect;

print_collection("list/integer", lc1);
print_collection("set/integer", sc1);
print_collection("multiset/integer", mc1);

EXEC SQL DEALLOCATE COLLECTION :lc1;
EXEC SQL DEALLOCATE COLLECTION :mc1;
EXEC SQL DEALLOCATE COLLECTION :sc1;

puts("OK");
return 0;
}
```

Optimizing OPEN, FETCH, and CLOSE

When an Informix ESQL/C application uses DECLARE and OPEN statements to
execute a cursor, each statement involves a round trip of message requests between
the application and the database server. The optimize-OPEN-FETCH-CLOSE
feature (OPTOFC) reduces the number of round trips by two, as follows:

- Informix ESQL/C saves one round trip because it does not send the OPEN
  statement as a separate command.

  When Informix ESQL/C executes the OPEN statement, it does not open the
  cursor. Instead, it saves any input value that was supplied in the USING clause
  of the OPEN statement. When Informix ESQL/C executes the initial FETCH
  statement, it sends this input value along with the FETCH statement. The
  database server opens the cursor and returns the first value in this cursor.

- Informix ESQL/C saves a second round trip because it does not send the CLOSE
  statement as a separate command.

  When the database server reaches the last value of an open cursor, it
  automatically closes the cursor after it sends the last value to the client
  application. Therefore, Informix ESQL/C does not need to send the CLOSE
  statement to the database server.

Important: Informix ESQL/C does not send the CLOSE statement to the database
server. However, if you include the CLOSE statement, no error is
generated.

Restrictions on OPTOFC

With the OPTOFC feature enabled, the following restrictions exist:

- You can only use the OPTOFC feature on select cursors whose SELECT
  statement was prepared. For example, the OPTOFC feature reduces network
  round trips for the following select cursor:

```
/* Valid select cursor for OPTOFC optimization */
EXEC SQL prepare sel_stmt 'select * from customer';
EXEC SQL declare sel_curs cursor for sel_stmt;
```

- The OPTOFC feature eliminates execution of the OPEN statement as a separate step. Therefore, any error conditions that opening the cursor might generate are not returned until after the initial FETCH.
- Static cursors are not freed when they are closed.

  With the OPTOFC feature enabled, neither static nor dynamic cursors are freed when they are closed. Because Informix ESQL/C does not actually send the CLOSE statement to the database server, a cursor is not implicitly freed. A subsequent OPEN and FETCH on a cursor actually opens the same cursor. Only at this time would the database server notice if the table was modified (if it was dropped, altered, or renamed), in which case it generates an error (-710).

  With the OPTOFC feature disabled, a static cursor is freed when it is closed. When ESQL/C reaches a CLOSE statement for a static cursor, it actually sends a message to close the cursor and free memory associated with this cursor. However, dynamic cursors are not implicitly freed when they are closed.
- The GET DIAGNOSTICS statement does not work for SQL statements that are delayed on the way to the database server. For example, in the following sequence of SQL statements, GET DIAGNOSTICS returns 0, indicating success, even though the OPEN is delayed until the first fetch:

```
EXEC SQL declare curs1 ...
EXEC SQL open curs1
EXEC SQL get diagnostic
EXEC SQL fetch curs1
```

Enabling the OPTOFC Feature

The **OPTOFC** environment variable enables the OPTOFC feature. You can assign the following values to the **OPTOFC** environment variable.

Value	Description
1	This value enables the **OPTOFC** feature. When you specify this value, you enable the **OPTOFC** feature for every cursor in every thread of the application.
0	This value disables the **OPTOFC** feature for all threads of the application.

The default value of the **OPTOFC** environment variable is 0 (zero). If you set this environment variable from the shell, make sure you set it before you start the ESQL/C application.

On UNIX operating systems, you can set **OPTOFC** in the application with the **putenv()** system call (as long as your system supports the **putenv()** function). For example, the following call to **putenv()** enables the OPTOFC feature:

```
putenv("OPTOFC=1");
```

In Windows environments, you can use the **ifx_putenv()** function.

With **putenv()** or **ifx_putenv()**, you can activate or deactivate the OPTOFC feature for each connection or within each thread. You must call the **putenv()** or **ifx_putenv()** function before you establish a connection.

Important: Informix utilities do not support the **IFX_AUTOFREE**, **OPTOFC**, and **IFX_DEFERRED_PREPARE** environment variables. Use these environment variables only with IBM Informix ESQL/C client applications.

Using OPTOFC and Deferred-PREPARE Together

To achieve the most optimized number of messages between the client application and the database server, use the Optimize OPEN, FETCH, CLOSE feature and the Deferred-PREPARE feature together. However, keep the following requirements in mind when you use these two optimization features together:

- If syntax errors exist in the statement text, the database server does not return the error to the application until it executes the FETCH.

  Informix ESQL/C does not send the PREPARE, DECLARE, and OPEN statements to the database server until it executes the FETCH statement. Therefore, any errors that any of these statements generate are not available until the database server executes the FETCH statement.

- You must use a special case of the GET DESCRIPTOR statement to obtain DESCRIBE information for a prepared statement.

  Typical use of the DESCRIBE statement is to execute it after the PREPARE to determine information about the prepared statement. However, with both the OPTOFC and Deferred-PREPARE features enabled, Informix ESQL/C does not send the DESCRIBE statement to the database until it reaches the FETCH statement. To allow you to obtain information about the prepared statement, Informix ESQL/C executes a statement similar to the SET DESCRIPTOR statement to obtain data type, length, and other system-descriptor fields for the prepared statement. You can then use the GET DESCRIPTOR statement after the FETCH to obtain this information.

  Also, Informix ESQL/C can only perform data conversions on the host variables in the GET DESCRIPTOR statement when the data types are built-in data types. For opaque data types and complex data types (collections and row types), the database server always returns the data to the client application in its native format. You can then perform data conversions on this data after the GET DESCRIPTOR statement.

  For example, the database server returns data from an opaque-type column in its internal (binary) format. Therefore, your Informix ESQL/C program must put column data into a **var binary** (or **fixed binary**) host variable when it executes the GET DESCRIPTOR statement. The **var binary** and **fixed binary** data types hold opaque-type data in its internal format. You cannot use an **lvarchar** host variable to hold the data because Informix ESQL/C cannot convert the opaque-type data from its internal format (which it receives from the database server) to its external (**lvarchar**) format.

- The FetArrSize feature does not work when both the Deferred-PREPARE and OPTOFC features are enabled. When these two features are enabled, Informix ESQL/C does not know the size of a row until after the FETCH completes. By this time, it is too late for the fetch buffer to be adjusted with the FetArrSize value.

Tip: To obtain the maximum optimization, use the OPTOFC, deferred-PREPARE and AUTOFREE features together. For more information about the AUTOFREE feature, see "Automatically Freeing a Cursor" on page 14-13.

SQL Statements That Are Known at Compile Time

The simplest type of dynamic SQL to execute is one for which you know both of the following items:

- The structure of the SQL statement to be executed, including information like the statement type and the syntax of the statement
- The number and data types of any data that passes between the Informix ESQL/C program and the database server

The following sections describe how to execute both non-SELECT statements and SELECT statements whose structure and data are known at compile time.

Executing Non-SELECT Statements

In this chapter, the term *non-SELECT statement* refers to any SQL statement that can be prepared, except SELECT and EXECUTE FUNCTION. This term includes the EXECUTE PROCEDURE statement.

Important: The INSERT statement is an exception to the rules for non-SELECT statements. If the INSERT inserts a single row, use PREPARE and EXECUTE to execute it. However, if the INSERT is associated with an insert cursor, you must declare the insert cursor.

For more information, see "Sending More Than One Row" on page 14-10.

For a list of SQL statements that cannot be prepared, see the entry for the PREPARE statement in the *IBM Informix Guide to SQL: Syntax*.

You can execute a non-SELECT statement in the following ways:

- If the statement is to be executed more than once, use the PREPARE and EXECUTE statements.
- If the statement is to be executed only once, use the EXECUTE IMMEDIATE statement. This statement does have some restrictions on the statements it can execute.

Using PREPARE and EXECUTE

The PREPARE and EXECUTE statements allow you to separate the execution of a non-SELECT statement into two steps:

1. PREPARE sends the statement string to the database server, which parses the statement and assigns it a statement identifier.
2. EXECUTE executes the prepared statement indicated by a statement identifier.

This two-step process is useful for statements that need to be executed more than once. You reduce the traffic between the client application and the database server when you parse the statement only once.

For example, you can write a general-purpose deletion program that works on any table. This 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 C variables such as **tabname** and **search_condition**. The **tabname** and **search_condition** variables do not need to be host variables because they do not appear in the actual SQL statement.

2. Create a text string by concatenating the following four components: DELETE FROM, **tabname**, WHERE, and **search_condition**. In this example, the string is in a host variable called **stmt_buf**:

```
sprintf(stmt_buf, "DELETE FROM %s WHERE %s",
        tabname, search_condition);
```

3. Prepare the entire statement. The following PREPARE statement operates on the string in **stmt_buf** and creates a statement identifier called **d_id**:

```
EXEC SQL prepare d_id from :stmt_buf;
```

4. Execute the statement. The following EXECUTE statement executes the DELETE:

```
EXEC SQL execute d_id;
```

5. If you do not need to execute the statement again, free the resources used by the statement identifier structure. This example would use the following FREE statement:

```
EXEC SQL free d_id;
```

If the non-SELECT statement contains input parameters, you must use the USING clause of the EXECUTE statement. For more information, see "Using an EXECUTE USING Statement" on page 14-34.

The EXECUTE statement is generally used to execute non-SELECT statements. You can use EXECUTE with the INTO clause for a SELECT or an EXECUTE FUNCTION statement as long as these statements return only *one* group of values (one row). For more information, see "Using PREPARE and EXECUTE INTO" on page 14-27. However, do *not* use the EXECUTE statement for:

- an INSERT...VALUES statement that is associated with an insert cursor.
  See "Sending More Than One Row" on page 14-10.
- an EXECUTE FUNCTION statement for a *cursor function* (a user-defined function that returns more than one group of values).
  See "Determining Return Values Dynamically" on page 15-18.

Using EXECUTE IMMEDIATE

Rather than prepare the statement and then execute it, you can prepare and execute the statement in the same step with the EXECUTE IMMEDIATE statement. The EXECUTE IMMEDIATE statement also frees statement-identifier resources upon completion.

For example, for the DELETE statement used in the previous section, you can replace the PREPARE-EXECUTE statement sequence with the following statement:

```
EXEC SQL execute immediate :stmt_buf;
```

You cannot use EXECUTE IMMEDIATE if the statement string contains input parameters. (For information, see "Using an EXECUTE USING Statement" on page 14-34.) The SQL statements also have restrictions that you can execute with EXECUTE IMMEDIATE. For a complete list of SQL statement restrictions, see the entry for EXECUTE IMMEDIATE in the *IBM Informix Guide to SQL: Syntax*.

Executing SELECT Statements

You can execute a SELECT statement in the following two ways:

- If the SELECT statement returns *only one row*, use PREPARE and EXECUTE INTO. This type of SELECT is often called a *singleton* SELECT.
- If the SELECT statement returns *more than one row*, you must use cursor-management statements.

Using PREPARE and EXECUTE INTO

The only prepared SELECT statement that you can execute with the EXECUTE statement is a singleton SELECT. Your Informix ESQL/C program must take the following actions:

1. Declare host variables to receive the values that the database server returns.

   For a prepared SELECT statement, these values are the select-list columns.

2. Assemble and prepare the statement.

   A prepared SELECT statement can contain input parameters in the WHERE clause. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.

3. Execute the prepared selection with the EXECUTE...INTO statement, with the host variables after the INTO keyword.

   If the SELECT statement contains input parameters, include the USING clause of EXECUTE.

Tip: To execute a singleton SELECT, using EXECUTE...INTO is usually more efficient than using the DECLARE, OPEN, and FETCH statements.

With the INTO clause of the EXECUTE statement, you can still use the following features:

- You can associate indicator variables with the host variables that receive the select-list column values.

  Use the INDICATOR keyword followed by the name of the indicator host variable, as follows:

```
EXEC SQL prepare sel1 from
  'select fname, lname from customer where customer_num = 123';
EXEC SQL execute sel1 into :fname INDICATOR :fname_ind,
  :lname INDICATOR :lname_ind;
```

- You can specify input parameter values.

  Include the USING clause of EXECUTE, as follows:

```
EXEC SQL prepare sel2 from
  'select fname, lname from customer where customer_num = ?';
EXEC SQL execute sel2 into :fname, :lname using :cust_num;
```

  For more information, see "Using an EXECUTE USING Statement" on page 14-34.

Warning: When you use the EXECUTE INTO statement, make sure that the SELECT statement is a singleton SELECT. If the SELECT returns more than one row, you receive a runtime error. An error is also generated if you attempt to execute a prepared statement that was declared (with DECLARE).

You are not required to prepare a singleton SELECT. If you do not need the benefits of a prepared statement, you can embed a singleton SELECT statement directly in your Informix ESQL/C program, as shown in the following example:

```
EXEC SQL select order_date from orders where order_num = 1004;
```

Figure 14-5 shows how to execute the **items_pct()** SPL function (which Figure 16-1 on page 16-15 shows). Because this function returns a single **decimal** value, the EXECUTE...INTO statement can execute it.

```
EXEC SQL prepare exfunc_id from
   'execute function items_pct(\"HSK\")';
EXEC SQL execute exfunc_id into :manuf_dec;
```

Figure 14-5. Executing the items_pct SPL Function

You can use host variables for routine arguments but not the routine name. For example, if the **manu_code** variable holds the value "HSK", the following EXECUTE statement replaces the input parameter in the prepared statement to perform the same task as the EXECUTE in Figure 14-5 on page 14-28.

```
EXEC SQL prepare exfunc_id from
   'execute function items_pict(?)';
EXEC SQL execute exfunc_id into :manuf_dec using :manu_code;
```

If you do not know the number or data types of the select-list columns or function return values, you must use a dynamic-management structure instead of host variables with the EXECUTE...INTO statement. The dynamic-management structure defines the select-list columns at runtime. For more information, see "Handling an Unknown Select List" on page 15-16.

Using a Select Cursor

To execute a SELECT statement that returns more than one row, you must declare a *select cursor*. The select cursor enables the Informix ESQL/C application to handle multiple rows that a query returns.

Your Informix ESQL/C program must take the following actions to use a select cursor:

1. Declare host variables to receive the values that the database server returns.

   For a prepared SELECT statement, these values are the select-list columns. For a prepared EXECUTE FUNCTION statement, these values are the return values of the user-defined function.

2. Assemble and prepare the statement.

   A prepared SELECT statement can contain input parameters in the WHERE clause. A prepared EXECUTE FUNCTION statement can contain input parameters as function arguments. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.

3. Declare the select cursor.

   The DECLARE statement associates the prepared SELECT statement with the select cursor.

4. Execute the query.

   The OPEN statement sends any input parameters that its USING clause specifies to the database server and tells the database server to execute the SELECT statement.

5. Retrieve the rows of values from the select cursor.

   The FETCH statement retrieves one row of data that matches the query criteria.

For more information about cursors and cursor-management statements, see "Using a Database Cursor" on page 14-8.

Warning: Do not use the INTO clause in both a SELECT statement that is associated with a cursor and in a FETCH statement that retrieves data from the cursor. Neither the Informix ESQL/C preprocessor nor the

executable program generate an error for this condition. Using the INTO clause in both statements, however, can generate unexpected results.

The lvarptr.ec Program

The lvarptr.ec example program, which follows, uses lvarchar pointers

```
/*
**
** Sample use of LVARCHAR to fetch collections in ESQL/C.
**
** Statically determined collection types.
*/

#include <stdio.h>

static void print_lvarchar_ptr(
const char *tag,
EXEC SQL BEGIN DECLARE SECTION;
parameter lvarchar **lv
EXEC SQL END DECLARE SECTION;
)
{
    char *data;

    data = ifx_var_getdata(lv);
    if (data == 0)
        data = "<<NO DATA>>";
    printf("%s: %s\n", tag, data);
}

static void process_stmt(char *stmt)
{
    EXEC SQL BEGIN DECLARE SECTION;
    lvarchar *lv1;
    lvarchar *lv2;
    lvarchar *lv3;
    mint seq;
    char *stmt1 = stmt;
    EXEC SQL END DECLARE SECTION;

    printf("SQL: %s\n", stmt);

    EXEC SQL WHENEVER ERROR STOP;
    EXEC SQL PREPARE p_collect FROM :stmt1;
    EXEC SQL DECLARE c_collect CURSOR FOR p_collect;
    EXEC SQL OPEN c_collect;

    ifx_var_flag(&lv1, 1);
    ifx_var_flag(&lv2, 1);
    ifx_var_flag(&lv3, 1);

    while (sqlca.sqlcode == 0)
    {
        EXEC SQL FETCH c_collect INTO :seq, :lv1, :lv2, :lv3;
        if (sqlca.sqlcode == 0)
        {
            printf("Sequence: %d\n", seq);
            print_lvarchar_ptr("LVARCHAR 1", &lv1);
            print_lvarchar_ptr("LVARCHAR 2", &lv2);
            print_lvarchar_ptr("LVARCHAR 3", &lv3);
            ifx_var_dealloc(&lv1);
            ifx_var_dealloc(&lv2);
            ifx_var_dealloc(&lv3);
```

```
            }
        }

        EXEC SQL CLOSE c_collect;
        EXEC SQL FREE c_collect;
        EXEC SQL FREE p_collect;
}

mint main(int argc, char **argv)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char *dbase = "stores7";
    char *stmt1 =
        "INSERT INTO t_collections VALUES(0, "
        "'LIST{-1,0,-2,3,0,0,32767,249}', 'SET{-1,0,-2,3}', "
        "'MULTISET{-1,0,0,-2,3,0}') ";
    char *data;
    EXEC SQL END DECLARE SECTION;

    if (argc > 1)
        dbase = argv[1];
    EXEC SQL WHENEVER ERROR STOP;
    printf("Connect to %s\n", dbase);
    EXEC SQL CONNECT TO :dbase;

    EXEC SQL CREATE TEMP TABLE t_collections
    (
        seq serial not null,
        l1 list    (integer  not null),
        s1 set     (integer  not null),
        m1 multiset(integer  not null)
    );

    EXEC SQL EXECUTE IMMEDIATE :stmt1;
    EXEC SQL EXECUTE IMMEDIATE :stmt1;
    EXEC SQL EXECUTE IMMEDIATE :stmt1;

    process_stmt("SELECT seq, l1, s1, m1 FROM t_collections");

    puts("OK");
    return 0;
}
```

Executing User-Defined Routines in IBM Informix

In IBM Informix, a *user-defined routine* is a collection of statements that performs a user-defined task. A *procedure* is a routine that can accept arguments but does not return any values. A *function* is a routine that can accept arguments and returns values. The following table summarizes the SQL statements for user-defined routines.

	SQL Statement for User Defined Routines	
Task	Procedure	Function
Create and register a routine	CREATE PROCEDURE	CREATE FUNCTION
Execute a routine	EXECUTE PROCEDURE	EXECUTE FUNCTION
Drop a routine	DROP PROCEDURE	DROP FUNCTION

For more information about these statements, see the *IBM Informix Guide to SQL: Syntax*.

IBM Informix supports several languages for user-defined routines:

- External routines are written in external languages such as C.

  An external function can return one value while an external procedure does not return a value. For information about how to write an external routine in C, see *IBM Informix User-Defined Routines and Data Types Developer's Guide* .

- SPL routines are written in Stored Procedure Language (SPL).

  An SPL function can return one or more values while an SPL procedure does not return any values. For information about how to write a stored routine, see the *IBM Informix Guide to SQL: Tutorial*.

Tip: In earlier versions of IBM Informix products, the term "*stored procedure*" was used for both SPL procedures and SPL functions. That is, a stored procedure can include the RETURN statement to return values. For compatibility with earlier products, IBM Informix continues to support the execution of SPL functions with the EXECUTE PROCEDURE statement. However, for new SPL routines, it is recommended that you use EXECUTE PROCEDURE only for procedures and EXECUTE FUNCTION only for functions.

A user-defined routine can use input parameters for its arguments. However, it cannot use an input parameter for its routine name. For more information, see "Execute statements with input parameters" on page 14-33.

A User-Defined Procedure

If you know the name of the user-defined procedure (external or SPL) at compile time, execute the user-defined procedure with the EXECUTE PROCEDURE statement. The following EXECUTE PROCEDURE statement executes a user-defined procedure called **revise_stats()**:

```
EXEC SQL execute procedure revise_stats("customer");
```

For more information about the syntax of the EXECUTE PROCEDURE statement, see the *IBM Informix Guide to SQL: Syntax*.

If you do not know the name of the user-defined procedure until runtime, you must dynamically execute the procedure. To dynamically execute a user-defined procedure, you can use:

- the PREPARE and EXECUTE statements (14-25)
- the EXECUTE IMMEDIATE statement (14-26)

A User-Defined Function

If you know the name of the user-defined function at compile time, execute the user-defined function (external or SPL) with the EXECUTE FUNCTION statement. In the INTO clause of EXECUTE FUNCTION, you list the host variables that hold the return value(s). The following EXECUTE FUNCTION statement executes a user-defined function called **items_pct()** (which Figure 16-1 on page 16-15 defines):

```
EXEC SQL execute function items_pct(\"HSK\")
   into :manuf_percent;
```

For more information about the syntax of the EXECUTE FUNCTION statement, see the *IBM Informix Guide to SQL: Syntax*.

If you do not know the name of the user-defined function until runtime, you must dynamically execute the function. Dynamic execution of a user-defined function is a similar dynamic execution of a SELECT statement (15-16). Both the SELECT and the user-defined function return values to the IBM Informix ESQL/C program.

Execute a user-defined function with the EXECUTE FUNCTION statement. You can execute an EXECUTE FUNCTION statement in the following two ways:

- If the user-defined function returns only one row, use PREPARE and EXECUTE INTO to execute the EXECUTE FUNCTION statement. This type of user-defined function is often called a *noncursor function*.
- If the user-defined function returns more than one row, you must declare a function cursor to execute the EXECUTE FUNCTION statement.

  This type of user-defined function is often called a *cursor function*. A cursor function that is written in SPL (an SPL function) has the WITH RESUME clause in its RETURN statement. A cursor function that is written in an external language such as C is an iterator function.

Tip: If you do *not* know the data type of the return value, you must use a dynamic-management structure to hold the value. For more information, see "Determining Return Values Dynamically" on page 15-18.

A Noncursor Function: You can also use the PREPARE and EXECUTE statement to execute a *noncursor user-defined function*. A noncursor function returns only one row of values.

Your Informix ESQL/C program must take the following actions:

1. Declare host variables to receive the values that the database server returns.

   For a prepared EXECUTE FUNCTION statement, these values are the return values of the user-defined function.

2. Assemble and prepare the statement.

   A prepared EXECUTE FUNCTION statement can contain input parameters as function arguments. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.

3. Execute the prepared user-defined function with the EXECUTE...INTO statement, with the host variables after the INTO keyword.

   If the EXECUTE FUNCTION contains input parameters, include the USING clause of EXECUTE.

Important: To execute a noncursor function, EXECUTE...INTO is usually more efficient than the DECLARE, OPEN, and FETCH statements. However, you often do not know the number of returned rows. When you do not use a cursor to execute a cursor function that returns multiple rows, Informix ESQL/C generates a runtime error. Therefore, it is a good practice to always associate a user-defined function with a cursor.

For more information, see "A cursor function" on page 14-33.

Most external functions can return *only one row* of data and only a single value. For example, the following code fragment executes an external function called **stnd_dev()**:

```
strcpy(func_name, "stnd_dev(ship_date)");
sprintf(exfunc_stmt, "%s %s %s",
        "execute function",
        func_name);
EXEC SQL prepare exfunc_id from :exfunc_stmt;
EXEC SQL execute exfunc_id into :ret_val;
```

To return more than one value, the external function must return a complex data type, such as a collection or a row type. (For more information about these data types, see Chapter 9, "Working with complex data types," on page 9-1.)

An SPL function can return one or more values. If the RETURN statement of the SPL function does not contain the WITH RESUME keywords, then the function returns only one row. To execute the SPL function dynamically, prepare the EXECUTE FUNCTION and execute it with the EXECUTE...INTO statement.

For more information, see "Using PREPARE and EXECUTE INTO" on page 14-27.

A cursor function: To execute an EXECUTE FUNCTION statement whose user-defined function returns more than one row, you must declare a *function cursor*. The function cursor enables the Informix ESQL/C application to handle the multiple rows that a user-defined function returns. Your Informix ESQL/C program must take the following actions to use a function cursor:

1. Declare host variables to receive the values that the user-defined function returns.
2. Assemble and prepare the statement.

   A prepared EXECUTE FUNCTION statement can contain input parameters as function arguments. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.
3. Declare the function cursor.

   The DECLARE statement associates the prepared EXECUTE FUNCTION statement with the function cursor.
4. Execute the user-defined function.

   The OPEN statement sends any input parameters that its USING clause specifies to the database server and tells the database server to execute the EXECUTE FUNCTION statement.
5. Retrieve the rows of values from the function cursor.

   The FETCH statement retrieves one row of values that the user-defined function returns.

Only an external function that is an iterator function can return more than one row of data. For information about how to write an iterator function, see the *IBM Informix DataBlade API Programmer's Guide*.

If the RETURN statement of the SPL function contains the WITH RESUME keywords, then the function can return more than one row. You must associate such an SPL function with a function cursor. To execute the SPL function dynamically, associate the EXECUTE FUNCTION statement with a cursor, use the OPEN statement to execute the function, and use the FETCH...INTO statement to retrieve the rows from the cursor into host variables.

For more information about cursors and cursor-management statements, see "Using a Database Cursor" on page 14-8.

Execute statements with input parameters

An *input parameter* is a placeholder in an SQL statement that indicates that the actual value is provided at runtime. You cannot list a host-variable name in the text of a dynamic SQL statement because the database server knows nothing about variables declared in the application. Instead, you can indicate an input parameter with a question mark (?), which serves as a placeholder, anywhere within a

statement where an expression is valid. You cannot use an input parameter to represent an identifier such as a database name, a table name, or a column name.

An SQL statement that contains input parameters is called a *parameterized statement*. For a parameterized SQL statement, your program must provide the following information to the database server about its input parameters:

- Your program must use a question mark (?) as a placeholder in the text of the statement to indicate where to expect an input parameter. For example, the following DELETE statement contains two input parameters:

```
EXEC SQL prepare dlt_stmt from
    'delete from orders where customer_num = ? \
    and order_date > ?';
```

  The first input parameter is defined for the value of the **customer_num** column and the second for the value of the **order_date** column.

- Your program must specify the value for the input parameter when the statement executes with the USING clause. To execute the DELETE statement in the previous step, you can use the following statement:

```
EXEC SQL execute dlt_stmt using :cust_num, :ord_date;
```

The statement that you use to provide an input parameter with a value at runtime depends on the type of SQL statement that you execute, as follows:

- For a non-SELECT statement (such as UPDATE, INSERT, DELETE, or EXECUTE PROCEDURE) with input parameters, the EXECUTE...USING statement executes the statement and provides input parameter values.
- For a SELECT statement associated with a cursor or for a cursor function (EXECUTE FUNCTION), the OPEN...USING statement executes the statement and provides input parameter values.
- For a singleton SELECT statement or for a noncursor function (EXECUTE FUNCTION), the EXECUTE...INTO...USING statement executes the statement and provides input parameter values.

When the statement executes, you can list host variables or literal values to substitute for each input parameter in the USING clause. The values must be compatible in number and data type with the associated input parameters. A host variable must also be large enough to hold the data.

Important: To use host variables with the USING clause, you must know the number of parameters in the SQL statement and their data types. If you do not know the number and data types of the input parameters at runtime, you must use a dynamic-management structure with the USING clause. For more information, see "Determining Unknown Input Parameters" on page 15-17.

The following sections show how to use host variables with the OPEN...USING and the EXECUTE...USING statements.

Using an EXECUTE USING Statement

You can execute a *parameterized non-SELECT statement* (a non-SELECT that contains input parameters) with the EXECUTE...USING statement. The following statements are parameterized non-SELECT statements:

- A DELETE or UPDATE statement with input parameters in the WHERE clause
- An UPDATE statement with input parameters in the SET clause
- An INSERT statement with input parameters in the VALUES clause

- An EXECUTE PROCEDURE statement with input parameters for its function arguments

Tip: You cannot use an input parameter as the procedure name for a user-defined procedure.

For example, the following UPDATE statement requires two parameters in its WHERE clause:

```
EXEC SQL prepare upd_id from
   'update orders set paid_date = NULL \
    where order_date > ? and customer_num = ?';
```

The USING clause lists the names of the host variables that hold the parameter data. If the input parameter values are stored in **hvar1** and **hvar2**, your program can execute this UPDATE with the following statement:

```
EXEC SQL execute upd_id using :hvar1, :hvar2;
```

The following steps describe how to handle a parameterized UPDATE or DELETE statement when the type and number of parameters are known at compile time:

1. Declare a host variable for each input parameter that is in the prepared statement.
2. Assemble the character string for the statement, with a question mark (?) placeholder for each input parameter. Once you have assembled the string, prepare it. For more information about these steps, see "Assembling and Preparing the SQL Statement" on page 14-2.
3. Assign a value to the host variable that is associated with each input parameter. (The application might obtain these values interactively.)
4. Execute the UPDATE or DELETE statement with the EXECUTE...USING statement. You must list the host variables that contain the input parameter values in the USING clause.
5. Optionally, use the FREE statement to release the resources that were allocated with the prepared statement.

Important: If you do not know the number and data types of the input parameters in the prepared statement at compile time, do not use host variables with the USING clause. Instead, use a dynamic-management structure to specify input parameter values.

For more information about determining the number and types of input parameters, see "Determining Unknown Input Parameters" on page 15-17.

For more information about the USING clause, see the entry for EXECUTE in the *IBM Informix Guide to SQL: Syntax*.

Using an OPEN USING Statement

You can execute the following statements with the OPEN...USING statement:

- A *parameterized SELECT statement* (a SELECT statement that contains input parameters in its WHERE clause) that returns one or more rows
- A *parameterized EXECUTE FUNCTION statement* (a cursor function that contains input parameters for its arguments)

Tip: You cannot use an input parameter as the function name for a user-defined function.

For example, the following SELECT statement is a parameterized SELECT that requires two parameters in its WHERE clause:

```
EXEC SQL prepare slct_id from
   'select from orders where customer_num = ? and order_date > ?';
EXEC SQL declare slct_cursor cursor for slct_id;
```

If the **cust_num** and **ord_date** host variables contain the input parameter values, the following OPEN statement executes the SELECT with these input parameters:

```
EXEC SQL open slct_id using :cust_num, :ord_date;
```

Use the USING *host_var* clause only when you know, at compile time, the type and number of input parameters in the WHERE clause of the SELECT statement. For more information about the USING clause, see the entry for OPEN in the *IBM Informix Guide to SQL: Syntax*.

The **demo2.ec** sample program shows how to handle a dynamic SELECT statement that has input parameters in its WHERE clause. The **demo2.ec** program uses a host variable to hold the value of the input parameter for a SELECT statement. It also uses host variables to hold the column values that are returned from the database.

```
 1. #include <stdio.h>
 2. EXEC SQL define FNAME_LEN      15;
 3. EXEC SQL define LNAME_LEN      15;
 4. main()
 5. {
 6. EXEC SQL BEGIN DECLARE SECTION;
 7.    char demoquery[80];
 8.    char queryvalue[2];
 9.    char fname[ FNAME_LEN + 1 ];
10.    char lname[ LNAME_LEN + 1 ];
11. EXEC SQL END DECLARE SECTION;
12.    printf("DEMO2 Sample ESQL program running.\n\n");
13.    EXEC SQL connect to'stores7';
14. /* The next three lines have hard-wired the query. This
15.  * information could have been entered from the terminal
16.  * and placed into the demoquery string
17.  */
18.    sprintf(demoquery, "%s %s",
19.          "select fname, lname from customer",
20.          "where lname > ? ");
21.    EXEC SQL prepare demo2id from :demoquery;
```

Lines 9 and 10

These lines declare a host variable (**fname**) for the parameter in the WHERE clause of the SELECT statement and declare host variables (**fname** and **lname**) for values that the SELECT statement returns.

Lines 14 to 21

These lines assemble the character string for the statement (in **demoquery**) and prepare it as the **demo2id** statement identifier. The question mark (?) indicates the input parameter in the WHERE clause. For more information about these steps, see "Assembling and Preparing the SQL Statement" on page 14-2.

```
22.    EXEC SQL declare demo2cursor cursor for demo2id;
23.    /* The next line has hard-wired the value for the parameter.
24.     * This information could also have been entered from the
25.       * terminal
       * and placed into the queryvalue string.
```

```
26.     */
27.    sprintf(queryvalue, "C");
28.       EXEC SQL open demo2cursor using :queryvalue;
29.    for (;;)
30.       {
31.       EXEC SQL fetch demo2cursor into :fname, :lname;
32.       if (strncmp(SQLSTATE, "00", 2) != 0)
33.          break;
34.       /* Print out the returned values */
35.       printf("Column: fname\tValue: %s\n", fname);
36.       printf("Column: lname\tValue: %s\n", lname);
37.       printf("\n");
38.       }
```

Line 22

This line declares the **demo2cursor** cursor for the prepared statement identifier, **demo2id**. All nonsingleton SELECT statements *must* have a declared cursor.

Lines 23 to 27

The **queryvalue** host variable is the input parameter for the SELECT statement. It contains the value "C". In an interactive application, this value probably would be obtained from the user.

Line 28

The database server executes the SELECT statement when it opens the **demo2cursor** cursor. Because the WHERE clause of the SELECT statement contains input parameters (lines 20 and 21), the OPEN statement includes the USING clause to specify the input parameter value in **queryvalue**.

Lines 29 to 38

This **for** loop executes for each row fetched from the database. The FETCH statement (line 31) includes the INTO clause to specify the **fname** and **lname** host variables for the column values. After this FETCH statement executes, the column values are stored in these host variables.

```
39.    if (strncmp(SQLSTATE, "02", 2) != 0)
40.       printf("SQLSTATE after fetch is %s\n", SQLSTATE);
41.    EXEC SQL close demo2cursor;
42.    EXEC SQL free demo2cursor;
43.    EXEC SQL free demo2id;
44.    EXEC SQL disconnect current;
45.    printf("\nProgram Over.\n");
46. }
```

Lines 39 and 40

Outside the **for** loop, the program tests the SQLSTATE variable again so it can notify the user in the event of successful execution, a runtime error, or a warning (class code not equal to "02").

Line 41

After all the rows are fetched, the CLOSE statement closes the **demo2cursor** cursor.

Lines 42 and 43

These FREE statements release the resources allocated for the prepared statement (line 42) and the database cursor (line 43). Once a cursor or prepared statement has been freed, it cannot be used again in the program.

SQL Statements That Are Not Known at Compile Time

An SQL statement that is *not* known at compile time is usually one that the user enters in an interactive application. When you write an interactive database-query application like DB-Access, you do not know in advance which databases, tables, or columns the user wants to access, or what conditions the user might apply in a WHERE clause. If an Informix ESQL/C application interprets and runs SQL statements that the end user enters, this application does not know what type of information is to be stored in host variables until after the user enters the statement at runtime.

For example, if a program contains the following DELETE statement, you know the number of values and the data types that you will receive, based on the affected columns:

```
DELETE FROM customer WHERE city = ? AND lname > ?
```

You can define host variables whose data types are compatible with the data they receive. However, suppose your program provides a prompt for the user such as:

```
Enter a DELETE statement for the stores7 database:
```

In this case, you do not know until runtime either the name of the table on which the DELETE will take place or the columns that will be listed in the WHERE clause. Therefore, you cannot declare the necessary host variables.

You can dynamically determine a prepared SQL statement as well as information about the tables and columns it accesses with the DESCRIBE statement and the dynamic-management structures. For more information, see Chapter 15, "Determining SQL statements," on page 15-1.

Chapter 15. Determining SQL statements

If you do not know until runtime what SQL statement to execute, you can
dynamically determine that statement with the DESCRIBE statement and use a
dynamic-management structure to hold any values that the statement sends to or
receives from the database server.

These topics contain the following information about how to dynamically
determine an SQL statement:

- What dynamic-management structures exist and which SQL statements access
  them.
- How to use the DESCRIBE statement with a dynamic-management structure.

Using Dynamic-Management Structure

If you do not know the number or data types of values sent to or received from
the database server, use a *dynamic-management structure*. A dynamic-management
structure allows you to pass a variable-length list of data to the database server, or
receive a variable-length list of data from it.

To execute dynamic SQL statements with unknown columns, you can use either of
the following dynamic-management structures in your Informix ESQL/C program:

- A *system-descriptor area* is a language-independent data structure that is the X/Open standard. You allocate and manipulate it with the SQL statements ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR, and DEALLOCATE DESCRIPTOR.
- The **sqlda** structure is a C-language data structure that you manipulate with the same types of C-language statements that you would use to allocate and manipulate other C structures (areas that have the **struct** data type).

  Because this method uses a C-language structure within SQL statements, it is language-dependent and does *not* conform to X/Open standards.

For a given dynamic SQL statement, the dynamic-management structure can hold any of the following information:
- The number of unknown columns in the statement
- For each unknown value, the data type and length, space for the data, and information about any associated indicator variable (its data type, length, and data)

The Informix ESQL/C program can then use this information to determine a host variable of appropriate length and type to hold the value.

A System-Descriptor Area

A *system-descriptor area* is an area of memory declared by Informix ESQL/C to hold data either returned from or sent by a prepared statement. It is the dynamic-management structure that conforms to the X/Open standards.

A system-descriptor area has two parts:
- A fixed-size portion is made up of the COUNT field. This field contains the number of columns described in the system-descriptor area.
- A variable-length portion contains an *item descriptor* for each value in the system-descriptor area. Each item descriptor is a fixed-size structure.

The following figure shows what a system-descriptor area looks like for two values.
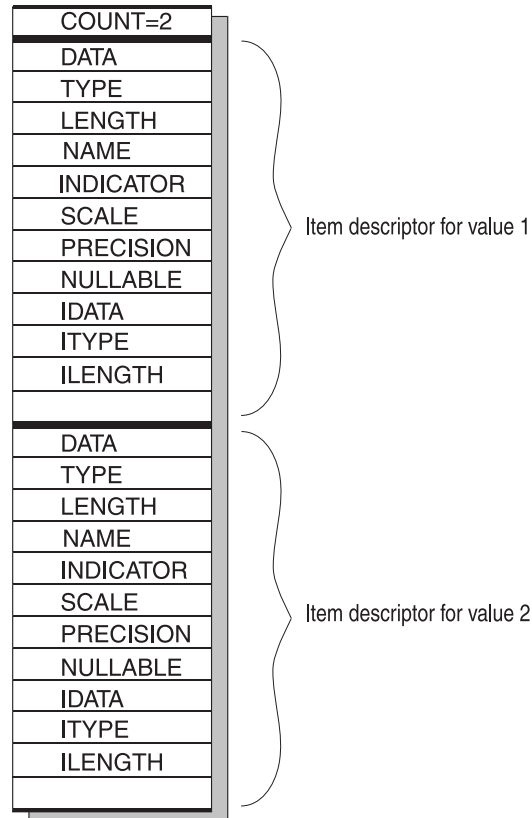
Figure 15-1. Schematic That Shows System-Descriptor Area for Two Values

Fixed-Length Portion

The fixed-size portion of the system-descriptor area consists of the single field, which Table 15-1 shows.

Table 15-1. Field in the Fixed-Size Portion of a System-Descriptor Area

Field	Data Type	Description
COUNT	short	The number of column values or *occurrences* in the system-descriptor area. This is the number of item descriptors, one for each column. The DESCRIBE...USING SQL DESCRIPTOR statement sets COUNT to the number of described columns. You must use SET DESCRIPTOR to initialize the field before you send column values to the database server.

An Item Descriptor

Each item descriptor in the system-descriptor area holds information about a data value that can be sent to or received from the database server. Each item descriptor consists of the fields that Table 15-2 summarizes.

Table 15-2. Fields in Each Item Descriptor of the System-Descriptor Area

Field	Data Type	Description
DATA	char *	A pointer to the column data that is to be sent to or received from the database server.

Table 15-2. Fields in Each Item Descriptor of the System-Descriptor Area (continued)

Field	Data Type	Description
TYPE	**short**	An integer that identifies the data type of the column that is being transferred. These values are defined in the **sqltypes.h** and **sqlxtype.h** header files (see 15-13).
LENGTH	**short**	The length, in bytes, of CHAR type data, the encoded qualifiers of DATETIME or INTERVAL data, or the size of a DECIMAL or MONEY value.
NAME	**char ***	A pointer to the character array that contains the column name or display label that is being transferred.
INDICATOR	**short**	An indicator variable that can contain one of two values: 0: Requires the DATA field to contain non-null data. -1: Inserts a NULL when no DATA field value is specified.
SCALE	**short**	Contains the scale of the column that is in the DATA field; defined *only* for the DECIMAL or MONEY data type.
PRECISION	**short**	Contains the precision of the column that is in the DATA field; defined *only* for the DECIMAL or MONEY data type.
NULLABLE	**short**	Specifies whether the column can contain a null value (after a DESCRIBE statement): 1: The column allows null values 0: The column does *not* allow null values. Before an EXECUTE statement or a dynamic OPEN statement is executed, it must be set to 1 to indicate that an indicator value is specified in the INDICATOR field, and to 0 if it is not specified. (When you execute a dynamic FETCH statement, the NULLABLE field is ignored.)
IDATA	**char ***	User-defined indicator data or the name of a host variable that contains indicator data for the DATA field. The IDATA field is not a standard X/Open field.
ITYPE	**short**	The data type for a user-defined indicator variable. These values are defined in the sqltypes.h and **sqlxtype.h** header files. (See 15-13.) The ITYPE field is not a standard X/Open field.
ILENGTH	**short**	The length, in bytes, of the user-defined indicator. The ILENGTH field is *not* a standard X/Open field.

Table 15-2. Fields in Each Item Descriptor of the System-Descriptor Area (continued)

Field	Data Type	Description
EXTYPEID	**int4**	The extended identifier for the user-defined (opaque or distinct) or complex (collection or row) data type. See the *IBM Informix Guide to SQL: Reference* for a description of the **extended_id** column of the **sysxtdtypes** system catalog table.
EXTYPENAME	**char ***	The name of the user-defined (opaque or distinct) or complex (collection or row) data type. See the *IBM Informix Guide to SQL: Reference* for a description of the **name** column of the **sysxtdtypes** system catalog table.
EXTYPELENGTH	**short**	The length, in bytes, of the string in the EXTYPENAME field.
EXTYPEOWNERNAME	**char ***	The name of the owner (for ANSI databases) of the user-defined (opaque or distinct) or complex (collection or row) data type. See the *IBM Informix Guide to SQL: Reference* for a description of the **owner** column of the **sysxtdtypes** system catalog.
EXTYPEOWNERLENGTH	**short**	The length, in bytes, of the string in the EXTYPEOWNERNAME field.
SOURCETYPE	**short**	The data-type constant (from **sqltypes.h**) of the source data type for a distinct-type column. See the *IBM Informix Guide to SQL: Reference* for a description of the **type** column of the **sysxtdtypes** system catalog.
SOURCEID	**int4**	The extended identifier of the source data type for a distinct-type column. See the *IBM Informix Guide to SQL: Reference* for a description of the **source** column of the **sysxtdtypes** system catalog.

For information about how to use a system-descriptor area to dynamically execute SQL statements, see Chapter 16, "Using a system-descriptor area," on page 16-1.

An sqlda Structure

The **sqlda** structure is a C structure (defined in the **sqlda.h** header file) that holds data returned from a prepared statement. Each **sqlda** structure has three parts:

- A fixed-size portion is made up of the **sqld** field, which contains the number of columns described in the **sqlda** structure.
- A variable-length portion contains an **sqlvar_struct** structure for each column value. Each **sqlvar_struct** structure is a fixed-size structure.
- Descriptive information is included about the **sqlda** structure itself. For more information, see Table 15-6 on page 15-8.

The following figure shows what an **sqlda** structure looks like for two values.
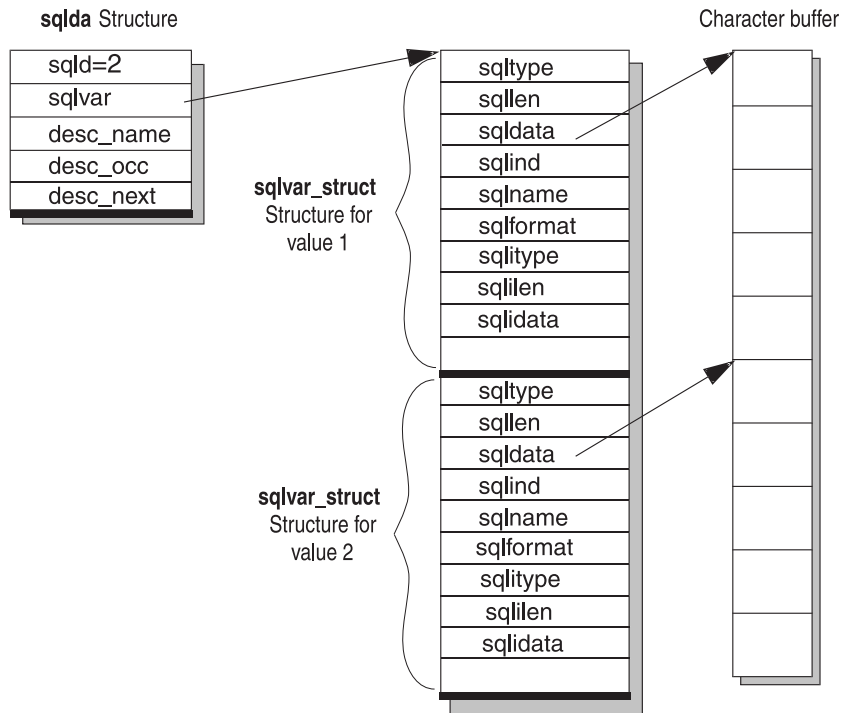
Figure 15-2. Schematic That Shows sqlda Structure for Two Values

Figure 15-2 shows the column data in the **sqldata** fields in a single data buffer. This data can also be stored in separate buffers. For more information, see "Allocating Memory for Column Data" on page 17-6.

Fixed-Length Portion

The following table describes the fixed-size portion of the **sqlda** structure, which consists of a single field.

Table 15-3. Field in the Fixed-Size Portion of an sqlda Structure

Field	Data Type	Description
sqld	**short**	The number of column values or *occurrences* in the **sqlda** structure. This is the number of **sqlvar_struct** structures, one for each column. The DESCRIBE...INTO statement sets **sqld** to the number of described columns. You must set **sqld** to initialize the field before you send column values to the database server.

An sqlvar_struct Structure

When all of its components are fully defined, the **sqlda** structure points to the initial address of a sequence of **sqlvar_struct** structures that contain the necessary information for each variable in the set. Each **sqlvar_struct** structure holds a data value that can be sent to or received from the database server. Your program accesses these **sqlvar_struct** structures through the **sqlvar** field of **sqlda**. Table 15-4 and Table 15-5 on page 15-7 summarize the variable-sized structure of **sqlda**.

Table 15-4. Field to Access the Variable-Sized Portion of an sqlda Structure

Field	Data Type	Description
sqlvar	**struct sqlvar_struct ***	A pointer to the variable-sized portion of an sqlda structure. There is one **sqlvar_struct** for each column value returned from or sent to the database server. The **sqlvar** field points to the *first* of the **sqlvar_struct** structures.

Table 15-5 shows the fields in the **sqlvar_struct** structure.

Table 15-5. Fields in the sqlvar_struct Structure

Field	Data Type	Description
sqltype	short	An integer that identifies the data type of the column that the database server sends or receives. These values are defined in the sqltypes.h and**sqlxtype.h** header files. (See 15-13.)
sqllen	short	The length, in bytes, of CHAR type data, or the encoded qualifier of a DATETIME or INTERVAL value.
sqldata	char *	A pointer to the column data that the database server sends or receives. (See 17-6.)
sqlind	short *	A pointer to an indicator variable for the column that can contain one of two values: 0: The **sqldata** field contains non-null data. -1: The **sqldata** field contains null data.
sqlname	char *	A pointer to a character array that contains the column name or display label that the database server sends or receives.
sqlformat	char *	*Reserved for future use.*
sqlitype	short	An integer that specifies the data type of a user-defined indicator variable. These values are defined in the **sqltypes.h** and **sqlxtype.h** header files. (See 15-13.)
sqlilen	int4	The length, in bytes, of a user-defined indicator variable.
sqlidata	char *	A pointer to the data of the user-defined indicator variable.
sqlxid	int4	The extended identifier for the user-defined (opaque or distinct) or complex (collection or row) data type. See the *IBM Informix Guide to SQL: Reference* for a description of the **extended_id** column of the **sysxtdtypes** system catalog table.
sqltypename	char *	The name of the user-defined (opaque or distinct) or complex (collection or row) data type. See the *IBM Informix Guide to SQL: Reference* for a description of the **name** column of the **sysxtdtypes** system catalog table.
sqltypelen	short	The length, in bytes, of the string in the sqltypename field.
sqlownername	char *	The name of the owner (for ANSI databases) of the user-defined (opaque or distinct) or complex (collection or row) data type. See the *IBM Informix Guide to SQL: Reference* for a description of the **owner** column of the **sysxtdtypes** system catalog.

Table 15-5. Fields in the sqlvar_struct Structure (continued)

Field	Data Type	Description
sqlownerlen	short	The length, in bytes, of the string in the sqlownername field.
sqlsourcetype	short	The data-type constant (from sqltypes.h) of the source data type for a distinct-type column. See the *IBM Informix Guide to SQL: Reference* for a description of the **type** column of the **sysxtdtypes** system catalog.
sqlsourceid	int4	The extended identifier of the source data type for a distinct-type column. See the *IBM Informix Guide to SQL: Reference* for a description of the **source** column of the **sysxtdtypes** system catalog.
sqlflags	int4	This field is usually for internal use. However, if the **sqlda** structure has been initialized by a DESCRIBE statement you can determine if the column accepts nulls by using the **ISCOLUMNNULLABLE()** macro on this field. If it returns 1 then the column accepts nulls. The **ISCOLUMNNULLABLE()** macro is defined in sqltypes.h.

Descriptive Information

Table 15-6 summarizes the **sqlda** fields that describe the **sqlda** structure itself.

Table 15-6. Descriptive Fields in the sqlda Structure

Field	Data Type	Description
desc_name	char[19]	The name of the descriptor; maximum of 18 characters
desc_occ	short	The size of the sqlda structure
desc_next	struct sqlda *	A pointer to the next sqlda structure

For information about how to use an **sqlda** structure to dynamically execute SQL statements, see Chapter 17, "Using an sqlda structure," on page 17-1.

Using the DESCRIBE Statement

This section provides a brief introduction to the DESCRIBE statement. For more information about the DESCRIBE statement, see its entry in the *IBM Informix Guide to SQL: Syntax*.

The DESCRIBE statement obtains information about database columns or expressions in a prepared statement. It can put this information in one of the following dynamic-management structures:

- DESCRIBE...USING SQL DESCRIPTOR stores information in a system-descriptor area.

  Each item descriptor describes a column. The COUNT field is set to the number of item descriptors (the number of columns in the column list). You can access this information with the GET DESCRIPTOR statement. For more information about the fields of a system-descriptor area, see Figures 15-1 through 15-2.

- DESCRIBE...INTO *sqlda_ptr* stores information in an **sqlda** structure whose address is stored in *sqlda_ptr*.

  Each **sqlvar_struct** structure describes a column. The **sqld** field is set to the number of **sqlvar_struct** structures (the number of columns in the column list).

You can access this information through the fields in the **sqlvar_struct** structures. For more information about the fields of an **sqlda** structure, see Figures 15-2 through 15-6.

Important: If the Deferred-PREPARE feature is enabled, you cannot use the DESCRIBE statement before an OPEN statement executes.

For more information, see "Deferring Execution of the PREPARE Statement" on page 14-17.

If the DESCRIBE is successful, it obtains the following information about a prepared statement:

- The SQLCODE value indicates the type of statement that was prepared. For more information, see "Determining the Statement Type" on page 15-9.

- A dynamic-management structure contains information about the number and data types of the columns in a column list of a SELECT, INSERT, or EXECUTE FUNCTION statement.

  For information about the column descriptions returned by DESCRIBE, see Handling an Unknown Select List and Handling an Unknown Column List on 15-16 and 15-17. For information about the data type values returned by DESCRIBE, see "Determining the Data Type of a Column" on page 15-13.

- When the DESCRIBE statement describes a DELETE or UPDATE statement, it can indicate whether the statement includes a WHERE clause. For more information, see "Checking for a WHERE Clause" on page 15-15.

Determining the Statement Type

The **sqlstype.h** file contains the defined integer constants for the SQL statements that can be prepared. The DESCRIBE statement returns one of these values in the SQLCODE (**sqlca.sqlcode**) variable to identify a prepared statement. That is, SQLCODE indicates whether the statement was an INSERT, SELECT, CREATE TABLE, or any other SQL statement.

Within an Informix ESQL/C program that uses dynamic SQL statements, you can use the constants that Table 15-7 shows to determine which SQL statement was prepared.

Table 15-7. The Constants for SQL Statement Types That the sqlstype.h File Defines

SQL Statement	Defined sqlstype.h Constant	Value
SELECT (no INTO TEMP clause)	None	0
DATABASE	SQ_DATABASE	1
	Internal use only	2
SELECT INTO TEMP	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
	Internal use only	9
LOCK TABLE	SQ_LOCK	10
UNLOCK TABLE	SQ_UNLOCK	11

Table 15-7. The Constants for SQL Statement Types That the sqlstype.h File Defines (continued)

SQL Statement	Defined sqlstype.h Constant	Value
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
	Internal use only	23–28
ALTER TABLE	SQ_ALTER	29
UPDATE STATISTICS	SQ_STATS	30
CLOSE DATABASE	SQ_CLSDB	31
DELETE (no WHERE clause)	SQ_DELALL	32
UPDATE (no WHERE clause)	SQ_UPDALL	33
BEGIN WORK	SQ_BEGWORK	34
COMMIT WORK	SQ_COMMIT	35
ROLLBACK WORK	SQ_ROLLBACK	36
	Internal use only	37–39
CREATE VIEW	SQ_CREVIEW	40
DROP VIEW	SQ_DROPVIEW	41
	Internal use only	42
CREATE SYNONYM	SQ_CREASYN	43
DROP SYNONYM	SQ_DROPSYN	44
CREATE TEMP TABLE	SQ_CTEMP	45
SET LOCK MODE	SQ_WAITFOR	46
ALTER INDEX	SQ_ALTIDX	47
SET ISOLATION, SET TRANSACTION	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, EXECUTE FUNCTION	SQ_EXECPROC	56

Table 15-7. The Constants for SQL Statement Types That the sqlstype.h File Defines (continued)

SQL Statement	Defined sqlstype.h Constant	Value
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
RESERVE (Optical)	SQ_OPRESERVE	61
RELEASE (Optical)	SQ_OPRELEASE	62
SET MOUNTING TIMEOUT	SQ_OPTIMEOUT	63
UPDATE STATS...for procedure	SQ_PROCSTATS	64
	Defined for Kanji version only	65 and 66
	Reserved	67–69
CREATE TRIGGER	SQ_CRETRIG	70
DROP TRIGGER	SQ_DRPTRIG	71
	SQ_UNKNOWN	72
SET DATASKIP	SQ_SETDATASKIP	73
SET PDQPRIORITY	SQ_PDQPRIORITY	74
ALTER FRAGMENT	SQ_ALTFRAG	75
SET	SQ_SETOBJMODE	76
START VIOLATIONS TABLE	SQ_START	77
STOP VIOLATIONS TABLE	SQ_STOP	78
	Internal use only	79
SET SESSION AUTHORIZATION	SQ_SETDAC	80
	Internal use only	81-82
CREATE ROLE	SQ_CREATEROLE	83
DROP ROLE	SQ_DROPROLE	84
SET ROLE	SQ_SETROLE	85
	Internal use only	86–89
CREATE ROW TYPE	SQ_CREANRT	90
DROP ROW TYPE	SQ_DROPNRT	91
CREATE DISTINCT TYPE	SQ_CREADT	92
CREATE CAST	SQ_CREACT	93
DROP CAST	SQ_DROPCT	94
CREATE OPAQUE TYPE	SQ_CREABT	95
DROP TYPE	SQ_DROPTYPE	96
	Reserved	97
CREATE ACCESS_METHOD	SQ_CREATEAM	98
DROP ACCESS_METHOD	SQ_DROPAM	99
	Reserved	100
CREATE OPCLASS	SQ_CREATEOPC	101
DROP OPCLASS	SQ_DROPOPC	102

Table 15-7. The Constants for SQL Statement Types That the sqlstype.h File Defines (continued)

SQL Statement	Defined sqlstype.h Constant	Value
CREATE CONSTRUCTOR	SQ_CREACST	103
SET (MEMORY/NON)_RESIDENT	SQ_SETRES	104
CREATE AGGREGATE	SQ_CREAGG	105
DROP AGGREGATE	SQ_DRPAGG	106
pload log file command	SQ_PLOADFILE	107
onutil check index command	SQ_CHKIDX	108
set schedule	SQ_SCHEDULE	109
"set environment..."	SQ_SETENV	110
	Reserved	111
	Reserved	112
	Reserved	113
	Reserved	114
SET STMT_CACHE	SQ_STMT_CACHE	115
RENAME INDEX	SQ_RENIDX	116
CREATE SEQUENCE	SQ_CRESEQ	124
DROP SEQUENCE	SQ_DRPSEQ	125
ALTER SEQUENCE	SQ_ALTERSEQ	126
RENAME SEQUENCE	SQ_RENSEQ	127
SET COLLATION	SQ_COLLATION	129
SET NO COLLATION	SQ_NOCOLLATION	130
SET ROLE DEFAULT	SQ_SETDEFROLE	131
SET ENCRYPTION	SQ_ENCRYPTION	132
save external directives	SQ_EXTD	133
CREATE XAdatasource TYPE	SQ_CRXASRCTYPE	134
CREATE XAdatasource	SQ_CRXADTSRC	135
DROP XAdatasource TYPE	SQ_DROPXATYPE	136
DROP XAdatasource	SQ_DROPXADTSRC	137
Truncate table	SQ_TRUNCATE	138
CREATE SECURITY LABEL COMPONENT	SQ_CRESECCMP	139
ALTER SECURITY LABEL COMPONENT	SQ_ALTSECCMP	140
DROP SECURITY LABEL COMPONENT	SQ_DRPSECCMP	141
RENAME SECURITY LABEL COMPONENT	SQ_RENSECCMP	142
CREATE SECURITY POLICY	SQ_CRESECPOL	143
DROP SECURITY POLICY	SQ_DRPSECPOL	144
RENAME SECURITY POLICY	SQ_RENSECPOL	145
CREATE SECURITY LABEL	SQ_CRESECLAB	146
DROP SECURITY LABEL	SQ_DRPSECLAB	147
RENAME SECURITY LABEL	SQ_RENSECLAB	148

SQL Statement	Defined sqlstype.h Constant	Value
GRANT DBSECADM	SQ_GRTSECADM	149
REVOKE DBSECADM	SQ_RVKSECADM	150
GRANT EXEMPTIONS	SQ_GRTSECEXMP	151
REVOKE EXEMPTIONS	SQ_RVKSECEXMP	152
GRANT SECURITY LABEL	SQ_GRTSECLAB	153
REVOKE SECURITY LABEL	SQ_RVKSECLAB	154
GRANT SETSESSIONAUTH	SQ_GRTSESAUTH	155
REVOKE SETSESSIONAUTH	SQ_RVKSESAUTH	156

Tip: Check the **sqlstype.h** header file on your system for the most updated list of SQL statement-type values.

To determine the type of SQL statement that was prepared dynamically, your Informix ESQL/C program must take the following actions:

- Use the **include** directive to include the **sqlstype.h** header file.
- Compare the value in the SQLCODE variable (**sqlca.sqlcode**) against the constants defined in the **sqlstype.h** file.

The sample program that starts in 16-15 uses the SQ_EXECPROC constant to verify that an EXECUTE FUNCTION statement has been prepared.

Determining the Data Type of a Column

The DESCRIBE statement identifies the data type of a column with an integer value. After DESCRIBE analyzes a prepared statement, it stores this value in a dynamic-management structure, as follows:

- In a system-descriptor area, in the TYPE field of the item descriptor for each column described
- In an **sqlda** structure, in the **sqltype** field of the **sqlvar_struct** structure for each column described

IBM Informix ESQL/C provides defined constants for these data types in the following two header files:

- The **sqltypes.h** header file contains defined constants for the Informix-specific SQL data types. These values are the default that the DESCRIBE statement uses.
- The **sqlxtype.h** header file contains defined constants for the X/Open SQL data types. DESCRIBE uses these values when you compile your Informix ESQL/C source file with the **-xopen** option of the preprocessor.

Use the SQL data type constants from **sqltypes.h** or **sqlxtype.h** to analyze the information returned by a DESCRIBE statement or to set the data type of a column before execution.

Tip: When you set the data type of a column in a system-descriptor area, you assign a data-type constant to the TYPE field (and optionally the ITYPE field) of an item descriptor with the SET DESCRIPTOR statement. For more information, see "Assigning and Obtaining Values from a System-Descriptor Area" on page 16-5. When you set the data type of a column in an **sqlda**

structure, you assign a data-type constant to the **sqltype** field (and optionally the **sqlitype** field) of an **sqlvar** structure. For more information, see "Assigning and Obtaining Values from an sqlda Structure" on page 17-8.

Informix-Specific SQL Data Types

The Informix-specific SQL data types are available to a column in an Informix database. The *IBM Informix Guide to SQL: Reference* describes these data types. If you do not include the **-xopen** option when you compile your Informix ESQL/C program, the DESCRIBE statement uses these data types to specify the data type of a column or the return value of a user-defined function. Constants for these Informix SQL data types are defined in the Informix ESQL/C **sqltypes.h** header file.

Figure 15-3 shows some of the SQL data-type entries in **sqltypes.h**.

```
#define SQLCHAR        0
#define SQLSMINT       1
#define SQLINT         2
#define SQLFLOAT       3
#define SQLSMFLOAT     4
#define SQLDECIMAL     5
#define SQLSERIAL      6
#define SQLDATE        7
#define SQLMONEY       8
    .
    .
    .
```

Figure 15-3. Some Informix SQL Data-Type Constants

For a complete list of constants for SQL data types, see Table 3-3 on page 3-5. The integer values in Figure 15-3 are language-independent constants; they are the same in *all* IBM Informix embedded products.

X/Open SQL Data Types

The X/Open standards support only a subset of the Informix-specific SQL data types. To conform to the X/Open standards, you must use the X/Open SQL data-type constants. The DESCRIBE statement uses these constants to specify the data type of a column (or a return value) when you compile your Informix ESQL/C program with the **-xopen** option.

The X/Open data type constants are defined in the **sqlxtype.h** header file. For more information, see "X/Open Data Type Constants" on page 3-6.

Constants for ESQL/C Data Types

The **sqltypes.h** header file also contains defined constants for the Informix ESQL/C data types. The Informix ESQL/C data types are assigned to host variables in an Informix ESQL/C program. If your program initializes a column description, it usually obtains the column value from an Informix ESQL/C host variable. To set the column data type for this value, the program must use the Informix ESQL/C data types.

Figure 15-4 shows only some of the Informix ESQL/C data type entries in the **sqltypes.h** header file. For a complete list of constants for Informix ESQL/C data types, see Table 3-3 on page 3-5.

```
#define CCHARTYPE            100
#define CSHORTTYPE           101
#define CINTTYPE             102
#define CLONGTYPE            103
#define CFLOATTYPE           104
#define CDOUBLETYPE          105
    ⋮
```

Figure 15-4. Some ESQL/C Data-Type Constants from the sqltypes.h Header File

Within an Informix ESQL/C program that uses dynamic SQL statements, you can use the constants that are shown in Figure 15-4 to set the data types of the associated host variables. Use the Informix ESQL/C data-type constants to set the data types of host variables used as input parameters to a dynamically defined SQL statement or as storage for column values that are returned by the database server. The sample program in 16-19 stores a TEXT value into a database table.

Determining Input Parameters

You can use the DESCRIBE and DESCRIBE INPUT to return input parameter information for a prepared statement before it is executed.

The DESCRIBE INPUT statement returns the number, data types, size of the values, and the name of the column or expression that the query returns. The DESCRIBE INPUT statement can return parameter information for the following statements:

- INSERT using WHERE clause
- UPDATE using WHERE clause
- SELECT with IN, BETWEEN, HAVING, and ON clauses.
- SELECT subqueries
- SELECT INTO TEMP
- DELECT
- EXECUTE

For more information about using the DESCRIBE and DESCRIBE INPUT statements, see the *IBM Informix Guide to SQL: Syntax*.

Checking for a WHERE Clause

When DESCRIBE analyzes a prepared DELETE or UPDATE statement, it indicates if the statement includes a WHERE clause, as follows:

- It sets the **sqlca.sqlwarn.sqlwarn0** and **sqlca.sqlwarn.sqlwarn4** fields to W if the prepared statement was an UPDATE or DELETE *without* a WHERE clause.
- It sets the SQLSTATE variable to an Informix-specific warning value of "01I07".

Your program can check for either of these conditions to determine the type of DELETE or UPDATE statement that was executed. If the DELETE or UPDATE does not contain a WHERE clause, the database server deletes or updates all rows in the table. For more information about how to execute DELETE and UPDATE statements dynamically with a system-descriptor area, see 16-28; for information about how to use an **sqlda** structure, see 17-27.

Determining Statement Information at Runtime

Consider a dynamic-management structure when you execute an SQL statement under the following conditions:

- Something is not known about the *structure* of an SQL statement:
  - The type of statement to execute is unknown.
  - The table name is unknown and therefore the columns to be accessed are unknown.
  - The WHERE clause is missing.
- Something is not known about *the number or type of values* that passes between the Informix ESQL/C program and the database server:
  - The number and data types of columns in the select list of a SELECT or in a column list of an INSERT
  - The number and data types of input parameters in the statement are unknown
  - The number and data types of return values of a user-defined function (executed with the EXECUTE FUNCTION statement) are unknown

The following sections briefly outline how to handle each of the categories of unknown number and data types of values.

Handling an Unknown Select List

For a SELECT statement, the columns in the select list identify the column values that are received from the database server. In the SELECT statement described and illustrated in the **demo1.ec** example program (see 1-39), the values returned from the query are placed into the host variables that are listed in an INTO *host_var* clause of the SELECT statement.

However, when your program creates a SELECT statement at runtime, you cannot use an INTO clause because you do not know at compile time what host variables are needed. If the type and number of the values that your Informix ESQL/C program receives are not known at compile time, your program must perform the following tasks:

1. Declare a dynamic-management structure to serve as storage for the select-list column definitions. This structure can be either a system-descriptor area or an **sqlda** structure.

   Use of the system-descriptor area conforms to X/Open standards.

2. Use the DESCRIBE statement to examine the select list of the prepared SELECT statement and describes the columns.

3. Specify the dynamic-management structure as the location of the data fetched from the database. From the dynamic-management structure, the program can move the column values into host variables.

Important: Use a dynamic-management structure only if you do not know the number and data types of the select-list columns at compile time.

For information about how to execute a SELECT if you do know the number and data types of select-list columns, see "Executing SELECT Statements" on page 14-26. For information about how to identify columns in the select list of a SELECT statement with a system-descriptor area, see 16-9. For more information about how to use an **sqlda** structure, see 17-11.

Handling an Unknown Column List

For an INSERT statement, the values in the VALUES clause identify the column values to be inserted into the new row. If the data types and number of the values that the IBM Informix ESQL/C program inserts are not known at compile time, you cannot simply use host variables to hold the data being inserted. Instead, your program must perform the following tasks:

1. Define a dynamic-management structure to serve as storage for the unknown column definitions. This structure can be either a system-descriptor area or an **sqlda** structure.

   Use of the system-descriptor area conforms to X/Open standards.

2. Use the DESCRIBE statement to examine the column list of the prepared INSERT statement and describe the columns.

3. Specify the dynamic-management structure as the location of the data to be inserted when the INSERT statement executes.

Important: Use a dynamic-management structure only if you do not know the number and data types of the column-list columns at compile time. For information about how to execute an INSERT if you do know the number and data types of column-list columns, see "Executing Non-SELECT Statements" on page 14-25.

For information about how to identify columns in the VALUES column list of an INSERT statement with a system-descriptor area, see 16-18. To use an **sqlda** structure, see 17-18.

Determining Unknown Input Parameters

If you know the data types and number of input parameters of an SQL statement, use the USING *host_var* clause (see 14-33). However, if you do not know the data types and number of these input parameters at compile time, you cannot use host variables to provide the parameter values; you do not have enough information about the parameters to declare the host variables.

Neither can you use the DESCRIBE statement to define the unknown parameters because DESCRIBE does not examine:

- A WHERE clause (for a SELECT, UPDATE, or DELETE statement
- The arguments of a user-defined routine (for an EXECUTE FUNCTION or EXECUTE PROCEDURE statement)

Your Informix ESQL/C program must follow these steps to define the input parameters in any of the preceding statements:

1. Determine the number and data types of the input parameters. Unless you write a general-purpose, interactive interpreter, you usually have this information. If you do not have it, you must write C code that analyzes the statement string and obtains the following information:

   - The number of input parameters [question marks (?)] that appear in the WHERE clause of the statement string or as arguments of a user-defined routine
   - The data type of each input parameter based on the column (for WHERE clauses) or parameter (for arguments) to which it corresponds

2. Store the definitions and values of the input parameters in a dynamic-management structure. This structure can be either a system-descriptor area or an **sqlda** structure.

Use of the system-descriptor area conforms to X/Open standards.

3. Specify the dynamic-management structure as the location of the input parameter values when the statement executes.

Important: Use a dynamic-management structure only if you do not know the number and data types of the input parameters at compile time. For information about how to execute a parameterized SQL statement if you do know the number and data types of column-list columns, see "Execute statements with input parameters" on page 14-33.

For information about how to handle input parameters in the WHERE clause of a dynamic SELECT statement with a system-descriptor area, see "Handling a Parameterized SELECT Statement" on page 16-22; to use an **sqlda** structure, see "Handling a Parameterized SELECT Statement" on page 17-20. For information about how to handle input parameters as arguments of a user-defined function with a system-descriptor area, see "Handling a parameterized user-defined routine" on page 16-27; to use an **sqlda** structure, see "Handling a Parameterized User-Defined Routine" on page 17-26. For information about how to handle input parameters in the WHERE clause of a dynamic UPDATE or DELETE statement with a system-descriptor area, see "Handling a Parameterized UPDATE or DELETE Statement" on page 16-28; to use an **sqlda** structure, see "Handling a Parameterized UPDATE or DELETE Statement" on page 17-27.

Determining Return Values Dynamically

For an EXECUTE FUNCTION statement, the values in the INTO clause identify where to store the return values of a user-defined function. If the data types and number of the function return values are *not* known at compile time, you cannot use host variables in the INTO clause of EXECUTE FUNCTION to hold the values. Instead, your program must perform the following tasks:

1. Define a dynamic-management structure to serve as storage for the definitions of the value(s) that the user-defined function returns.

   You can use either a system-descriptor area or an **sqlda** structure to hold the return value(s).

   Use of the system-descriptor area conforms to X/Open standards.

2. Use the DESCRIBE statement to examine the prepared EXECUTE FUNCTION statement and describe the return value(s).

3. Specify the dynamic-management structure as the location of the data returned by the user-defined function.

   From the dynamic-management structure, the program can move the return values into host variables.

Important: Use a dynamic-management structure only if you do not know at compile time the number and data types of the return values that the user-defined function returns. If you know this information at compile time, see "Executing User-Defined Routines in IBM Informix" on page 14-30 for more information.

For information about how to use a system-descriptor area to hold function return values, see "Handling Unknown Return Values" on page 16-13. To use an **sqlda** structure to hold return values, see "Handling Unknown Return Values" on page 17-16.

Handling Statements That Contain User-Defined Data Types

This section provides information about how to perform dynamic SQL on statements that contain columns with the following user-defined data types:

- Opaque data types: an encapsulated data type that the user can define
- Distinct data types: a data type that has the same internal storage representation as its source type, but has a different name

SQL Statements with Opaque-Type Columns

For dynamic execution of opaque-type columns, keep the following items in mind:

- You must ensure that the type and length fields of the dynamic-management structure (system-descriptor area or **sqlda** structure) match the data type of the data you insert into an opaque-type column.
- Informix ESQL/C truncates opaque-type data at 32 kilobytes if the host variable is not large enough to hold the data.

Inserting Opaque-Type Data: When the DESCRIBE statement describes a prepared INSERT statement, it sets the type and length fields of a dynamic-management structure to the data type of the column. Table 15-8 shows the type and length fields for the dynamic-management structures.

Table 15-8. Type and Length fields of Dynamic-Management Structures

Dynamic-Management Structure	Type Field	Length Field
system-descriptor area	TYPE field of an item descriptor	LENGTH field of an item descriptor
sqlda structure	**sqltype** field of an **sqlvar_struct** structure	**sqllen** field of an **sqlvar_struct** structure

If the INSERT statement contains a column whose data type is an opaque data type, the DESCRIBE statement identifies this column with one of the following type-field values:

- The SQLUDTFIXED constant for fixed-length opaque types
- The SQLUDTVAR constant for varying-length opaque types

These data-type constants represent an opaque type in its internal format.

When you put opaque-type data into a dynamic-management structure, you must ensure that the type field and length field are compatible with the data type of the data that you provide for the INSERT, as follows:

- If you provide the opaque-type data in *internal* format, then the type and length fields that DESCRIBE sets are correct.
- If you provide the data in *external* format (or any format other than the internal format), you must change the type and length fields that DESCRIBE has set to be compatible with the data type of the data.

The input and output support functions for the opaque type do not reside on the client computer. Therefore, the client application cannot call them to convert the opaque-type data in the dynamic-management structure from its external to its internal format. To provide the opaque-type data in its external representation, set the type-field value to a character data type. When the database server receives the character data (the external representation of the opaque type), it calls the input support function to convert the external representation of the opaque type to its

internal representation. If the data is some other type and valid support or casting functions exist, the database server can call these functions instead to convert the value.

For example, suppose you use a system-descriptor area to hold the insert values and you want to send the opaque-type data to the database server in its external representation. In the following code fragment, the SET DESCRIPTOR statement resets the TYPE field to SQLCHAR, so that the TYPE field matches the data type of the host variable (char) that it assigns to the DATA field:

```
EXEC SQL BEGIN DECLARE SECTION;
   char extrn_value[100];
   int extrn_lngth;
   int extrn_type;
EXEC SQL END DECLARE SECTION;
   .
   .
   .

EXEC SQL allocate descriptor 'desc1' with max 100;
EXEC SQL prepare ins_stmt from
   'insert into tab1 (opaque_col) values(?)';
EXEC SQL describe ins_stmt using sql descriptor 'desc1';

/* At this point the TYPE field of the item descriptor is
 * SQLUDTFIXED
 */

stcopy("(1, 2, 3, 4)", extrn_value);
extrn_lngth = stleng(extrn_value);
dtype = SQLCHAR;

/* This SET DESCRIPTOR statement assigns the external
 * representation of the data to the item descriptor and
 * resets the TYPE field to SQLCHAR.
 */
EXEC SQL set descriptor 'desc1' value 1
   data = :extrn_value, type = :extrn_type,
   length = :extrn_lngth;
EXEC SQL execute ins_stmt using sql descriptor 'desc1';
```

Truncation of Opaque-Type Data: If you specify a host variable that is not large enough to hold the full return value from the server, Informix ESQL/C normally truncates the data to fit the host variable and puts the actual length in an indicator variable. This indicator variable can be one that you explicitly provide or, for dynamic SQL, one of the following fields of a dynamic-management structure.

Dynamic-Management Structure	Indicator Field
system-descriptor area	INDICATOR field of an item descriptor
sqlda structure	**sqlind** field of an **sqlvar_struct** structure

However, these indicator fields are defined as a **short** integer and therefore can only store sizes up to 32 kilobytes.

This size limitation of the indicator field affects how Informix ESQL/C handles truncation of opaque-type data that is larger than 32 kilobytes. When Informix ESQL/C receives opaque-type data that is larger than 32 kilobytes and the host variable is not large enough to hold the opaque-type data, Informix ESQL/C truncates the data to 32 kilobytes. Informix ESQL/C performs this truncation at 32 kilobytes even if you program a host variable that is larger than 32 kilobytes (but still not large enough for the data).

SQL Statements with Distinct-Type Columns

For dynamic execution of distinct-type columns, the dynamic-management structures have been modified to hold the following information about a distinct type:

- The data-type constant (from **sqltypes.h**) for the source type of the distinct-type column
- The extended identifier for the source type of the distinct-type column

These values are in the following fields of a dynamic-management structure.

Dynamic-Management Structure	Source-Type Field	Extended-Identifier Field
system-descriptor area	SOURCETYPE field of an item descriptor	SOURCEID field of an item descriptor
sqlda structure	**sqlsourcetype** field of an **sqlvar_struct** structure	**sqlsourceid** field of an **sqlvar_struct** structure

When the DESCRIBE statement describes a prepared statement, it stores information about columns of the statement in a dynamic-management structure. (For more information, see "Using the DESCRIBE Statement" on page 15-8.) There is no special constant in the **sqltypes.h** file to indicate a distinct data type. Therefore, the type field of the dynamic-management structure cannot directly indicate a distinct type. (Table 15-8 on page 15-19 shows the type fields of the dynamic-management structures.)

Instead, the type field in the dynamic-management structure has a special value to indicate that a *distinct bit* is set for a distinct-type column. The type field indicates the source type of the distinct data combined with the distinct bit. The **sqltypes.h** header file provides the following data-type constants and macros to identify the distinct bit for a distinct column.

Source Type	Distinct-Bit Constant	Distinct-Bit Macro
LVARCHAR	SQLDLVARCHAR	ISDISTINCTLVARCHAR(*type_id*)
BOOLEAN	SQLDBOOLEAN	ISDISTINCTBOOLEAN(*type_id*)
Any other data type	SQLDISTINCT	ISDISTINCTTYPE(*type_id*)

Use the following algorithm to determine if a column is a distinct type:

```
if (one of the distinct bits is set)
   {
   /* Have a distinct type, now find the source type */
   if (ISDISTINCTLVARCHAR(sqltype))
      {
      /* Is a distinct of LVARCHAR:
       *    type field = SQLUDTVAR + SQLDLVARCHAR
       *    source-type field = 0
       *    source-id field = extended identifier of lvarchar
       */
      }
   else if (ISDISTINCTBOOLEAN(sqltype))
      {
      /* Is a distinct of BOOLEAN
       *    type field = SQLUDTFIXED + SQLDBOOLEAN
       *    source-type field = 0
       *    source-id field = extended id of boolean
       */
```

```
            }
        else
            {
            /* SQLDISTINCT is set */
            if (ISUDTTYPE(sqltype))
                {
                /* Source type is either a built-in simple type or an
                 * opaque data type
                 */
                if (source-id field > 0)
                    /* Is a distinct of an opaque type.
                     * Pick up the xtended identifier of the source type
                     * from the source-id field
                     */
                else
                    /* Is a distinct of a built-in simple type.
                     * Pick up the type id of the source type from the
                     * source-type field
                     */
                }
            else
                {
                /* Source type is a non-simple type, a complex type.
                 * Both the source-type and source-id fields should be 0,
                 * the source type is embedded in the type field:
                 *     type = source type + SQLDISTINCT
                 */
                }
            }
        }
```

The following table summarizes the pseudo-code of the preceding algorithm.

Source Type	Type Field	Source-Type Field	Extended-Identifier Field
Built-in data type	SQLUDTVAR + SQLDISTINCT	Data-type constant of built-in data type	0
LVARCHAR	SQLUDTVAR + SQLDLVARCHAR	0	Extended identifier of LVARCHAR
BOOLEAN	SQLUDTFIXED + SQLDBOOLEAN	0	Extended identifier of BOOLEAN
All other data types	source type + SQLDISTINCT	0	0

Using a Fetch Array

A *fetch array* enables you to increase the number of rows that a single FETCH statement returns from the fetch buffer to an **sqlda** structure in your program. A fetch array is especially useful when you fetch simple-large-object (TEXT or BYTE) data. A fetch of simple-large-object data without a fetch array requires the following two exchanges with the database server:

- When Informix ESQL/C fetches a TEXT or BYTE column, the database server returns the descriptor for the column.

- Informix ESQL/C subsequently requests the database server to obtain the column data.

When you use a fetch array, Informix ESQL/C sends a *series* of simple-large-object descriptors to the database server and the database server returns the corresponding column data all at one time.

To use a fetch array:

1. Declare an **sqlda** structure to hold the columns you want to fetch.

   You cannot use host variables or system-descriptor areas in a FETCH statement to hold fetch arrays for columns. You must use an sqlda structure and the FETCH...USING DESCRIPTOR statement. For information about how to declare and use sqlda structures, see "An sqlda Structure" on page 15-5.

2. Use the DESCRIBE...INTO statement to initialize the **sqlda** structure and obtain information about the prepared query.

   The DESCRIBE...INTO statement allocates memory for the **sqlda** structure and the **sqlvar_struct** structures.

3. For the **sqldata** field, allocate a buffer that is large enough to hold the fetch array for each column.

   To allocate the memory for an **sqldata** field, you must set the **FetArrSize** global variable to the size of the fetch array for the associated column. For more information, see "Allocating Memory for the Fetch Arrays" on page 15-30.

4. Issue the FETCH...USING DESCRIPTOR statement to retrieve the column data into the fetch arrays.

   The FETCH statement puts the retrieved rows into the **sqldata** fields of the **sqlvar_struct** structures in **sqlda**. Each FETCH statement returns into the **sqldata** fields the number of values specified by **FetArrSize**.

5. Obtain the column values from the fetch arrays of each **sqlvar_struct** structure.

   You must obtain these values from the fetch arrays before you perform the next FETCH statement. You can check the **sqlca.sqlerrd[2]** field to determine the number of valid rows that the FETCH has returned. The value in **sqlerrd[2]** should be equal to or smaller than the value you set in **FetArrSize**. For information about the **sqlerrd** array, see Chapter 11, "Exception handling," on page 11-1. For more information about obtaining the column values, see "Obtaining Values from Fetch Arrays" on page 15-34.

6. Repeat steps 4 and 5 until all rows are fetched.

7. Free the memory that the **sqlda** structure uses.

   As with other uses of the sqlda structure, Informix ESQL/C does not release resources for this structure. Your application must free memory allocated to the **sqlda** structure when it no longer needs it. For more information, see "Freeing Memory for a Fetch Array" on page 15-35.

Important: The **FetArrSize** feature does not work when both the Deferred-PREPARE and OPTOFC features are enabled. When these two features are enabled, Informix ESQL/C does not know the size of a row until after the FETCH statement completes. By this time, it is too late for the fetch buffer to be adjusted with the **FetArrSize** value.

The sample program below shows how to perform the preceding steps. It uses separate functions to initialize, print, and free the **sqlda** structure. These functions are described in the following sections.

```
#include <windows.h>
#include
#include

EXEC SQL include sqlda.h;
```

```
EXEC SQL include locator.h;
EXEC SQL include sqltypes.h;


#define BLOBSIZE 32275        /* using a predetermined length for blob */

EXEC SQL begin declare section;
    long blobsize;     /* finding the maximum blob size at runtime */
EXEC SQL end declare section;


/**********************************************************************
* Function: init_sqlda()
* Purpose: With the sqlda pointer that was returned from the DESCRIBE
* statement, function allocates memory for the fetch arrays
* in the sqldata fields of each column. The function uses
* FetArrSize to determine the size to allocate.
* Returns: < 0 for error
* > 0 error with messagesize
**********************************************************************/
int init_sqlda(struct sqlda *in_da, int print)
{
    int i, j,
   row_size=0,
   msglen=0,
   num_to_alloc;
    struct sqlvar_struct *col_ptr;
    ifx_loc_t *temp_loc;
    char *type;


    if (print)
   printf("columns: %d. \n", in_da->sqld);

    /* Step 1: determine row size */
    for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++, col_ptr++)
    {
   /* The msglen variable holds the sum of the column sizes in the
    * database; these are the sizes that DESCRIBE returns. This
    * sum is the amount of memory that ESQL/C needs to store
    * one row from the database. This value is <= row_size. */
   msglen += col_ptr->sqllen; /* get database sizes */


    /* calculate size for C data: string columns get extra byte added
     * to hold null terminator */
    col_ptr->sqllen = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);

    /* The row_size variable holds the sum of the column sizes in
        * the client application; these are the sizes that rtypmsize()
        * returns. This sum is amount of memory that the client
        * application needs to store one row. */
    row_size += col_ptr->sqllen;
    if(print)
        printf("Column %d size:  %d\n", i+1, col_ptr->sqllen);
    }

    if (print)
    {
   printf("Total message size = %d\n", msglen);
   printf("Total row size = %d\n", row_size);
    }

    EXEC SQL select max(length(cat_descr)) into :blobsize from catalog;
```

```
 /* Step 2: set FetArrSize global variable to number of elements
  * in fetch array; this function calculates the FetArrSize
  * value that can fit into the existing fetch buffer.
  * If FetBufSize is not set (equals zero), the code assigns a
  * default size of 4096 bytes (4 kilobytes). Alternatively, you
  * could set FetArrSize to the number elements you wanted to
  * have and let ESQL/C size the fetch buffer. See the text in
  * "Allocating Memory for the Fetch Arrays" for more information.*/
 if (FetArrSize <= 0) /* if FetArrSize not yet initialized */
 {
if (FetBufSize == 0) /* if FetBufSize not set */
   FetBufSize = 4096; /* default FetBufSize */
FetArrSize = FetBufSize/msglen;
 }
 num_to_alloc = (FetArrSize == 0)? 1: FetArrSize;
 if (print)
 {
printf("Fetch Buffer Size %d\n", FetBufSize);
printf("Fetch Array Size:  %d\n", FetArrSize);
 }

 /* set type in sqlvar_struct structure to corresponding C type */
 for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++,
 col_ptr++)
 {
switch(col_ptr->sqltype)
    {
   case SQLCHAR:
       type = "char ";
       col_ptr->sqltype = CCHARTYPE;
       break;
   case SQLINT:
    case SQLSERIAL:
       type = "int ";
       col_ptr->sqltype = CINTTYPE;
       break;
   case SQLBYTES:
   case SQLTEXT:
       if (col_ptr->sqltype == SQLBYTES)
      type = "blob ";
       else
      type = "text ";
       col_ptr->sqltype = CLOCATORTYPE;

       /* Step 3 (TEXT & BLOB only): allocate memory for sqldata
        * that contains ifx_loc_t structures for TEXT or BYTE column */
       temp_loc = (ifx_loc_t *)malloc(col_ptr->sqllen * num_to_alloc);
       if (!temp_loc)
       {
      fprintf(stderr, "blob sqldata malloc failed\n");
      return(-1);
       }
       col_ptr->sqldata = (char *)temp_loc;

       /* Step 4 (TEXT & BLOB only): initialize ifx_loc_t structures to
       hold blob values in a user-defined buffer in memory */
       byfill( (char *)temp_loc, col_ptr->sqllen*num_to_alloc ,0);
       for (j = 0; j< num_to_alloc; j++, temp_loc++)
       {
      /* blob data to go in memory */
      temp_loc->loc_loctype = LOCMEMORY;

          /* assume none of the blobs are larger than BLOBSIZE */
```

```
                temp_loc->loc_bufsize = blobsize;
                temp_loc->loc_buffer = (char *)malloc(blobsize+1);
                if (!temp_loc->loc_buffer)
                {
                    fprintf(stderr, "loc_buffer malloc failed\n");
                    return(-1);
                }
                    temp_loc->loc_oflags = 0; /* clear flag */
                } /* end for */
                break;
            default: /* all other data types */
                fprintf(stderr, "not yet handled(%d)!\n", col_ptr->sqltype);
                return(-1);
            } /* switch */

        /* Step 5: allocate memory for the indicator variable */
        col_ptr->sqlind = (short *)malloc(sizeof(short) * num_to_alloc);
        if (!col_ptr->sqlind)
            {
            printf("indicator malloc failed\n");
            return -1;
            }

        /* Step 6 (other data types): allocate memory for sqldata. This
         * function
         * casts the pointer to this memory as a (char *). Subsequent
         * accesses to the data would need to cast it back to the data
         * type that corresponds to the column type. See the print_sqlda()
         * function for an example of this casting. */
        if (col_ptr->sqltype != CLOCATORTYPE)
        {
            col_ptr->sqldata = (char *) malloc(col_ptr->sqllen *
num_to_alloc);
            if (!col_ptr->sqldata)
            {
                printf("sqldata malloc failed\n");
                return -1;
            }
            if (print)
                printf("column %3d, type = %s(%3d), len=%d\n", i+1, type,
                    col_ptr->sqltype, col_ptr->sqllen);
        }
    }   /* end for */
    return msglen;
}

/************************************************************************
 * Function: print_sqlda
 * Purpose: Prints contents of fetch arrays for each column that the
 * sqlda structure contains. Current version only implements
 * data types found in the blobtab table. Other data types
 * would need to me implemented to make this function complete.
 ************************************************************************/
void print_sqlda(struct sqlda *sqlda, int count)
{
    void *data;
    int i, j;
    ifx_loc_t *temp_loc;
    struct sqlvar_struct *col_ptr;
    char *type;
    char buffer[512];
    int ind;
    char i1, i2;
```

```
 /* print number of columns (sqld) and number of fetch-array elements
 */
 printf("\nsqld: %d, fetch-array elements: %d.\n", sqlda->sqld,
 count);

 /* Outer loop: loop through each element of a fetch array */
 for (j = 0; j < count; j ++)
 {
if (count > 1)
    {
    printf("record[%4d]:\n", j);
    printf("col | type | id | len | ind | rin | data ");
    printf("| value\n");
    printf("-----------------------------------------");
    printf("-----------------\n");
    }

 /* Inner loop: loop through each of the sqlvar_struct structures */
 for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++, col_ptr++)
    {
    data = col_ptr->sqldata + (j*col_ptr->sqllen);
    switch (col_ptr->sqltype)
    {
        case CFIXCHARTYPE:
        case CCHARTYPE:
        type = "char";
        if (col_ptr->sqllen > 40)
           sprintf(buffer, " %39.39s<..", data);
        else
           sprintf(buffer, "%*.*s", col_ptr->sqllen,
                   col_ptr->sqllen, data);
        break;
        case CINTTYPE:
        type = "int";
        sprintf(buffer, " %d", *(int *) data);
        break;
        case CLOCATORTYPE:
        type = "byte";
        temp_loc = (ifx_loc_t *)(col_ptr->sqldata +
                   (j * sizeof(ifx_loc_t)));
        sprintf(buffer, " buf ptr: %p, buf sz: %d, blob sz: %d",
temp_loc->loc_buffer,
                temp_loc->loc_bufsize, temp_loc->loc_size);
        break;
        default:
        type = "??????";
        sprintf(buffer, " type not implemented: ",
                "can't print %d", col_ptr->sqltype);
        break;
    } /* end switch */

    i1 = (col_ptr->sqlind==NULL) ? 'X' :
        (((col_ptr->sqlind)[j] != 0) ? 'T' : 'F');
    i2 = (risnull(col_ptr->sqltype, data)) ? 'T' : 'F';

     printf("%3d | %-6.6s | %3d | %3d | %c | %c | ",
         i, type, col_ptr->sqltype, col_ptr->sqllen, i1, i2);
    printf("%8p |%s\n", data, buffer);
    } /* end for (i=0...) */
  } /* end for (j=0...) */
}
```

```
/***********************************************************************
 * Function: free_sqlda
 * Purpose: Frees memory used by sqlda. This memory includes:
 * o loc_buffer memory (used by TEXT & BYTE)
 * o sqldata memory
 * o sqlda structure
 ***********************************************************************/
void free_sqlda(struct sqlda *sqlda)
{
    int i,j, num_to_dealloc;
    struct sqlvar_struct *col_ptr;
    ifx_loc_t *temp_loc;

    for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++,
    col_ptr++)
    {
    if ( col_ptr->sqltype == CLOCATORTYPE )
        {
        /* Free memory for blob buffer of each element in fetch array */
        num_to_dealloc = (FetArrSize == 0)? 1: FetArrSize;
        temp_loc = (ifx_loc_t *) col_ptr->sqldata;
        for (j = 0; j< num_to_dealloc; j++, temp_loc++)
        {
            free(temp_loc->loc_buffer);
        }
        }
    /* Free memory for sqldata (contains fetch array) */
    free(col_ptr->sqldata);
    }

    /* Free memory for sqlda structure */
    free(sqlda);
}


void main()
{
    int i = 0;
    int row_count, row_size;


     EXEC SQL begin declare section;
    char *db   = "stores7";
    char *uid  = "odbc";
    char *pwd  = "odbc";
     EXEC SQL end declare section;

    /***********************************************************************
     * Step 1: declare an sqlda structure to hold the retrieved column
     * values
     ***********************************************************************/
     struct sqlda *da_ptr;

     EXEC SQL connect to :db user :uid using :pwd;
     if ( SQLCODE < 0 )
     {
    printf("CONNECT failed: %d\n", SQLCODE);
    exit(0);
     }

     /* Prepare the SELECT */
     EXEC SQL prepare selct_id from 'select catalog_num, cat_descr from
     catalog';
     if ( SQLCODE < 0 )
```

```
 {
printf("prepare failed: %d\n", SQLCODE);
exit(0);
 }

 /**********************************************************************
  * Step 2: describe the prepared SELECT statement to allocate memory
  * for the sqlda structure and the sqlda.sqlvar structures
  * (DESCRIBE can allocate sqlda.sqlvar structures because
  * prepared statement is a SELECT)
  **********************************************************************/
 EXEC SQL describe selct_id into da_ptr;
 if ( SQLCODE < 0 )
 {
printf("describe failed: %d\n", SQLCODE);
exit(0);
 }


 /**********************************************************************
  * Step 3: initialize the sqlda structure to hold fetch arrays for
  * columns
  **********************************************************************/
 row_size = init_sqlda(da_ptr, 1);

 /* declare and open a cursor for the prepared SELECT */
 EXEC SQL declare curs cursor for selct_id;
 if ( SQLCODE < 0 )
 {
printf("declare failed: %d\n", SQLCODE);
exit(0);
 }
 EXEC SQL open curs;
 if ( SQLCODE < 0 )
 {
printf("open failed: %d\n", SQLCODE);
exit(0);
 }
 while (1)
 {
/**********************************************************************
 * Step 4: perform fetch to get "FetArrSize" array of rows from
 * the database server into the sqlda structure
 **********************************************************************/
EXEC SQL fetch curs using descriptor da_ptr;

/* Reached last set of matching rows? */
if ( SQLCODE == SQLNOTFOUND )
   break;

/**********************************************************************
 * Step 5: obtain the values from the fetch arrays of the sqlda
 * structure; use sqlca.sqlerrd[2] to determine number
 * of array elements actually retrieved.
 **********************************************************************/
printf("\n===============\n");
printf("FETCH %d\n", i++);
printf("===============");
print_sqlda(da_ptr, ((FetArrSize == 0) ? 1 : sqlca.sqlerrd[2]));

/**********************************************************************
 * Step 6: repeat the FETCH until all rows have been fetched (SQLCODE
 * is SQLNOTFOUND
 **********************************************************************/
```

```
    }

    /*****************************************************************
     * Step 7: Free resources:
     * o statement id, selct_id
     * o select cursor, curs
     * o sqlda structure (with free_sqlda() function)
     * o delete sample table and its rows from database
     ****************************************************************/

    EXEC SQL free selct_id;
    EXEC SQL close curs;
    EXEC SQL free curs;
    free_sqlda(da_ptr);
}
```

Allocating Memory for the Fetch Arrays

The DESCRIBE...INTO statement allocates memory for the **sqlda** structure and its **sqlvar_struct** structures. However, it does not allocate memory for the **sqldata** field of the **sqlvar_struct** structures. The **sqldata** field holds the fetch array for a retrieved column. Therefore, you must allocate sufficient memory to each **sqldata** field to hold the elements of the fetch array.

A new global variable, **FetArrSize**, indicates the number of rows to be returned per FETCH statement. This variable is defined as a C language short integer data type. It has a default value of zero, which disables the fetch array feature. You can set **FetArrSize** to any integer value in the following range:

```
0 <= FetArrSize <= MAXSMINT
```

The MAXSMINT value is the maximum amount of the data type that Informix ESQL/C can retrieve. It's value is 32767 bytes (32 kilobytes). If the size of the fetch array is greater than MAXSMINT, Informix ESQL/C automatically reduces its size to 32 kilobytes.

You can use the following calculation to determine the appropriate size of the fetch array:

```
(fetch-array size) = (fetch-buffer size) / (row size)
```

The preceding equation uses the following information:

fetch-array
size
: The size of the fetch array, which the **FetArrSize** global variable indicates

fetch-buffer
size
: The size of the fetch buffer, which the **FetBufSize** and **BigFetBufSize** global variables indicate. For information about the size of the fetch buffer, see "Optimizing Cursor Execution" on page 14-11.

row size
: The size of the row to be fetched. To determine the size of the row to be fetched, call the **rtypmsize()** function for each column of the row. This function returns the number of bytes that are needed to store the data type. For more information about the **rtypmsize()** function, see Chapter 3, "Informix ESQL/C data types," on page 3-1.

However, if you set **FetArrSize** so that the following relationship is true,

```
(FetArrSize * row size) > FetBufSize
```

Informix ESQL/C automatically adjusts the size of the fetch buffer (**FetBufSize**) as follows to hold the size of the fetch array:

```
FetBufSize = FetArrSize * row size
```

If the result is greater than 32 kilobytes (MAXSMINT), Informix ESQL/C sets **FetBufSize** to 32 kilobytes and **FetArrSize** as follows:

```
FetArrSize = MAXSMINT / (row size)
```

Important: The **FetArrSize** global variable can be used in thread-safe Informix ESQL/C applications.

Follow these steps to allocate memory for a fetch array:

1. Determine the size of the row that you are retrieving from the database.
2. Determine the size of the fetch array and set the **FetArrSize** global variable to this value.
3. For each simple-large-object column (TEXT or BYTE), allocate a fetch array of **ifx_loc_t** structures.
4. For each simple-large-object column (TEXT or BYTE), initialize the **ifx_loc_t** data structures as follows.
   - Set the **loc_loctype** field to LOCMEMORY
   - Set the **loc_buffer** field to the address of the buffer you allocated in Step 3 above.
   - Set the **loc_bufsize** field to the size of the buffer you allocated.

   Alternatively, you can set **loc_bufsize** to **-1** to have Informix ESQL/C automatically allocate memory for the simple-large-object columns. For more information about how to initialize a **ifx_loc_t** structure to retrieve simple large objects in memory, see "Selecting a simple large object into memory" on page 7-9.
5. Allocate memory for the indicator variable.
6. For all other columns, allocate a fetch array that holds the data type of that column.

The following example code illustrates how you would allocate memory for fetch arrays for the following prepared query:

```
SELECT * from blobtab;
```

The function is called **init_sqlda()**:

```
/*************************************************************************
* Function: init_sqlda()
* Purpose: With the sqlda pointer that was returned from the DESCRIBE
* statement, function allocates memory for the fetch arrays
* in the sqldata fields of each column. The function uses
* FetArrSize to determine the size to allocate.
* Returns: < 0 for error
* > 0 error with messagesize
*************************************************************************/
int init_sqlda(struct sqlda *in_da, int print)
{
    int i, j,
   row_size=0,
   msglen=0,
   num_to_alloc;
    struct sqlvar_struct *col_ptr;
```

```
 ifx_loc_t *temp_loc;
 char *type;


 if (print)
printf("columns: %d. \n", in_da->sqld);

 /* Step 1: determine row size */
 for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++,
 col_ptr++)
 {
/* The msglen variable holds the sum of the column sizes in the
 * database; these are the sizes that DESCRIBE returns. This
 * sum is the amount of memory that ESQL/C needs to store
 * one row from the database. This value is <= row_size. */
msglen += col_ptr->sqllen; /* get database sizes */


/* calculate size for C data: string columns get extra byte added
 * to hold null terminator */
col_ptr->sqllen = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);

/* The row_size variable holds the sum of the column sizes in
     * the client application; these are the sizes that rtypmsize()
     * returns. This sum is amount of memory that the client
     * application needs to store one row. */
row_size += col_ptr->sqllen;
if(print)
    printf("Column %d size:  %d\n", i+1, col_ptr->sqllen);
 }

 if (print)
 {
printf("Total message size = %d\n", msglen);
printf("Total row size = %d\n", row_size);
 }

 EXEC SQL select max(length(cat_descr)) into :blobsize from catalog;

 /* Step 2: set FetArrSize global variable to number of elements
  * in fetch array; this function calculates the FetArrSize
  * value that can fit into the existing fetch buffer.
  * If FetBufSize is not set (equals zero), the code assigns a
  * default size of 4096 bytes (4 kilobytes). Alternatively, you
  * could set FetArrSize to the number elements you wanted to
  * have and let ESQL/C size the fetch buffer. See the text in
  * "Allocating Memory for the Fetch Arrays" for more information.*/
 if (FetArrSize <= 0) /* if FetArrSize not yet initialized */
 {
if (FetBufSize == 0) /* if FetBufSize not set */
    FetBufSize = 4096; /* default FetBufSize */
FetArrSize = FetBufSize/msglen;
 }
 num_to_alloc = (FetArrSize == 0)? 1: FetArrSize;
 if (print)
 {
printf("Fetch Buffer Size %d\n", FetBufSize);
printf("Fetch Array Size:  %d\n", FetArrSize);
 }

 /* set type in sqlvar_struct structure to corresponding C type */
 for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++,
 col_ptr++)
 {
```

```
switch(col_ptr->sqltype)
    {
case SQLCHAR:
     type = "char ";
     col_ptr->sqltype = CCHARTYPE;
     break;
case SQLINT:
 case SQLSERIAL:
     type = "int ";
     col_ptr->sqltype = CINTTYPE;
     break;
case SQLBYTES:
case SQLTEXT:
     if (col_ptr->sqltype == SQLBYTES)
   type = "blob ";
     else
   type = "text ";
     col_ptr->sqltype = CLOCATORTYPE;

     /* Step 3 (TEXT & BLOB only): allocate memory for sqldata
      * that contains ifx_loc_t structures for TEXT or BYTE column */
     temp_loc = (ifx_loc_t *)malloc(col_ptr->sqllen * num_to_alloc);
     if (!temp_loc)
     {
   fprintf(stderr, "blob sqldata malloc failed\n");
   return(-1);
     }
     col_ptr->sqldata = (char *)temp_loc;

     /* Step 4 (TEXT & BLOB only): initialize ifx_loc_t structures to
      hold blob values in a user-defined buffer in memory */
     byfill( (char *)temp_loc, col_ptr->sqllen*num_to_alloc ,0);
     for (j = 0; j< num_to_alloc; j++, temp_loc++)
     {
   /* blob data to go in memory */
   temp_loc->loc_loctype = LOCMEMORY;

       /* assume none of the blobs are larger than BLOBSIZE */
   temp_loc->loc_bufsize = blobsize;
   temp_loc->loc_buffer = (char *)malloc(blobsize+1);
   if (!temp_loc->loc_buffer)
   {
      fprintf(stderr, "loc_buffer malloc failed\n");
      return(-1);
       }
       temp_loc->loc_oflags = 0; /* clear flag */
     } /* end for */
     break;
default: /* all other data types */
     fprintf(stderr, "not yet handled(%d)!\n", col_ptr->sqltype);
     return(-1);
    } /* switch */

/* Step 5: allocate memory for the indicator variable */
col_ptr->sqlind = (short *)malloc(sizeof(short) * num_to_alloc);
if (!col_ptr->sqlind)
    {
   printf("indicator malloc failed\n");
   return -1;
    }

/* Step 6 (other data types): allocate memory for sqldata. This function
 * casts the pointer to this memory as a (char *). Subsequent
 * accesses to the data would need to cast it back to the data
```

```
          * type that corresponds to the column type. See the print_sqlda()
          * function for an example of this casting. */
         if (col_ptr->sqltype != CLOCATORTYPE)
         {
            col_ptr->sqldata = (char *) malloc(col_ptr->sqllen *
    num_to_alloc);
            if (!col_ptr->sqldata)
            {
               printf("sqldata malloc failed\n");
               return -1;
            }
            if (print)
               printf("column %3d, type = %s(%3d), len=%d\n", i+1, type,
                      col_ptr->sqltype, col_ptr->sqllen);
         }
      }  /* end for */
      return msglen;
   }
```

For more information about how to allocate memory for the **sqldata** field, see
"Allocating Memory for the sqlda Structure" on page 17-3.

Obtaining Values from Fetch Arrays

Each FETCH attempts to return **FetArrSize** number of values into the sqldata
fields of the **sqlvar_struct** structures of the **sqlda** structure. You can check the
sqlca.sqlerrd[2] value to determine the actual number of rows that the FETCH did
return.

Each fetch array holds the values for one column of the query. To obtain a row of
values, you must access the element at the same index of each the fetch arrays. For
example, to obtain the first row of values, access the first element of each of the
fetch arrays.

The sample program calls the **print_sqlda()** function to obtain values from the
fetch arrays for the following prepared query:

```
SELECT * from blobtab

/*********************************************************************
 * Function: print_sqlda
 * Purpose: Prints contents of fetch arrays for each column that the
 * sqlda structure contains. Current version only implements
 * data types found in the blobtab table. Other data types
 * would need to me implemented to make this function complete.
 ********************************************************************/
void print_sqlda(struct sqlda *sqlda, int count)
{
   void *data;
   int i, j;
   ifx_loc_t *temp_loc;
   struct sqlvar_struct *col_ptr;
   char *type;
   char buffer[512];
   int ind;
   char i1, i2;

   /* print number of columns (sqld) and number of fetch-array elements
   */
   printf("\nsqld: %d, fetch-array elements: %d.\n", sqlda->sqld,
   count);

   /* Outer loop: loop through each element of a fetch array */
```

```
     for (j = 0; j < count; j ++)
     {
   if (count > 1)
      {
      printf("record[%4d]:\n", j);
      printf("col | type | id | len | ind | rin | data ");
      printf("| value\n");
      printf("-------------------------------------------");
      printf("-----------------\n");
      }

   /* Inner loop: loop through each of the sqlvar_struct structures */
   for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++, col_ptr++)
      {
      data = col_ptr->sqldata + (j*col_ptr->sqllen);
      switch (col_ptr->sqltype)
      {
          case CFIXCHARTYPE:
          case CCHARTYPE:
          type = "char";
          if (col_ptr->sqllen > 40)
              sprintf(buffer, " %39.39s<..", data);
          else
              sprintf(buffer, "%*.*s", col_ptr->sqllen,
                     col_ptr->sqllen, data);
          break;
          case CINTTYPE:
          type = "int";
          sprintf(buffer, " %d", *(int *) data);
          break;
          case CLOCATORTYPE:
          type = "byte";
          temp_loc = (ifx_loc_t *)(col_ptr->sqldata +
                     (j * sizeof(ifx_loc_t)));
          sprintf(buffer, " buf ptr: %p, buf sz: %d, blob sz: %d",
temp_loc->loc_buffer,
                     temp_loc->loc_bufsize, temp_loc->loc_size);
          break;
          default:
          type = "??????";
          sprintf(buffer, " type not implemented: ",
                 "can't print %d", col_ptr->sqltype);
          break;
      } /* end switch */

      i1 = (col_ptr->sqlind==NULL) ? 'X' :
          (((col_ptr->sqlind)[j] != 0) ? 'T' : 'F');
      i2 = (risnull(col_ptr->sqltype, data)) ? 'T' : 'F';

       printf("%3d | %-6.6s | %3d | %3d | %c | %c | ",
          i, type, col_ptr->sqltype, col_ptr->sqllen, i1, i2);
      printf("%8p |%s\n", data, buffer);
      } /* end for (i=0...) */
   } /* end for (j=0...) */
}
```

Freeing Memory for a Fetch Array

Informix ESQL/C does not release resources for the **sqlda** structure. When your
application no longer needs the **sqlda** structure, it must free all memory that it
uses. For more information, see "Freeing Memory Allocated to an sqlda Structure"
on page 17-10.

The sample program calls the **free_sqlda()** function to free the memory that the **sqlda** structure uses.

```
/**********************************************************************
* Function: free_sqlda
* Purpose: Frees memory used by sqlda. This memory includes:
* o loc_buffer memory (used by TEXT & BYTE)
* o sqldata memory
* o sqlda structure
**********************************************************************/
void free_sqlda(struct sqlda *sqlda)
{
    int i,j, num_to_dealloc;
    struct sqlvar_struct *col_ptr;
    ifx_loc_t *temp_loc;

    for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++,
    col_ptr++)
    {
    if ( col_ptr->sqltype == CLOCATORTYPE )
      {
      /* Free memory for blob buffer of each element in fetch array */
      num_to_dealloc = (FetArrSize == 0)? 1: FetArrSize;
      temp_loc = (ifx_loc_t *) col_ptr->sqldata;
      for (j = 0; j< num_to_dealloc; j++, temp_loc++)
      {
          free(temp_loc->loc_buffer);
      }
      }
    /* Free memory for sqldata (contains fetch array) */
    free(col_ptr->sqldata);
     }

     /* Free memory for sqlda structure */
     free(sqlda);
}
```

Chapter 16. Using a system-descriptor area

A *system-descriptor area* is a dynamic-management structure that can hold data that a prepared statement either returns from or sends to the database server.

A system-descriptor area is the dynamic-management structure that conforms to X/Open standards.

These topics contain the following information about how to use a system-descriptor area:
- Managing a system-descriptor area for dynamic SQL
- Using a system-descriptor area to handle unknown values in dynamic SQL statements

For information about the fields of the system-descriptor area, see 15-2.

The end of this chapter presents an annotated example program called **dyn_sql**
that uses a system-descriptor area to process a SELECT statement entered at
runtime.

Managing a System-Descriptor Area

Your IBM Informix ESQL/C program can manipulate a system-descriptor area
with the SQL statements that Table 16-1 summarizes.

Table 16-1. SQL Statements That Can Be Used to Manipulate a System-Descriptor Area

SQL Statement	Purpose	See
ALLOCATE DESCRIPTOR	Allocates memory for a system-descriptor area	16-3
DESCRIBE...USING SQL DESCRIPTOR	Initializes the system-descriptor area with information about column-list columns	16-4
GET DESCRIPTOR	Obtains information from the fields of the system-descriptor area	16-5
SET DESCRIPTOR	Places information into a system- descriptor area for the database server to access	16-5
For SELECT and EXECUTE FUNCTION statements that use cursors:		
OPEN...USING SQL DESCRIPTOR	Takes any input parameters from the specified system-descriptor area	16-7
FETCH...USING SQL DESCRIPTOR	Puts the contents of the row into the system-descriptor area	16-7
For SELECT and EXECUTE FUNCTION statements that return only one row:		
EXECUTE...INTO SQL DESCRIPTOR	Puts the contents of the singleton row into the system-descriptor area	16-7
For non-SELECT statements:		
EXECUTE...USING SQL DESCRIPTOR	Takes any input parameters from the specified system-descriptor area	16-7
For an INSERT statement that uses an insert cursor:		
PUT...USING SQL DESCRIPTOR	Puts a row into the insert buffer, obtaining the column values from the specified system-descriptor area	16-18
DEALLOCATE DESCRIPTOR	Frees memory allocated for the system-descriptor area when your program is finished with it	16-8

Allocating Memory for a System-Descriptor Area

To allocate memory for a system-descriptor area, use the ALLOCATE
DESCRIPTOR statement. This statement performs the following tasks:

- It assigns the specified descriptor name to identify this region of memory. This
  name is an identifier that must be provided in all the SQL statements listed in
  Table 16-1 on page 16-3 to designate the system descriptor on which to take
  action.
- It allocates item descriptors. By default, it allocates 100 item descriptors in the
  system-descriptor area. You can change this default with the WITH MAX clause.
- It initializes the COUNT field in the system-descriptor area to the number of
  item descriptors allocated.

Important: ALLOCATE DESCRIPTOR does not allocate memory for column data (DATA field). This memory is allocated by the DESCRIBE statement on an as-needed basis. For more information, see the next section.

Initializing the System-Descriptor Area

The DESCRIBE...USING SQL DESCRIPTOR statement initializes the system-descriptor area with information about the prepared statement. This statement takes the following actions:

- It sets the COUNT field, which contains the number of item descriptors initialized with data.

  This value is the number of columns and expressions in the column list (SELECT and INSERT) or the number of returned values (EXECUTE FUNCTION).

- It describes each unknown column in a prepared SELECT statement (without an INTO TEMP), EXECUTE FUNCTION, or INSERT statement.

  The DESCRIBE statement initializes the fields of the item descriptor for each column, as follows:

  – It allocates memory for the DATA field based on the TYPE and LENGTH information.

  – It initializes the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE fields to provide information from the database about a column.

  For descriptions of these fields, see Table 15-2 on page 15-3.

- It returns the type of SQL statement prepared.

  For more information, see "Determining the Data Type of a Column" on page 15-13.

As noted earlier, the DESCRIBE statement provides information about the columns of a column list. Therefore, you usually use this statement *after* a SELECT (without an INTO TEMP clause), INSERT, or EXECUTE FUNCTION statement was prepared.

The DESCRIBE Statement and Input Parameters

When you use the system-descriptor area to hold an input parameter, you cannot use DESCRIBE to initialize the system-descriptor area. Your code must define the input parameters with the SET DESCRIPTOR statement to explicitly set the appropriate fields of the system-descriptor area. For more information, see "Specifying Input Parameter Values" on page 16-7.

The DESCRIBE Statement and Memory Allocation

When you use a system-descriptor area to hold columns of prepared SQL statements, the ALLOCATE DESCRIPTOR statement allocates memory for the item descriptors of each column and the DESCRIBE...USING SQL DESCRIPTOR statement allocates memory for the DATA field of each item descriptor.

However, the DESCRIBE...USING SQL DESCRIPTOR statement does not allocate memory for the DATA field of a system-descriptor area when you describe a prepared SELECT statement that fetches data from a column into a host variable of type **lvarchar**.

Before you fetch **lvarchar** data into the system-descriptor area, you must explicitly assign memory to the DATA field to hold the column value, as follows:

1. Declare an **lvarchar** host variable of the appropriate size.

   Make sure this variable is not just a pointer but has memory associated with it.

2. Assign this host variable to the DATA field with the SET DESCRIPTOR statement.

   This SET DESCRIPTOR statement should occur after the DESCRIBE...USING SQL DESCRIPTOR statement but before the FETCH...USING SQL DESCRIPTOR statement.
3. Execute the FETCH...USING SQL DESCRIPTOR statement to retrieve the column data into the DATA field of the system-descriptor area.

The following code fragment shows the basic steps to allocate memory for an LVARCHAR column called **lvarch_col** in the **table1** table:

```
EXEC SQL BEGIN DECLARE SECTION;
   lvarchar lvarch_val[50];
   int i;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc';
EXEC SQL prepare stmt1 from 'select opaque_col from table1';
EXEC SQL describe stmt1 using sql descriptor 'desc';
EXEC SQL declare cursor curs1 for stmt1;
EXEC SQL open curs1;
EXEC SQL set descriptor 'desc' value 1
   data = :lvarch_val, length = 50;

while (1)
   {
   EXEC SQL fetch curs1 using sql descriptor 'desc';
   EXEC SQL get descriptor 'desc' value 1 :lvarch_val;
   printf("Column value is %s\n", lvarch_val);
  ⋮

   }
```

The preceding code fragment does not perform exception handling.

Assigning and Obtaining Values from a System-Descriptor Area

The following SQL statements allow your program to access the fields of the system-descriptor area:

- The SET DESCRIPTOR statement assigns values to the fields of the system-descriptor area.
- The GET DESCRIPTOR statement obtains values from the fields of the system-descriptor area.

Using the SET DESCRIPTOR Statement

To assign values to the system-descriptor-area fields, use the SET DESCRIPTOR statement. You can use the SET DESCRIPTOR statement to:

- Set the COUNT field to match the number of items for which you provide descriptions in the system-descriptor area. This value is typically the number of input parameters in a WHERE clause.

  ```
  EXEC SQL set descriptor sysdesc COUNT=:hostvar;
  ```

- Set the item-descriptor fields for each column value for which you provide a description.

  ```
  EXEC SQL set descriptor sysdesc VALUE :item_num
     DESCRIP_FIELD=:hostvar;
  ```

In this example, *item_num* is the number of the item descriptor that corresponds to the desired column, and *DESCRIP_FIELD* is one of the item-descriptor fields that is listed in Table 15-2 on page 15-3.

Set field values to provide values for input parameters in a WHERE clause (16-7) or to modify the contents of an item descriptor field after you use the DESCRIBE...USING SQL DESCRIPTOR statement to fill the system-descriptor area (16-7).

The database server provides data-type constants in the **sqltypes.h** header file to identify the data type of a column in the TYPE field (and optionally the ITYPE field) of a system-descriptor area. However, you cannot assign a data-type constant directly in a SET DESCRIPTOR statement. Instead, assign the constant value to an integer host variable and specify this variable in the SET DESCRIPTOR statement, as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   int i;

   .
   .
   .

EXEC SQL END DECLARE SECTION;

   .
   .
   .

i = SQLINT;
EXEC SQL set descriptor 'desc1' VALUE 1
   TYPE = :i;
```

For more information about the data-type constants, see "Determining the Data Type of a Column" on page 15-13. For more information about how to set individual system-descriptor fields, see the entry for the SET DESCRIPTOR statement in the *IBM Informix Guide to SQL: Syntax*.

Using an lvarchar Pointer Host Variable with a Descriptor: If you use an lvarchar pointer host variable with a FETCH or PUT statement that uses a system descriptor area, you must explicitly set the type to 124 (CLVCHARPTRTYPE from **incl/esql/sqltypes.h**) in the SET DESCRIPTOR statement. The following example illustrates:

```
EXEC SQL BEGIN DECLARE SECTION;
lvarchar *lv;
EXEC SQL END DECLARE SECTION;
   /* where tab has lvarchar * column */
EXEC SQL prepare stmt from "select col from tab";
EXEC SQL allocate descriptor 'd';
   /* The following describe will return SQLLVARCHAR for the
      type of the column */
EXEC SQL describe stmt using sql descriptor 'd';
   /* You must set type for *lv variable */
EXEC SQL set descriptor 'd' value 1 DATA = :lv, TYPE = 124;
EXEC SQL declare c cursor for stmt;
EXEC SQL open c;
EXEC SQL fetch c using sql descriptor 'd';
```

Using the GET DESCRIPTOR Statement

The GET DESCRIPTOR statement obtains values from the system-descriptor-area fields. You can use the GET DESCRIPTOR statement to:

- Get the COUNT field to determine how many values are described in a system-descriptor area.

  ```
  EXEC SQL get descriptor sysdesc :hostvar=COUNT;
  ```

- Get the item-descriptor fields for each described column.

```
EXEC SQL get descriptor sysdesc VALUE :item_num
    :hostvar=DESCRIP_FIELD;
```

In this example, *item_num* is the number of the item descriptor that corresponds to the desired column, and *DESCRIP_FIELD* is one of the item-descriptor fields listed in Table 15-2 on page 15-3.

These item-descriptor values are typically descriptions of columns in a SELECT, INSERT, or EXECUTE FUNCTION statement. GET DESCRIPTOR is also used after a FETCH...USING SQL DESCRIPTOR to copy a column value that is returned by the database server from the system-descriptor area into a host variable (16-7).

The data type of the host variable must be compatible with the type of the associated system-descriptor area field. When you interpret the TYPE field, make sure that you use the data type values that match your environment. For some data types, X/Open values differ from Informix values. For more information, see "Determining the Data Type of a Column" on page 15-13.

For more information about how to get individual system-descriptor fields, see the entry for the GET DESCRIPTOR statement in the *IBM Informix Guide to SQL: Syntax*.

Specifying Input Parameter Values

Because the DESCRIBE...USING SQL DESCRIPTOR statement does not analyze a WHERE clause, your program must store the number, data types, and values of the input parameters in the fields of the system-descriptor area to explicitly describe these parameters. For general information about how to define input parameters dynamically, see 15-17.

When you execute a parameterized statement, you must specify the system-descriptor area as the location of input parameter values with the USING SQL DESCRIPTOR clause, as follows:

- For input parameters in the WHERE clause of a SELECT, use the OPEN...USING SQL DESCRIPTOR statement. This statement handles a sequential, scrolling, hold, or update cursor. If you are certain that the SELECT will return only *one* row, you can use the EXECUTE...INTO...USING SQL DESCRIPTOR statement instead of a cursor. See "Handling an Unknown Select List" on page 16-9.
- For input parameters in the WHERE clause of a non-SELECT statement such as DELETE or UPDATE, use the EXECUTE...USING SQL DESCRIPTOR statement. See "Handling a Parameterized UPDATE or DELETE Statement" on page 16-28.
- For input parameters in the VALUES clause of an INSERT statement, use the EXECUTE...USING SQL DESCRIPTOR statement. If the INSERT statement is associated with an insert cursor, use the PUT...USING SQL DESCRIPTOR statement instead. For more information, see "Handling an unknown column list" on page 16-18.

Putting Column Values into a System-Descriptor Area

When you create a SELECT statement dynamically, you cannot use the INTO *host_var* clause of FETCH because you cannot name the host variables in the prepared statement. To fetch column values into a system-descriptor area, use the USING SQL DESCRIPTOR clause of FETCH instead of the INTO clause. The FETCH...USING SQL DESCRIPTOR statement puts each column value into the DATA field of its item descriptor.

Use of the FETCH...USING SQL DESCRIPTOR statement assumes the existence of a cursor associated with the prepared statement. You must *always* use a cursor for SELECT statements and cursor functions (EXECUTE FUNCTION statements that return multiple rows). However, if the SELECT (or EXECUTE FUNCTION) returns only one row, you can omit the cursor and retrieve the column values into a system-descriptor area with the EXECUTE...INTO SQL DESCRIPTOR statement.

Warning: If you execute a SELECT statement or user-defined function that returns more than one row and do not associate the statement with a cursor, your program generates a runtime error. When you associate a singleton SELECT (or EXECUTE FUNCTION) statement with a cursor, Informix ESQL/C does not generate an error. Therefore, it is a good practice to always associate a dynamic SELECT or EXECUTE FUNCTION statement with a cursor and to use a FETCH...USING SQL DESCRIPTOR statement to retrieve the column values from this cursor into the system-descriptor area.

Once the column values are in the system-descriptor area, you can use the GET DESCRIPTOR statement to transfer these values from their DATA fields to the appropriate host variables. You must use the LENGTH and TYPE fields to determine, at runtime, the data types for these host variables. You might need to perform data type or length conversions between the SQL data types in the TYPE fields and the Informix ESQL/C data types that are needed for host variables that hold the return value.

For more information about how to execute SELECT statements dynamically, see "Handling an Unknown Select List" on page 16-9. For more information about how to execute user-defined functions dynamically, see "Handling Unknown Return Values" on page 16-13.

Freeing Memory Allocated to a System-Descriptor Area

The DEALLOCATE DESCRIPTOR statement deallocates, or frees, memory that the specified system-descriptor area uses. The freed memory includes memory used by the item descriptors to hold data (in the DATA fields). Make sure you deallocate a system-descriptor area only after you no longer have need of it. A deallocated system-descriptor area cannot be reused.

For information about how to free other program resources, see "Freeing Resources" on page 14-7. For more information about DEALLOCATE DESCRIPTOR, see the *IBM Informix Guide to SQL: Syntax*.

Using a System-Descriptor Area

Use a system-descriptor area to execute SQL statements that contain unknown values. Table 16-2 summarizes the types of dynamic statements that the remaining sections of this chapter cover.

Table 16-2. Using a System-Descriptor Area to Execute Dynamic SQL Statements

Purpose of a System-Descriptor Area	See
Holds select-list column values retrieved by a SELECT statement	16-9
Holds returned values from user-defined functions	16-13
Describes unknown columns in an INSERT statement	16-18
Describes input parameters in the WHERE clause of a SELECT statement	16-22

Purpose of a System-Descriptor Area	See
Describes input parameters in the WHERE clause of a DELETE or UPDATE statement	16-28

Handling an Unknown Select List

For an introduction to how to handle unknown columns in an unknown select list, see 15-16. This section describes how to use a system-descriptor area to handle a SELECT statement.

To use a system-descriptor area to handle unknown select-list columns:

1. Prepare the SELECT statement (with the PREPARE statement) to give it a statement identifier. The SELECT statement cannot include an INTO TEMP clause. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.
2. Allocate a system-descriptor area with the ALLOCATE DESCRIPTOR statement. For more information, see "Allocating Memory for a System-Descriptor Area" on page 16-3.
3. Determine the number and data types of the select-list columns with the DESCRIBE...USING SQL DESCRIPTOR statement. DESCRIBE fills an item descriptor for each column in the select list. For more information about DESCRIBE, see "Initializing the System-Descriptor Area" on page 16-4.
4. Save the number of select-list columns in a host variable with the GET DESCRIPTOR statement to obtain the value of the COUNT field.
5. Declare and open a cursor and then use the FETCH...USING SQL DESCRIPTOR statement to fetch column values, one row at a time, into an allocated system-descriptor area. See "Putting Column Values into a System-Descriptor Area" on page 16-7.
6. Retrieve the row data from the system-descriptor area into host variables with the GET DESCRIPTOR statement to access the DATA field. For more information about GET DESCRIPTOR, see "Assigning and Obtaining Values from a System-Descriptor Area" on page 16-5.
7. Deallocate the system-descriptor area with the DEALLOCATE DESCRIPTOR statement. For more information, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Important: If the SELECT statement has unknown input parameters in the WHERE clause, your program must also handle these input parameters with a system-descriptor area.

For more information, see "Handling a Parameterized SELECT Statement" on page 16-22.

The following sections demonstrate how to handle a SELECT statement that returns many rows and one that returns only one row.

Executing a SELECT That Returns Multiple Rows

The **demo4.ec** sample program, shown in this section, shows how to execute a dynamic SELECT statement with the following conditions:

- The SELECT returns more than one row.

The SELECT must be associated with a cursor, executed with the OPEN statement, and have its return values retrieved with the FETCH...USING SQL DESCRIPTOR statement.

- The SELECT has either no input parameters or no WHERE clause.

  The OPEN statement does not need to include the USING clause.

- The SELECT has unknown columns in its select list.

  The FETCH statement includes the USING SQL DESCRIPTOR clause to store the return values in an **sqlda** structure.

This **demo4** program is a version of the **demo3** sample program ("Executing a SELECT That Returns Multiple Rows" on page 17-11) that uses a system-descriptor area to hold select-list columns. The **demo4** program does *not* include exception handling.

```
========================================================================
 1.  #include <stdio.h>
 2.  EXEC SQL define NAME_LEN        15;
 3.  main()
 4.  {
 5.  EXEC SQL BEGIN DECLARE SECTION;
 6.     mint i;
 7.     mint desc_count;
 8.     char demoquery[80];
 9.     char colname[19];
10.     char result[ NAME_LEN + 1 ];
11.  EXEC SQL END DECLARE SECTION;
========================================================================
```

Lines 5 to 11

These lines declare host variables to hold the data that is obtained from the user and the column values that are retrieved from the system-descriptor area.

```
========================================================================
12.     printf("DEMO4 Sample ESQL program running.\n\n");
13.     EXEC SQL connect to 'stores7';
14.  /* These next three lines have hard-wired both the query and
15.      * the value for the parameter. This information could have been
16.      * been entered from the terminal and placed into the strings
17.      * demoquery and the query value string (queryvalue),
         * respectively.
18.      */
19.     sprintf(demoquery, "%s %s",
20.        "select fname, lname from customer",
21.        "where lname < 'C' ");
22.     EXEC SQL prepare demo4id from :demoquery;
23.     EXEC SQL declare demo4cursor cursor for demo4id;
24.     EXEC SQL allocate descriptor 'demo4desc' with max 4;
25.     EXEC SQL open demo4cursor;
========================================================================
```

Lines 14 to 22

These lines assemble the character string for the statement (in **demoquery**) and prepare it as the **demo4id** statement identifier. For more information about these steps, see "Assembling and Preparing the SQL Statement" on page 14-2.

Line 23

This line declares the **demo4cursor** cursor for the prepared statement identifier, **demo4id**. All non-singleton SELECT statements *must* have a declared cursor.

Line 24

To be able to use a system-descriptor area for the select-list columns, you must first allocate it. This ALLOCATE DESCRIPTOR statement allocates the **demo4desc** system-descriptor area with four item descriptors.

Line 25

The database server executes the SELECT statement when it opens the **demo4cursor** cursor. If the WHERE clause of your SELECT statement contains input parameters, you also need to specify the USING SQL DESCRIPTOR clause of the OPEN statement. (See "Handling a Parameterized SELECT Statement" on page 16-22.)

```
=========================================================================
26.     EXEC SQL describe demo4id using sql descriptor 'demo4desc';
27.     EXEC SQL get descriptor 'demo4desc' :desc_count = COUNT;
28.     printf("There are %d returned columns:\n", desc_count);
29.     /* Print out what DESCRIBE returns */
30.     for (i = 1; i <= desc_count; i++)
31.         prsysdesc(i);
32.     printf("\n\n");
=========================================================================
```

Line 26

The DESCRIBE statement describes the select-list columns for the prepared statement in the **demo4id** statement identifier. For this reason, the DESCRIBE must follow the PREPARE. This DESCRIBE includes the USING SQL DESCRIPTOR clause to specify the **demo4desc** system-descriptor area as the location for these column descriptions.

Lines 27 and 28

Line 27 uses the GET DESCRIPTOR statement to obtain the number of select-list columns found by the DESCRIBE. This number is read from the COUNT field of the **demo4desc** system-descriptor area and saved in the **desc_count** host variable. Line 28 displays this information to the user.

Lines 29 to 31

This **for** loop goes through the item descriptors for the columns of the select list. It uses the **desc_count** host variable to determine the number of item descriptors initialized by the DESCRIBE statement. For each item descriptor, the **for** loop calls the **prsysdesc()** function (line 31) to save information such as the data type, length, and name of the column in host variables. See lines 58 to 77 for a description of **prsysdesc()**.

```
=========================================================================
33.     for (;;)
34.         {
35.         EXEC SQL fetch demo4cursor using sql descriptor 'demo4desc';
36.         if (strncmp(SQLSTATE, "00", 2) != 0)
37.             break;
38.         /* Print out the returned values */
39.         for (i = 1; i <= desc_count; i++)
40.             {
41.             EXEC SQL get descriptor 'demo4desc' VALUE :i
42.                 :colname=NAME, :result = DATA;
```

```
43.          printf("Column: %s\tValue:%s\n ", colname, result);
44.              }
45.      printf("\n");
46.          }
```

===

Lines 33 to 46

This inner **for** loop executes for each row fetched from the database. The FETCH
statement (line 35) includes the USING SQL DESCRIPTOR clause to specify the
demo4desc system-descriptor area as the location of the column values. After this
FETCH executes, the column values are stored in the specified system-descriptor
area.

The **if** statement (lines 36 and 37) tests the value of the SQLSTATE variable to
determine if the FETCH was successful. If SQLSTATE contains a class code other
than "00", then the FETCH generates a warning ("01"), the NOT FOUND
condition ("02"), or an error (> "02"). In any of these cases, line 37 ends the **for**
loop.

Lines 39 to 45 access the fields of the item descriptor for each column in the select
list. After each FETCH statement, the GET DESCRIPTOR statement (lines 41 and
42) loads the contents of the DATA field into a host variable of the appropriate
type and length. The second **for** loop (lines 39 to 44) ensures that GET
DESCRIPTOR is called for *each* column in the select list.

Important: In this GET DESCRIPTOR statement, the demo4 program assumes that
the returned columns are of the CHAR data type. If the program did
not make this assumption, it would need to check the TYPE and
LENGTH fields to determine the appropriate data type for the host
variable to hold the DATA value.

===
```
47.      if(strncmp(SQLSTATE, "02", 2) != 0)
48.          printf("SQLSTATE after fetch is %s\n", SQLSTATE);
49.      EXEC SQL close demo4cursor;
50.      /* free resources for prepared statement and cursor*/
51.      EXEC SQL free demo4id;
52.      EXEC SQL free demo4cursor;
53.      /* free system-descriptor area */
54.      EXEC SQL deallocate descriptor 'demo4desc';
55.      EXEC SQL disconnect current;
56.      printf("\nDEMO4 Sample Program Over.\n\n");
57. }
```

===

Lines 47 and 48

Outside the **for** loop, the program tests the SQLSTATE variable again so that it can
notify the user in the event of successful execution, a runtime error, or a warning
(class code not equal to "02").

Line 49

After all the rows are fetched, the CLOSE statement closes the **demo4cursor** cursor.

Lines 50 to 54

These FREE statements release the resources that are allocated for the prepared
statement (line 51) and the database cursor (line 52).

The DEALLOCATE DESCRIPTOR statement (line 54) releases the memory allocated to the **demo4desc** system-descriptor area. For more information, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

```
=======================================================================
58. prsysdesc(index)
59. EXEC SQL BEGIN DECLARE SECTION;
60.    PARAMETER mint index;
61. EXEC SQL END DECLARE SECTION;
62. {
63.    EXEC SQL BEGIN DECLARE SECTION;
64.       mint type;
65.       mint len;
66.       mint nullable;
67.       char name[40];
68.    EXEC SQL END DECLARE SECTION;
69.    EXEC SQL get descriptor 'demo4desc' VALUE :index
70.       :type = TYPE,
71.       :len = LENGTH,
72.       :nullable = NULLABLE,
73.       :name = NAME;
74.    printf("    Column %d: type = %d, len = %d, nullable=%d, name =
       %s\n",
75.       index, type, len, nullable, name);
76. }

=======================================================================
```

Lines 58 to 77

The **prsysdesc()** function displays information about a select-list column. It uses the GET DESCRIPTOR statement to access one item descriptor from the **demo4desc** system-descriptor area.

The GET DESCRIPTOR statement (lines 70 to 74) accesses the TYPE, LENGTH, NULLABLE, and NAME fields from an item descriptor in **demo4desc** to provide information about a column. It stores this information in host variables of appropriate lengths and data types. The VALUE keyword indicates the number of the item descriptor to access.

Executing a Singleton SELECT

The **demo4** program, described in the previous section, assumes that the SELECT statement returns more than one row and therefore the program associates the statement with a cursor. If you know at the time that you write the program that the dynamic SELECT *always* returns just one row, you can omit the cursor and use the EXECUTE...INTO SQL DESCRIPTOR statement instead of the FETCH...USING SQL DESCRIPTOR. You need to use the DESCRIBE statement to define the select-list columns. For more information, see "Putting Column Values into a System-Descriptor Area" on page 16-7.

Handling Unknown Return Values

For an introduction on how to handle unknown return values from a user-defined function, see 15-18. This section describes how to use a system-descriptor area to save values that a dynamically executed user-defined function returns.

To use a system-descriptor area to handle unknown function return values:

1. Assemble and prepare an EXECUTE FUNCTION statement.

The EXECUTE FUNCTION statement cannot include an INTO clause. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.

2. Allocate a system-descriptor area with the ALLOCATE DESCRIPTOR statement.

   For more information, see "Allocating Memory for a System-Descriptor Area" on page 16-3.

3. Determine the number and data type(s) of the return value(s) with the DESCRIBE...USING SQL DESCRIPTOR statement.

   The DESCRIBE...USING SQL DESCRIPTOR statement fills an item descriptor for each value that the user-defined function returns. For more information about DESCRIBE, see "Initializing the System-Descriptor Area" on page 16-4.

4. After the DESCRIBE statement, you can test the SQLCODE variable (**sqlca.sqlcode**) for the SQ_EXECPROC defined constant to check for a prepared EXECUTE FUNCTION statement.

   This constant is defined in the **sqlstype.h** header file. For more information, see "Determining the Statement Type" on page 15-9.

5. Execute the EXECUTE FUNCTION statement and store the return values in the system-descriptor area.

   The statement you use to execute a user-defined function depends on whether the function is a noncursor function or a cursor function. The following sections discuss how to execute each type of function.

6. Deallocate the system-descriptor area with the DEALLOCATE DESCRIPTOR statement.

   See "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Executing a noncursor function

A noncursor function returns only one row of return values to the application. Use the EXECUTE...INTO SQL DESCRIPTOR statement to execute the function and save the return value(s) in a system-descriptor area.

An external function that is not explicitly defined as an iterator function returns only a single row of data. Therefore, you can use EXECUTE...INTO SQL DESCRIPTOR to execute most external functions dynamically. This single row of data consists of only one value because external function can only return a single value. The system-descriptor area contains only one item descriptor with the single return value.

An SPL function whose RETURN statement does not include the WITH RESUME keywords returns only a single row of data. Therefore, you can use EXECUTE...INTO SQL DESCRIPTOR to execute most SPL functions dynamically. An SPL function can return one or more values at one time so the system-descriptor area contains one or more item descriptors.

Warning: Because you usually do not know the number of returned rows that a user-defined function returns, you cannot guarantee that only one row will be returned. If you do not use a cursor to execute cursor function, Informix ESQL/C generates a runtime error. Therefore, it is a good practice to always associate a user-defined function with a function cursor.

The following program fragment dynamically executes an SPL function called **items_pct**. This SPL function calculates the percentage that the items of a given manufacturer represent out of the total price of all items in the **items** table. It

accepts one argument, the **manu_code** value for the chosen manufacturer, and it returns the percentage as a decimal value. The following figure shows the **items_pct** SPL function.

```
create function items_pct(mac char(3)) returning decimal;
   define tp money;
   define mc_tot money;
   define pct decimal;
   let tp = (select sum(total_price) from items);
   let mc_tot = (select sum(total_price) from items
         where manu_code = mac);
   let pct = mc_tot / tp;
   return pct;
end function;
```

Figure 16-1. Code for items_pct SPL function

The following program fragment uses a system-descriptor area to dynamically execute an SPL function that returns more than one set of return values.

```
=========================================================================
 1. #include <stdio.h>
 2. #include <ctype.h>
 3. EXEC SQL include sqltypes;
 4. EXEC SQL include sqlstype;
 5. EXEC SQL include decimal;
 6. EXEC SQL include datetime;
 7. extern char statement[80];
 8. main()
 9. {
10. EXEC SQL BEGIN DECLARE SECTION;
11.    int sp_cnt, desc_count;
12.    char dyn_stmnt[80], rout_name[30];
13. EXEC SQL END DECLARE SECTION;
14. int whenexp_chk();
15.    printf("Sample ESQL program to execute an SPL function
       running.\n\n");
16.    EXEC SQL whenever sqlerror call whenexp_chk;
17.    EXEC SQL connect to 'stores7';
18.    printf("Connected to stores7 database.\n");
19.    /* These next five lines hard-wire the execute function
20.     * statement.  This information could have been entered
21.     * by the user and placed into the string dyn_stmnt.
22.     */
23.    stcopy("items_pct(\"HSK\")", rout_name);
24.    sprintf(dyn_stmnt, "%s %s",
25.          "execute function", rout_name);

=========================================================================
```

Lines 19 to 25
The call to **sprintf()** (line 24) assembles the character string for the EXECUTE FUNCTION statement that executes the **items_pct()** SPL function.

```
=========================================================================
26.    EXEC SQL prepare spid from :dyn_stmnt;
27.    EXEC SQL allocate descriptor 'spdesc';
28.    EXEC SQL describe spid using sql descriptor 'spdesc';
29.    if(SQLCODE != SQ_EXECPROC)
30.       {
```

```
31.        printf("\nPrepared statement is not EXECUTE FUNCTION.\n");
32.        exit();
33.        }
```

==

Line 26

The PREPARE statement then creates the **spid** statement identifier for the
EXECUTE FUNCTION statement. For more information about these steps, see
"Assembling and Preparing the SQL Statement" on page 14-2.

Line 27

The ALLOCATE DESCRIPTOR statement allocates the **spdesc** system-descriptor
area. For more information, see "Allocating Memory for a System-Descriptor Area"
on page 16-3.

Lines 28 to 33

The DESCRIBE statement determines the number and data types of values that the
items_pct SPL function returns. This DESCRIBE includes the USING SQL
DESCRIPTOR clause to specify the **spdesc** system-descriptor area as the location
for these descriptions.

On line 28, the program tests the value of the SQLCODE variable (**sqlca.sqlcode**)
against the constant values defined in the **sqlstype.h** file to verify that the
EXECUTE FUNCTION statement was prepared. (For more information, see
"Determining the Statement Type" on page 15-9.)

==
```
34.    EXEC SQL get descriptor 'spdesc' :sp_cnt = COUNT;
35.    if(sp_cnt == 0)
36.        {
37.        sprintf(dyn_stmnt, "%s %s", "execute procedure", rout_name);
38.        EXEC SQL prepare spid from :dyn_stmnt;
39.        EXEC SQL execute spid;
40.        }
41.    else
42.        {
43.        EXEC SQL declare sp_curs cursor for spid;
44.        EXEC SQL open sp_curs;
45.        while(getrow("spdesc"))
46.            disp_data(:sp_cnt, "spdesc");
47.        EXEC SQL close sp_curs;
48.        EXEC SQL free sp_curs;
49.        }
```

==

Lines 34 to 40

To obtain the number of return values in a host variable, the GET DESCRIPTOR
statement retrieves the value of the COUNT field into a host variable. This value is
useful when you need to determine how many values the SPL routine returns. If
the SPL routine does *not* return values, that is, the value of COUNT is zero, the
SPL routine is a procedure, not a function. Therefore, the program prepares an
EXECUTE PROCEDURE statement (line 38) and then uses the EXECUTE statement
(line 39) to execute the procedure. The EXECUTE statement does not need to use
the system-descriptor area because the SPL procedure does not have any return
values.

Lines 41 to 49

If the SPL routine *does* return values, that is, if the value of COUNT is greater than zero, the program declares and opens the **sp_curs** cursor for the prepared SPL function.

A **while** loop (lines 45 and 46) executes for each set of values that is returned by the SPL function. This loop calls the **getrow()** function to fetch one set of values into the **spdesc** system-descriptor area. It then calls the **disp_data()** function to display the returned values. For descriptions of the **getrow()** and **disp_data()** functions, see "Guide to the dyn_sql.ec File" on page 16-28.

After all the sets of return values are returned, the CLOSE statement (line 47) closes the **sp_curs** cursor and the FREE statement (line 48) releases the resources allocated to the cursor.

```
=====================================================================
50.        EXEC SQL free spid;
51.        EXEC SQL deallocate descriptor 'spdesc';
52.        EXEC SQL disconnect current;
53. }

=====================================================================
```

Lines 50 and 51

This FREE statement releases the resources allocated for the prepared statement. The DEALLOCATE DESCRIPTOR statement releases the memory allocated to the **spdesc** system-descriptor area. For more information, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Executing a Cursor Function

A cursor function can return one or more rows of return values to the application. To execute a cursor function, you must associate the EXECUTE FUNCTION statement with a function cursor and use the FETCH...INTO SQL DESCRIPTOR statement to save the return value(s) in a system-descriptor area.

To use a system-descriptor area to hold cursor-function return values:

1. Declare a function cursor for the user-defined function.

   Use the DECLARE statement to associate the EXECUTE FUNCTION statement with a function cursor.

2. Use the OPEN statement to execute the function and open the cursor.

3. Use the FETCH...USING SQL DESCRIPTOR statement to retrieve the return values from the cursor into the system-descriptor area.

   For more information, see "Putting Column Values into a System-Descriptor Area" on page 16-7.

4. Use the GET DESCRIPTOR statement to retrieve the return values from the system-descriptor area into host variables

   The DATA field of each item descriptor contains the return values. For more information, see "Assigning and Obtaining Values from a System-Descriptor Area" on page 16-5.

5. Deallocate the system-descriptor area with the DEALLOCATE DESCRIPTOR statement.

   For more information, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Only an external function that is defined as an iterator function can return more than one row of data. Therefore, you must define a function cursor to execute an iterator function dynamically. Each row of data consists of only one value because an external function can only return a single value. For each row, the system-descriptor area contains only one item descriptor with the single return value.

An SPL function whose RETURN statement includes the WITH RESUME keywords can return one or more rows of data. Therefore, you must define a function cursor to execute these SPL functions dynamically. Each row of data can consists of one or more values because an SPL function can return one or more values at one time. For each row, the system-descriptor area contains an item descriptor for each return value.

Handling an unknown column list

For an introduction on how to handle columns in a VALUES clause of an INSERT, see 15-17. This section describes how to use a system-descriptor area to handle the INSERT...VALUES statement.

To use a system-descriptor area to handle input parameters in an INSERT:

1. Prepare the INSERT statement (with the PREPARE statement) to give it a statement identifier. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.
2. Allocate a system-descriptor area with the ALLOCATE DESCRIPTOR statement. For more information, see "Allocating Memory for a System-Descriptor Area" on page 16-3.
3. Determine the number and data types of the columns with the DESCRIBE...USING SQL DESCRIPTOR statement. The DESCRIBE statement fills an item descriptor for each column in the select list. For more information about DESCRIBE, see "Initializing the System-Descriptor Area" on page 16-4.
4. Save the number of unknown columns in a host variable with the GET DESCRIPTOR statement, which obtains the value of the COUNT field.
5. Set the columns to their values with the SET DESCRIPTOR statement, which sets the appropriate DATA and VALUE fields. The column values must be compatible with the data type of their associated column. If you want to insert a NULL value, set the INDICATOR field to -1, and do not specify any DATA field in the SET DESCRIPTOR statement. For more information about SET DESCRIPTOR, see "Assigning and Obtaining Values from a System-Descriptor Area" on page 16-5.
6. Execute the INSERT statement to insert the values into the database.

   The following sections demonstrate how to execute a simple INSERT statement that inserts only one row and one that uses an insert cursor to insert several rows from an insert buffer.
7. Deallocate the system-descriptor area with the DEALLOCATE DESCRIPTOR statement. See "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Executing a Simple Insert

The following steps outline how to execute a simple INSERT statement with a system-descriptor area:

1. Prepare the INSERT statement (with the PREPARE statement) and give it a statement identifier.

2. Set the columns to their values with the SET DESCRIPTOR statement.
3. Execute the INSERT statement with the EXECUTE...USING SQL DESCRIPTOR statement.

The sample program described on the following pages shows how to execute a dynamic INSERT statement. The program inserts two TEXT values into the **txt_a** table. It reads the text values from a named file called **desc_ins.txt**. The program then selects columns from this table and stores the TEXT values in two named files, **txt_out1** and **txt_out2**. The program illustrates the use of a system-descriptor area to handle the columns that are in the column list. This INSERT statement is *not* associated with an insert cursor.

```
========================================================================
1.  EXEC SQL include locator;
2.  EXEC SQL include sqltypes;
3.  main()
4.  {
5.     EXEC SQL BEGIN DECLARE SECTION;
6.         int i;
7.         int cnt;
8.         ifx_loc_t loc1;
9.         ifx_loc_t loc2;
10.    EXEC SQL END DECLARE SECTION;
11.    EXEC SQL create database txt_test;
12.    chkerr("CREATE DATABASE txt_test");
13.    EXEC SQL create table txt_a (t1 text not null, t2 text);
14.    chkerr("CREATE TABLE t1");
15.    /* The INSERT statement could have been created at runtime. */
16.    EXEC SQL prepare sid from 'insert into txt_a values (?, ?)';
17.    chkerr("PREPARE sid");
========================================================================
```

Lines 5 to 10
These lines declare host variables to hold the column values to insert (obtained from the user).

Lines 15 to 17
These lines assemble the character string for the statement and prepare it as the **sid** statement identifier. The input parameter specifies the missing columns of the INSERT statement. The INSERT statement is hard coded here, but it can be created at runtime. For more information about these steps, see "Assembling and Preparing the SQL Statement" on page 14-2.

```
========================================================================
18.    EXEC SQL allocate descriptor 'desc';
19.    chkerr("ALLOCATE DESCRIPTOR desc");
20.    EXEC SQL describe sid using sql descriptor 'desc';
21.    chkerr("DESCRIBE sid");
22.    EXEC SQL get descriptor 'desc' :cnt = COUNT;
23.    chkerr("GET DESCRIPTOR desc");
24.    for (i = 1; i <= cnt; i++)
25.        prsysdesc(i);
========================================================================
```

Lines 18 and 19

To be able to use a system-descriptor area for the columns, you must first allocate the system-descriptor area. This ALLOCATE DESCRIPTOR statement allocates a system-descriptor area named **desc**.

Line 20 and 21

The DESCRIBE statement describes the columns for the prepared INSERT that **sid** identifies. This DESCRIBE statement includes the USING SQL DESCRIPTOR clause to specify the **desc** system-descriptor area as the location for these column descriptions.

Lines 22 and 23

The GET DESCRIPTOR statement obtains the number of columns (COUNT field) found by the DESCRIBE. This number is stored in the **cnt** host variable.

Lines 24 and 25

This **for** loop goes through the item descriptors for the columns of the INSERT statement. It uses the **cnt** variable to determine the number of item descriptors that are initialized by the DESCRIBE. For each item descriptor, the **prsysdesc()** function saves information such as the data type, length, and name in host variables. For a description of **prsysdesc()**, see lines 58 to 77 in 16-13.

```
=======================================================================
26.    loc1.loc_loctype = loc2.loc_loctype = LOCFNAME;
27.    loc1.loc_fname = loc2.loc_fname = "desc_ins.txt";
28.    loc1.loc_size = loc2.loc_size = -1;
29.    loc1.loc_oflags = LOC_RONLY;
30.    i = CLOCATORTYPE;
31.    EXEC SQL set descriptor 'desc' VALUE 1
32.       TYPE = :i, DATA = :loc1;
33.    chkerr("SET DESCRIPTOR 1");
34.    EXEC SQL set descriptor 'desc' VALUE 2
35.       TYPE = :i, DATA = :loc2;
36.    chkerr("SET DESCRIPTOR 2");
37.    EXEC SQL execute sid using sql descriptor 'desc';
38.    chkerr("EXECUTE sid");

=======================================================================
```

Lines 26 to 29

To insert a TEXT value, the program must first locate the value with an Informix ESQL/C locator structure. The **loc1** locator structure stores a TEXT value for the **t1** column of the **txt_a** table; **loc2** is the locator structure for the **t2** column of **txt_a**. (See line 13.) The program includes the Informix ESQL/C **locator.h** header file (line 1) to define the **ifx_loc_t** structure.

Both TEXT values are located in a named file (**loc_loctype** = LOCFNAME) called **desc_ins.txt**. When you set the **loc_size** fields to **-1**, the locator structure tells Informix ESQL/C to send the TEXT value to the database server in a single operation. For more information about how to locate TEXT values in named files, see "Inserting a Simple Large Object from a Named File" on page 7-20.

Lines 30 to 36

The first SET DESCRIPTOR statement sets the TYPE and DATA fields in the item descriptor of the **t1** column (VALUE 1). The data type is CLOCATORTYPE (defined in the Informix ESQL/C **sqltypes.h** header file) to indicate that the column value is stored in an Informix ESQL/C locator structure; the data is set to the **loc1** locator

structure. The second SET DESCRIPTOR statement performs this same task for the **t2** column value; it sets its DATA field to the **loc2** locator structure.

Lines 37 and 38
The database server executes the INSERT statement with the EXECUTE...USING SQL DESCRIPTOR statement to obtain the new column values from the **desc** system-descriptor area.

```
==========================================================================
39.     loc1.loc_loctype = loc2.loc_loctype = LOCFNAME;
40.     loc1.loc_fname = "txt_out1";
41.     loc2.loc_fname = "txt_out2";
42.     loc1.loc_oflags = loc2.loc_oflags = LOC_WONLY;
43.     EXEC SQL select * into :loc1, :loc2 from a;
44.     chkerr("SELECT");
45.     EXEC SQL free sid;
46.     chkerr("FREE sid");
47.     EXEC SQL deallocate descriptor 'desc';
48.     chkerr("DEALLOCATE DESCRIPTOR desc");
49.     EXEC SQL close database;
50.     chkerr("CLOSE DATABASE txt_test");
51.     EXEC SQL drop database txt_test;
52.     chkerr("DROP DATABASE txt_test");
53.     EXEC SQL disconnect current;
54.  }
55.  chkerr(s)
56.  char *s;
57.  {
58.  if (SQLCODE)
59.     printf("%s error %d\n", s, SQLCODE);
60.  }
==========================================================================
```

Lines 39 to 44
The program uses the **loc1** and **loc2** locator structures to select the values just inserted. These TEXT values are read into named files: the **t1** column (in **loc1**) into **txt_out1** and the **t2** column (in **loc2**) into **txt_out2**. The **loc_oflags** value of LOC_WONLY means that this TEXT data will overwrite any existing data in these output files.

Lines 45 to 48
The FREE statement (line 45) releases the resources allocated for the **sid** prepared statement. Once a prepared statement was freed, it cannot be used again in the program. The DEALLOCATE DESCRIPTOR statement (line 46) releases the memory allocated to the **desc** system-descriptor area. For more information, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Lines 55 to 60
The **chkerr()** function is a very simple exception-handling routine. It checks the global SQLCODE variable for a nonzero value. Since zero indicates successful execution of an SQL statement, the **printf()** (line 58) executes whenever a runtime error occurs. For more detailed exception-handling routines, see Chapter 11, "Exception handling," on page 11-1.

Executing an INSERT That Is Associated with a Cursor

Your Informix ESQL/C program must still use the DESCRIBE and SET DESCRIPTOR statements ("Handling an unknown column list" on page 16-18) to use a system-descriptor area for column-list values of an INSERT statement that inserts rows from an insert buffer. It must also use the PUT...USING SQL DESCRIPTOR statement with an insert cursor, as follows:

1. Prepare the INSERT statement and associate it with an insert cursor with the DECLARE statement. All multirow INSERT statements *must* have a declared insert cursor.

2. Create the cursor for the INSERT statement with the OPEN statement.

3. Insert the first set of column values into the insert buffer with a PUT statement and its USING SQL DESCRIPTOR clause. After this PUT statement, the column values stored in the specified system-descriptor area are stored in the insert buffer. Repeat the PUT statement within a loop until there are no more rows to insert.

4. After all the rows are inserted, exit the loop and flush the insert buffer with the FLUSH statement.

5. Close the insert cursor with the CLOSE statement.

You handle the insert cursor in much the same way as you handle the cursor associated with a SELECT statement ("Handling an Unknown Select List" on page 16-9). For more information about how to use an insert cursor, see the PUT statement in the *IBM Informix Guide to SQL: Syntax*.

Handling a Parameterized SELECT Statement

For an introduction on how to determine input parameters, see 15-17. This section describes how to handle a parameterized SELECT statement with a system-descriptor area. If a prepared SELECT statement has a WHERE clause with input parameters of unknown number and data type, your IBM Informix ESQL/C program must use a system-descriptor area to define the input parameters.

To use a system-descriptor area to define input parameters for a WHERE clause:

1. Determine the number and data types of the input parameters of the SELECT statement. For more information, see "Determining Unknown Input Parameters" on page 15-17.

2. Allocate a system-descriptor area and assign it a name with the ALLOCATE DESCRIPTOR statement. For more information about ALLOCATE DESCRIPTOR, see "Allocating Memory for a System-Descriptor Area" on page 16-3.

3. Indicate the number of input parameters in the WHERE clause with the SET DESCRIPTOR statement, which sets the COUNT field.

4. Store the definition and value of each input parameter with the SET DESCRIPTOR statement, which sets the DATA, TYPE, and LENGTH fields in the appropriate item descriptor:

   • The TYPE field must use the Informix ESQL/C data type constants defined in the **sqltypes.h** header file to represent the data types of the input parameters. For more information, see 15-13.

   • For a CHAR or VARCHAR value, LENGTH is the size, in bytes, of the character array; for a DATETIME or INTERVAL value, the LENGTH field stores the encoded qualifiers.

Important: If you use X/Open code (and compile with the **-xopen** flag), you must use the X/Open data type values for the TYPE and ITYPE fields. For more information, see "Determining the Data Type of a Column" on page 15-13.

If you use an indicator variable, you also need to set the INDICATOR field and perhaps the IDATA, ILENGTH, and ITYPE fields (for non-X/Open applications only). Use the VALUE keyword of SET DESCRIPTOR to identify the item descriptor. For more information about SET DESCRIPTOR, see "Assigning and Obtaining Values from a System-Descriptor Area" on page 16-5.

1. Pass the defined input parameters from the system-descriptor area to the database server with the USING SQL DESCRIPTOR clause.

   The statement that provides the input parameters depends on how many rows that the SELECT statement returns. The following sections discuss how to execute each type of SELECT statement.

2. Deallocate the system-descriptor area with the DEALLOCATE DESCRIPTOR statement. For more information, see "Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Important: If the SELECT statement has unknown columns in the select list, your program must also handle these columns with a system-descriptor area. For more information, see "Handling an Unknown Select List" on page 16-9.

The following sections demonstrate how to use a system-descriptor area to handle a parameterized SELECT statement that returns many rows and one that returns only a single row.

Executing a parameterized SELECT that returns multiple rows

The sample program described on the following pages shows how to use a dynamic SELECT statement with the following conditions:

- The SELECT returns more than row.

  The SELECT must be associated with a cursor, executed with the OPEN statement, and have its return values retrieved with the FETCH...USING SQL DESCRIPTOR statement.

- The SELECT has input parameters in its WHERE clause.

  The OPEN statement includes the USING SQL DESCRIPTOR clause to provide the parameter values in a system-descriptor area.

- The SELECT has unknown columns in the select list.

  The FETCH statement includes the USING SQL DESCRIPTOR clause to store the return values in a system-descriptor area.

This program is a version of the **demo4.ec** sample program; **demo4** uses a system-descriptor area for select-list columns while this modified version of **demo4** uses a system-descriptor area for *both* select-list columns and input parameters of a WHERE clause.

```
======================================================================
1. #include <stdio.h>
2. EXEC SQL include sqltypes;
3.
4. EXEC SQL define NAME_LEN 15;
5. EXEC SQL define MAX_IDESC 4;
6. main()
```

```
 7.  {
 8.  EXEC SQL BEGIN DECLARE SECTION;
 9.      int i;
10.      int desc_count;
11.      char demoquery[80];
12.      char queryvalue[2];
13.      char result[ NAME_LEN + 1 ];
14.  EXEC SQL END DECLARE SECTION;
15.      printf("Modified DEMO4 Sample ESQL program running.\n\n");
16.      EXEC SQL connect to 'stores7';
```

==

Lines 8 to 14

These lines declare host variables to hold the data obtained from the user and the
column values retrieved from the system descriptor.

==

```
17.      /* These next three lines have hard-wired both the query and
18.       * the value for the parameter. This information could have
19.       * been entered from the terminal and placed into the strings
20.       * demoquery and queryvalue, respectively.
21.       */
22.      sprintf(demoquery, "%s %s",
23.         "select fname, lname from customer",
24.         "where lname < ? ");
25.      EXEC SQL prepare demoid from :demoquery;
26.      EXEC SQL declare democursor cursor for demoid;
27.      EXEC SQL allocate descriptor 'demodesc' with max MAX_IDESC;
```

==

Lines 17 to 25

The lines assemble the character string for the statement (in **demoquery**) and
prepare it as the **demoid** statement identifier. The question mark (?) indicates the
input parameter in the WHERE clause. For more information about these steps, see
"Assembling and Preparing the SQL Statement" on page 14-2.

Line 26

This line declares the **democursor** cursor for the prepared statement identifier
demoid. All nonsingleton SELECT statements must have a declared cursor.

Line 27

To be able to use a system-descriptor area for the input parameters, you must first
allocate the system-descriptor area. This ALLOCATE DESCRIPTOR statement
allocates the **demodesc** system-descriptor area. For more information about
ALLOCATE DESCRIPTOR, see "Allocating Memory for a System-Descriptor Area"
on page 16-3.

==

```
28.      /*   This section of the program must evaluate :demoquery
29.       *    to count how many question marks are in the where
30.       *    clause and what kind of data type is expected for each
31.       *    question mark.
32.       *    For this example, there is one parameter of type
33.       *    char(15). It would then obtain the value for
34.       *    :queryvalue. The value of queryvalue is hard-wired in
35.       *    the next line.
36.       */
37.      sprintf(queryvalue, "C");
```

```
38.    desc_count = 1;
39.    if(desc_count > MAX_IDESC)
40.        {
41.        EXEC SQL deallocate descriptor 'demodesc';
42.        EXEC SQL allocate descriptor 'demodesc' with max :desc_count;
43.        }
44.    /* number of parameters to be held in descriptor is 1 */
45.    EXEC SQL set descriptor 'demodesc' COUNT = :desc_count;
```

==

Lines 28 to 38

These lines simulate the dynamic entry of the input parameter value. Although the parameter value is hard-coded here (line 37), the program would more likely obtain the value from user input. Line 38 simulates code that would determine how many input parameters exist in the statement string. If you did not know this value, you would need to include C code at this point to parse the statement string for the question mark (?) character.

Lines 39 to 43

This **if** statement determines if the **demodesc** system-descriptor area contains enough item descriptors for the parameterized SELECT statement. It compares the number of input parameters in the statement string (**desc_count**) with the number of item descriptors currently allocated (MAX_IDESC). If the program has not allocated enough item descriptors, the program deallocates the existing system-descriptor area (line 41) and allocates a new one (line 42); it uses the actual number of input parameters in the WITH MAX clause to specify the number of item descriptors to allocate.

Lines 44 and 45

This SET DESCRIPTOR statement stores the number of input parameters in the COUNT field of the **demodesc** system-descriptor area.

==

```
46.    /* Put the value of the parameter into the descriptor */
47.    i = SQLCHAR;
48.    EXEC SQL set descriptor 'demodesc' VALUE 1
49.         TYPE = :i, LENGTH = 15, DATA = :queryvalue;
50.    /* Associate the cursor with the parameter value */
51.    EXEC SQL open democursor using sql descriptor :demodesc;
52.    /*Reuse the descriptor to determine the contents of the Select-
        * list*/
53.    EXEC SQL describe qid using sql descriptor 'demodesc';
54.    EXEC SQL get descriptor 'demodesc' :desc_count = COUNT;
55.    printf("There are %d returned columns:\n", desc_count);
56.    /* Print out what DESCRIBE returns */
57.    for (i = 1; i <= desc_count; i++)
58.        prsysdesc(i);
59.    printf("\n\n");
```

==

Lines 47 to 49

This SET DESCRIPTOR statement sets the TYPE, LENGTH (for a CHAR value), and DATA fields for each of the parameters in the WHERE clause. The program only calls SET DESCRIPTOR once because it assumes that the SELECT statement has only one input parameter. If you do not know the number of input parameters at compile time, put the SET DESCRIPTOR in a loop for which the **desc_count** host variable controls the number of iterations.

Lines 50 and 51

The database server executes the SELECT statement when it opens the **democursor**
cursor. This OPEN statement includes the USING SQL DESCRIPTOR clause to
specify the **demodesc** system-descriptor area as the location of the input-parameter
values.

Lines 52 to 59

The program also uses the **demodesc** system-descriptor area to hold the columns
that are returned by the SELECT statement. The DESCRIBE statement (line 53)
examines the select list to determine the number and data types of these columns.
The GET DESCRIPTOR statement (line 54) then obtains the number of described
columns from the COUNT field of **demodesc**. Lines 55 to 58 then display the
column information for each returned column. For more information about how to
use a system-descriptor area to receive column values, see "Handling an Unknown
Select List" on page 16-9.

```
==========================================================================
60.     for (;;)
61.        {
62.        EXEC SQL fetch democursor using sql descriptor 'demodesc';
63.        if (sqlca.sqlcode != 0) break;
64.        for (i = 1; i <= desc_count; i++)
65.           {
66.           EXEC SQL get descriptor 'demodesc' VALUE :i :result = DATA;
67.           printf("%s ", result);
68.           }
69.        printf("\n");
70.        }
71.     if(strncmp(SQLSTATE, "02", 2) != 0)
72.        printf("SQLSTATE after fetch is %s\n", SQLSTATE);
73.     EXEC SQL close democursor;
74.     EXEC SQL free demoid; /* free resources for statement */
75.     EXEC SQL free democursor; /* free resources for cursor */
76.     /* free system-descriptor area */
77.     EXEC SQL deallocate descriptor 'demodesc';
78.     EXEC SQL disconnect current;
79.     printf("\nModified DEMO4 Program Over.\n\n");
80. }
==========================================================================
```

Lines 60 to 70

These lines access the fields of the item descriptor for each column in the select
list. After each FETCH statement, the GET DESCRIPTOR statement loads the
contents of the DATA field into the **result** host variable.

Line 73

After all the rows are fetched, the CLOSE statement frees the resources allocated to
the active set of the **democursor** cursor.

Lines 74 to 77

The FREE statement on line 74 frees the resources allocated to the **demoid**
statement identifier while the FREE statement on line 75 frees the resources to the
democursor cursor. The DEALLOCATE DESCRIPTOR statement frees the resources
allocated to the **demodesc** system-descriptor area. For more information, see
"Freeing Memory Allocated to a System-Descriptor Area" on page 16-8.

Executing a Parameterized Singleton SELECT

The instructions in the preceding section assume that the parameterized SELECT statement returns more than one row and, therefore, is associated with a cursor. If you know at the time you write the program that the parameterized SELECT statement will *always* return just one row, you can omit the cursor and use the EXECUTE...USING SQL DESCRIPTOR...INTO statement instead of the OPEN...USING SQL DESCRIPTOR statement to specify parameter values from a system-descriptor area. For more information, see "Specifying Input Parameter Values" on page 16-7.

Handling a parameterized user-defined routine

For an introduction on how to determine input parameters, see "Determining Unknown Input Parameters" on page 15-17. This section describes how to handle a parameterized user-defined routine with a system-descriptor area. The following statements execute user-defined routines:

- The EXECUTE FUNCTION statement executes a user-defined function (external and SPL).
- The EXECUTE PROCEDURE statement executes a user-defined procedure (external and SPL).

If a prepared EXECUTE PROCEDURE or EXECUTE FUNCTION statement has arguments specified as input parameters of unknown number and data type, your Informix ESQL/C program can use a system-descriptor area to define the input parameters.

Executing a parameterized function

You handle the input parameters of a user-defined function in the same way you handle input parameters in the WHERE clause of a SELECT statement, as follows:

- Execute a *noncursor* function in the same way as a singleton SELECT statement.

  If you know at the time that you write the program that the dynamic user-defined function *always* returns just one row, you can use the EXECUTE...USING SQL DESCRIPTOR...INTO statement to provide the argument values from a system-descriptor area and to execute the function. For more information, see "Executing a Parameterized Singleton SELECT" on page 16-27.

- Execute a *cursor* function in the same way as a SELECT statement that returns one or more rows.

  If you are not sure at the time that you write the program that the dynamic user-defined function *always* returns just one row, define a function cursor and use the OPEN...USING SQL DESCRIPTOR statement to provide the argument values from a system-descriptor area. For more information, see "Executing a parameterized SELECT that returns multiple rows" on page 16-23.

The only difference between the execution of these EXECUTE FUNCTION and SELECT statements is that you prepare the EXECUTE FUNCTION statement for the noncursor function, instead of the SELECT statement.

Executing a parameterized procedure

To execute a parameterized user-defined procedure, you can use the EXECUTE...USING SQL DESCRIPTOR statement to provide the argument values from a system-descriptor area and to execute the procedure. You handle the input parameters of a user-defined procedure in the same way you handle input

parameters in a noncursor function. The only difference between the execution of the EXECUTE PROCEDURE statement and the EXECUTE FUNCTION statement (for a noncursor function) is that you do not need to specify the INTO clause of the EXECUTE...USING SQL DESCRIPTOR statement for the user-defined procedure. For more information, see "Executing a parameterized function" on page 16-27.

Handling a Parameterized UPDATE or DELETE Statement

How you determine the input parameters in the WHERE clause of a DELETE or UPDATE statement is very similar to how you determine them in the WHERE clause of a SELECT statement. For more information, see "Handling a Parameterized SELECT Statement" on page 16-22. The major differences between these two types of dynamic parameterized statements are as follows:

- You do not need to use a cursor to handle a DELETE or UPDATE statement. Therefore, you provide the parameter values from a system-descriptor area with the USING SQL DESCRIPTOR clause of the EXECUTE statement instead of the OPEN statement.

- You can use the DESCRIBE...USING SQL DESCRIPTOR statement to determine if the DELETE or UPDATE statement has a WHERE clause. For more information, see "Checking for a WHERE Clause" on page 15-15.

The dyn_sql Program

The **dyn_sql.ec** program is an Informix ESQL/C demonstration program that uses dynamic SQL. The program prompts the user to enter a SELECT statement for the **stores7** demonstration database and then uses a system-descriptor area to execute the SELECT dynamically.

By default, the program opens the **stores7** database. If the demonstration database was given a name other than **stores7**, however, you can specify the database name on the command line. The following command runs the **dyn_sql** executable on the **mystores7** database:

```
dyn_sql mystores7
```

Compiling the Program

Use the following command to compile the **dyn_sql** program:

```
esql -o dyn_sql dyn_sql.ec
```

The **-o dyn_sql** option causes the executable program to be named **dyn_sql**. Without the **-o** option, the name of the executable program defaults to **a.out**. For more information about the **esql** preprocessor command, see "Using the esql Command" on page 2-4.

Guide to the dyn_sql.ec File

```
======================================================================
1.  /*
2.      This program prompts the user to enter a SELECT statement
3.      for the stores7 database. It processes the statement using
        dynamic sql
4.      and system descriptor areas and displays the rows returned by the
5.      database server.
6.  */
7.  #include <stdio.h>
```

```
 8.  #include <stdlib.h>
 9.  #include <ctype.h>
10.  EXEC SQL include sqltypes;
11.  EXEC SQL include locator;
12.  EXEC SQL include datetime;
13.  EXEC SQL include decimal;
14.  #define WARNNOTIFY      1
15.  #define NOWARNNOTIFY     0
16.  #define LCASE(c) (isupper(c) ? tolower(c) : (c))
17.  #define BUFFSZ 256
18.  extern char statement[80];
```

===

Lines 7 to 13

These lines specify C and Informix ESQL/C files to include in the program. The
stdio.h file enables **dyn_sql** to use the standard C I/O library. The **stdlib.h** file
contains string-to-number conversion functions, memory allocation functions, and
other miscellaneous standard library functions. The **ctypes.h** file contains macros
that check the attributes of a character. For example, one macro determines
whether a character is uppercase or lowercase.

The **sqltypes.h** header file contains symbolic constants that correspond to the data
types that are found in Informix databases. The program uses these constants to
determine the data types of columns that the dynamic SELECT statement returns.

The **locator.h** file contains the definition of the locator structure (**ifx_loc_t**), which
is the type of host variable needed for TEXT and BYTE columns. The **datetime.h**
file contains definitions of the **datetime** and **interval** structures, which are the data
types of host variables for DATETIME and INTERVAL columns. The **decimal.h** file
contains the definition of the **dec_t** structure, which is the type of host variable
needed for DECIMAL columns.

Lines 14 to 17

The **exp_chk()** exception-handling function uses the WARNNOTIFY and
NOWARNNOTIFY constants (lines 14 and 15). The second argument of **exp_chk()**
tells the function to display information in the SQLSTATE and SQLCODE variables
for warnings (WARNNOTIFY) or not to display information for warnings
(NOWARNNOTIFY). The **exp_chk()** function is in the **exp_chk.ec** source file. For a
description, see "Guide to the exp_chk.ec File" on page 11-32.

Line 16 defines LCASE, a macro that converts an uppercase character to a
lowercase character. Line 17 defines BUFFSZ to be the number 256. The program
uses BUFFSZ to specify the size of arrays that store input from the user.

Line 18

Line 18 declares **statement** as an external global variable to hold the name of the
last SQL statement that the program asked the database server to execute. The
exception-handling functions use this information. (See lines 399 to 406.)

===
```
19.  EXEC SQL BEGIN DECLARE SECTION;
20.      ifx_loc_t lcat_descr;
21.      ifx_loc_t lcat_picture;
22.  EXEC SQL END DECLARE SECTION;
23.  mint whenexp_chk();
24.  main(argc, argv)
25.  mint argc;
```

```
26.  char *argv[];
27.  {
28.     int4 ret, getrow();
29.     short data_found = 0;
30.     EXEC SQL BEGIN DECLARE SECTION;
31.         char ans[BUFFSZ], db_name[30];
32.         char name[40];
33.         mint sel_cnt, i;
34.         short type;
35.     EXEC SQL END DECLARE SECTION;
36.      printf("DYN_SQL Sample ESQL Program running.\n\n");
37.      EXEC SQL whenever sqlerror call whenexp_chk;
38.      if (argc > 2)                  /* correct no. of args? */
39.        {
40.        printf("\nUsage: %s [database]\nIncorrect no. of
           argument(s)\n",
41.            argv[0]);
42.        printf("\nDYN_SQL Sample Program over.\n\n");
43.        exit(1);
44.        }
45.     strcpy(db_name, "stores7");
46.     if(argc == 2)
47.        strcpy(db_name, argv[1]);
48.     sprintf(statement,"CONNECT TO %s",db_name);
49.     EXEC SQL connect to :db_name;
50.     printf("Connected to %s\n", db_name);
51.     ++argv;
```
==

Lines 19 to 23

Lines 19 to 23 define the global host variables that are used in SQL statements.
Lines 20 and 21 define the locator structures that are the host variables for the
cat_descr and **cat_picture** columns of the **catalog** table. Line 23 declares the
whenexp_chk() function, which the program calls when an error occurs on an SQL
statement.

Lines 24 to 27

The **main()** function is the point where the program begins to execute. The **argc**
parameter gives the number of arguments from the command line when the
program was invoked. The **argv** parameter is an array of pointers to command-line
arguments. This program expects only one argument (the name of the database to
be accessed), and it is optional.

Lines 28 to 51

Line 28 defines an **int4** data type (**ret**) to receive a return value from the **getrow()**
function. Line 28 also declares that the **getrow()** function returns a **int4** data type.
Lines 30 to 35 define the host variables that are local to the **main()** program block.
Line 37 executes the WHENEVER statement to transfer control to **whenexp_chk()**
if any errors occur in SQL statements. For more information about the
whenexp_chk() function, see "Guide to the exp_chk.ec File" on page 11-32.

Lines 38 to 51 establish a connection to a database. If **argc** equals 2, the program
assumes that the user entered a database name on the command line (by
convention the first argument is the name of the program), and the program opens
this database. If the user did not enter a database name on the command line, the
program opens the **stores7** database (see line 45), which is the default. In both

cases, the program connects to the default database server that is specified by the
INFORMIXSERVER environment variable because no database server is specified.

```
=======================================================================
52.    while(1)
53.        {
54.        /* prompt for SELECT statement */
55.        printf("\nEnter a SELECT statement for the %s database",
56.                db_name);
57.        printf("\n\t(e.g.  select * from customer;)\n");
58.        printf("\tOR a ';' to terminate program:\n>> ");
59.        if(!getans(ans, BUFFSZ))
60.            continue;
61.        if (*ans == ';')
62.            {
63.            strcpy(statement, "DISCONNECT");
64.            EXEC SQL disconnect current;
65.            printf("\nDYN_SQL Sample Program over.\n\n");
66.            exit(1);
67.            }
68.        /* prepare statement id */
69.        printf("\nPreparing statement (%s)...\n", ans);
70.        strcpy(statement, "PREPARE sel_id");
71.        EXEC SQL prepare sel_id from :ans;
72.        /* declare cursor */
73.        printf("Declaring cursor 'sel_curs' for SELECT...\n");
74.        strcpy(statement, "DECLARE sel_curs");
75.        EXEC SQL declare sel_curs cursor for sel_id;
76.        /* allocate descriptor area */
77.        printf("Allocating system-descriptor area...\n");
78.        strcpy(statement, "ALLOCATE DESCRIPTOR selcat");
79.        EXEC SQL allocate descriptor 'selcat';
80.     /* Ask the database server to describe the statement */
81.        printf("Describing prepared SELECT...\n");
82.        strcpy(statement,
83.           "DESCRIBE sel_id USING SQL DESCRIPTOR selcat");
84.        EXEC SQL describe sel_id using sql descriptor 'selcat';
85.        if (SQLCODE != 0)
86.            {
87.            printf("** Statement is not a SELECT.\n");
88.            free_stuff();
89.            strcpy(statement, "DISCONNECT");
90.            EXEC SQL disconnect current;
91.            printf("\nDYN_SQL Sample Program over.\n\n");
92.            exit(1);
93.            }

=======================================================================
```

Lines 52 to 67

The **while(1)** on line 52 begins a loop that continues to the end of the **main()**
function. Lines 55 to 58 prompt the user to enter either a SELECT statement or, to
terminate the program, a semicolon. The **getans()** function receives the input from
the user. If the first character is not a semicolon, the program continues to process
the input.

Lines 68 to 75

The PREPARE statement prepares the SELECT statement (which the user enters) from the array **ans[]** and assigns it the statement identifier **sel_id.** The PREPARE statement enables the database server to parse, validate, and generate an execution plan for the statement.

The DECLARE statement (lines 72 to 75) creates the **sel_curs** cursor for the set of rows that the SELECT statement returns, in case it returns more than one row.

Lines 76 to 79

The ALLOCATE DESCRIPTOR statement allocates the **selcat** system-descriptor area in memory. The statement does not include the WITH MAX clause and, therefore, uses the default memory allocation, which is for 100 columns.

Lines 80 to 93

The DESCRIBE statement obtains information from the database server about the statement that is in the **sel_id** statement identifier. The database server returns the information in the **selcat** system-descriptor area, which the preceding ALLOCATE DESCRIPTOR statement creates. The information that DESCRIBE puts into the system-descriptor area includes the number, names, data types, and lengths of the columns in the select list.

The DESCRIBE statement also sets the SQLCODE variable to a number that indicates the type of statement that was described. To check whether the statement type is SELECT, line 85 compares the value of SQLCODE to 0 (the value defined in the **sqlstypes.h** file for a SELECT statement with no INTO TEMP clause). If the statement is not a SELECT, line 87 displays a message to that effect and the program frees the cursor and the resources that have been allocated. Then it closes the connection and exits.

```
========================================================================
94.     /* Determine the number of columns in the select list */
95.        printf("Getting number of described values from ");
96.        printf("system-descriptor area...\n");
97.        strcpy(statement, "GET DESCRIPTOR selcat: COUNT field");
98.        EXEC SQL get descriptor 'selcat' :sel_cnt = COUNT;
99.     /* open cursor; process select statement */
100.        printf("Opening cursor 'sel_curs'...\n");
101.        strcpy(statement, "OPEN sel_curs");
102.        EXEC SQL open sel_curs;
103.    /*
104.     * The following loop checks whether the cat_picture or
105.     * cat_descr columns are described in the system-descriptor area.
106.     * If so, it initializes a locator structure to read the simple
107.     * large-object data into memory and sets the address of the
108.     * locator structure in the system-descriptor area.
109.     */
110.        for(i = 1; i <= sel_cnt; i++)
111.          {
112.          strcpy(statement,
113.             "GET DESCRIPTOR selcat: TYPE, NAME fields");
114.          EXEC SQL get descriptor 'selcat' VALUE :i
115.             :type = TYPE,
116.             :name = NAME;
117.          if (type == SQLTEXT && !strncmp(name, "cat_descr",
118.              strlen("cat_descr")))
119.             {
```

```
120.            lcat_descr.loc_loctype = LOCMEMORY;
121.            lcat_descr.loc_bufsize = -1;
122.            lcat_descr.loc_oflags = 0;
123.            strcpy(statement, "SET DESCRIPTOR selcat: DATA field");
124.            EXEC SQL set descriptor 'selcat' VALUE :i
125.               DATA = :lcat_descr;
126.            }
127.        if (type == SQLBYTES && !strncmp(name, "cat_picture",
128.             strlen("cat_picture")))
129.            {
130.            lcat_picture.loc_loctype = LOCMEMORY;
131.            lcat_picture.loc_bufsize = -1;
132.            lcat_picture.loc_oflags = 0;
133.            strcpy(statement, "SET DESCRIPTOR selcat: DATA field");
134.            EXEC SQL set descriptor 'selcat' VALUE :i
135.               DATA = :lcat_picture;
136.            }
137.         }
```

===

Lines 94 to 98

The GET DESCRIPTOR statement retrieves the COUNT value from the **selcat**
system-descriptor area. The COUNT value indicates how many columns are
described in the system-descriptor area.

Lines 99 to 102

The OPEN statement begins execution of the dynamic SELECT statement and
activates the **sel_curs** cursor for the set of rows that it returns.

Lines 114 to 137

This section of the code uses the GET DESCRIPTOR statement to determine
whether the simple large-object columns from the **catalog** table (**cat_descr** and
cat_picture) are included in the select list. If you dynamically select a simple
large-object column, you must set the address of a locator structure into the DATA
field of the item descriptor to tell the database server where to return the locator
structure.

First, however, the program initializes the locator structure, as follows:

- The data is returned in a memory buffer
  (**loc_loctype** = LOCMEMORY).
- The database server allocates the memory buffer (**loc_bufsize** = **-1**).

(For more information about how to work with the TEXT and BYTE data types,
see Chapter 7, "Working with simple large objects," on page 7-1.) Then the
program uses the SET DESCRIPTOR statement to load the address of the locator
structure into the DATA field of the descriptor area.

===
```
138.      while(ret = getrow("selcat"))            /* fetch a row */
139.         {
140.         data_found = 1;
141.         if (ret < 0)
142.            {
143.            strcpy(statement, "DISCONNECT");
144.            EXEC SQL disconnect current;
145.            printf("\nDYN_SQL Sample Program over.\n\n");
```

```
146.             exit(1);
147.               }
148.           disp_data(sel_cnt, "selcat");        /* display the data */
149.             }
150.         if (!data_found)
151.             printf("** No matching rows found.\n");
152.         free_stuff();
153.         if (!more_to_do())        /* More to do? */
154.             break;          /* no, terminate loop */
155.         }
156. }
157. /* fetch the next row for selected items */
158. int4 getrow(sysdesc)
159. EXEC SQL BEGIN DECLARE SECTION;
160.     PARAMETER char *sysdesc;
161. EXEC SQL END DECLARE SECTION;
162. {
163.     int4 exp_chk();
164.     sprintf(statement, "FETCH %s", sysdesc);
165.     EXEC SQL fetch sel_curs using sql descriptor :sysdesc;
166.     return((exp_chk(statement)) == 100 ? 0 : 1);
167. }
```
===

Lines 138 to 149

The **getrow()** function retrieves the selected rows one by one. Each iteration of the **while** loop retrieves one row, which the program then processes with the **disp_data()** function (line 148). When all the rows are retrieved, **getrow()** returns a 0 (zero) and the **while** loop terminates. For more information about the **getrow()** function, see "Lines 157 to 167."

Line 152

The **free_stuff()** function frees resources that were allocated when the dynamic SELECT statement was processed. See "Lines 381 to 387" on page 16-40.

Lines 153 to 156

When all the selected rows are processed, the program calls the **more_to_do()** function, which asks whether the user would like to process more SELECT statements. If the answer is no, **more_to_do()** returns 0 and the **break** statement terminates the **while** loop that began on line 52. If the answer is yes, the program begins the next iteration of the **while** statement on line 52 to accept and process another SELECT statement.

Lines 157 to 167

The **getrow()** function moves the cursor to and then fetches the next row in the set of rows that are returned by the dynamic SELECT statement. It fetches the row values into the system-descriptor area that is specified in the **sysdesc** variable. If there are no more rows to fetch (**exp_chk()** returns 100), **getrow()** returns 0. If the FETCH encounters a runtime error, **getrow()** returns 1.

===
```
168. {/*
169.   * This function loads a column into a host variable of the correct
170.   * type and displays the name of the column and the value, unless
      * the
171.   * value is NULL.
172.   */
```

```
173.  disp_data(col_cnt, sysdesc)
174.  mint col_cnt;
175.  EXEC SQL BEGIN DECLARE SECTION;
176.      PARAMETER char *sysdesc;
177.  EXEC SQL END DECLARE SECTION;
178.      EXEC SQL BEGIN DECLARE SECTION;
179.          mint int_data, i;
180.          char *char_data;
181.          int4 date_data;
182.          datetime dt_data;
183.          interval intvl_data;
184.          decimal dec_data;
185.          short short_data;
186.          char name[40];
187.          short char_len, type, ind;
188.      EXEC SQL END DECLARE SECTION;
189.      int4 size;
190.      unsigned amount;
191.      mint x;
192.      char shdesc[81], str[40], *p;
193.      printf("\n\n");
194.      /* For each column described in the system descriptor area,
195.       * determine its data type. Then retrieve the column name and its
196.       * value, storing the value in a host variable defined for the
197.       * particular data type. If the column is not NULL, display the
198.       * name and value.
199.       */
200.      for(i = 1; i <= col_cnt; i++)
201.        {
202.        strcpy(statement, "GET DESCRIPTOR: TYPE field");
203.        EXEC SQL get descriptor :sysdesc VALUE :i
204.            :type = TYPE;
205.        switch(type)
206.            {
207.            case SQLSERIAL:
208.            case SQLINT:
209.                strcpy(statement,
210.                    "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
211.                EXEC SQL get descriptor :sysdesc VALUE :i
212.                    :name = NAME,
213.                    ind = INDICATOR,
214.                    :int_data = DATA;
215.                if(ind == -1)
216.                    printf("\n%.20s: NULL", name);
217.                else
218.                    printf("\n%.20s: %d", name, int_data);
219.                break;
220.            case SQLSMINT:
221.                strcpy(statement,
222.                    "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
223.                EXEC SQL get descriptor :sysdesc VALUE :i
224.                    :name = NAME,
225.                    :ind = INDICATOR,
226.                    :short_data = DATA;
227.                if(ind == -1)
228.                    printf("\n%.20s: NULL", name);
```

```
229.              else
230.                  printf("\n%.20s: %d", name, short_data);
231.              break;
232.          case SQLDECIMAL:
233.          case SQLMONEY:
234.              strcpy(statement,
235.                  "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
236.              EXEC SQL get descriptor :sysdesc VALUE :i
237.                  :name = NAME,
238.                  :ind = INDICATOR,
239.                  :dec_data = DATA;
240.              if(ind == -1)
241.                  printf("\n%.20s: NULL", name);
242.              else
243.                  {
244.                  if(type == SQLDECIMAL)
245.                      rfmtdec(&dec_data, "###,###,###.##", str);
246.                  else
247.                      rfmtdec(&dec_data, "$$$,$$$,$$$.$$", str);
248.                  printf("\n%.20s: %s", name, str);
249.                  }
250.              break;
251.          case SQLDATE:
252.              strcpy(statement,
253.                  "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
254.              EXEC SQL get descriptor :sysdesc VALUE :i
255.                  :name = NAME,
256.                  :ind = INDICATOR,
257.                  :date_data = DATA;
258.              if(ind == -1)
259.                  printf("\n%.20s: NULL", name);
260.              else
261.                  {
262.                  if((x = rfmtdate(date_data, "mmm. dd, yyyy",
263.                          str)) < 0)
264.                      printf("\ndisp_data() - DATE - fmt error");
265.                  else
266.                      printf("\n%.20s: %s", name, str);
267.                  }
268.              break;
269.          case SQLDTIME:
270.              strcpy(statement,
271.                  "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
272.              EXEC SQL get descriptor :sysdesc VALUE :i
273.                  :name = NAME,
274.                  :ind = INDICATOR,
275.                  :dt_data = DATA;
276.              if(ind == -1)
277.                  printf("\n%.20s: NULL", name);
278.              else
279.                  {
280.                  x = dttofmtasc(&dt_data, str, sizeof(str), 0);
281.                  printf("\n%.20s: %s", name, str);
282.                  }
283.              break;
284.          case SQLINTERVAL:
```

```
285.            strcpy(statement,
286.               "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
287.            EXEC SQL get descriptor :sysdesc VALUE :i
288.               :name = NAME,
289.               :ind = INDICATOR,
290.               :intvl_data = DATA;
291.            if(ind == -1)
292.               printf("\n%.20s: NULL", name);
293.            else
294.               {
295.               if((x = intofmtasc(&intvl_data, str,
296.                     sizeof(str),
297.                     "%3d days, %2H hours, %2M minutes"))
298.                     < 0)
299.                  printf("\nINTRVL - fmt error %d", x);
300.               else
301.                  printf("\n%.20s: %s", name, str);
302.               }
303.            break;
304.         case SQLVCHAR:
305.         case SQLCHAR:
306.            strcpy(statement,
307.               "GET DESCRIPTOR: LENGTH, NAME fields");
308.            EXEC SQL get descriptor :sysdesc VALUE :i
309.               :char_len = LENGTH,
310.               :name = NAME;
311.            amount = char_len;
312.            if(char_data = (char *)(malloc(amount + 1)))
313.               {
314.               strcpy(statement,
315.                  "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
316.               EXEC SQL get descriptor :sysdesc VALUE :i
317.                  :char_data = DATA,
318.                  :ind =  INDICATOR;
319.               if(ind == -1)
320.                  printf("\n%.20s: NULL", name);
321.               else
322.                  printf("\n%.20s: %s", name, char_data);
323.               }
324.            else
325.               {
326.               printf("\n%.20s: ", name);
327.               printf("Can't display: out of memory");
328.               }
329.            break;
330.         case SQLTEXT:
331.            strcpy (statement,
332.               "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
333.            EXEC SQL get descriptor :sysdesc VALUE :i
334.               :name = NAME,
335.               :ind = INDICATOR,
336.               :lcat_descr = DATA;
337.            size = lcat_descr.loc_size;    /* get size of data */
338.            printf("\n%.20s: ", name);
339.            if(ind == -1)
340.               {
```

```
341.                  printf("NULL");
342.                  break;
343.                  }
344.               p = lcat_descr.loc_buffer;      /* set p to buf addr */
345.            /* print buffer 80 characters at a time */
346.               while(size >= 80)
347.                  {
348.                  /* mv from buffer to shdesc */
349.                  ldchar(p, 80, shdesc);
350.                  printf("\n%80s", shdesc);      /* display it */
351.                  size -= 80;          /* decrement length */
352.                  p += 80;          /* bump p by 80 */
353.                  }
354.               strncpy(shdesc, p, size);
355.               shdesc[size] = '\0';
356.               printf("%-s\n", shdesc);      /* dsply last segment */
357.               break;
358.            case SQLBYTES:
359.               strcpy (statement,
360.                  "GET DESCRIPTOR: NAME, INDICATOR fields");
361.               EXEC SQL get descriptor :sysdesc VALUE :i
362.                  :name = NAME,
363.                  :ind = INDICATOR;
364.               if(ind == -1)
365.                  printf("%.20s: NULL", name);
366.               else
367.                  {
368.                  printf("%.20s: ", name);
369.                  printf("Can't display BYTE type value");
370.                  }
371.               break;
372.            default:
373.               printf("\nUnexpected data type: %d", type);
374.               EXEC SQL disconnect current;
375.               printf("\nDYN_SQL Sample Program over.\n\n");
376.               exit(1);
377.            }
378.         }
379.     printf("\n");
380. }
```

==

Lines 168 to 380

The **disp_data()** function displays the values that are stored in each row that the
SELECT statement returns. The function must be able to receive and process any
data type within the scope of the dynamic SELECT statement (in this case, any
column within the **stores7** database). This function accepts two arguments: *col_cnt*
contains the number of columns that are contained in the system-descriptor area,
and *sysdesc* contains the name of the system-descriptor area that contains the
column information. This second argument must be declared with the
PARAMETER keyword because the argument is used in the FETCH statement.

The **disp_data()** function first defines host variables for each of the data types that
are found in the **stores7** database (lines 178 to 188), except for the locator
structures that have been globally defined already for the **cat_descr** and
cat_picture columns of the **catalog** table (lines 19 to 22).

For each column that is described in the system-descriptor area, **disp_data()** retrieves its data type with a GET DESCRIPTOR statement. Next, **disp_data()** executes a **switch** on that data type and, for each type (column), it executes another GET DESCRIPTOR statement to retrieve the name of the column, the indicator flag, and the data. Unless the column is null, **disp_data()** moves the column data from the DATA field of the system-descriptor area to a corresponding host variable. Then it displays the column name and the content of the host variable.

The **disp_data()** function uses the symbolic constants defined in **sqltypes.h** to compare data types. It also uses the Informix ESQL/C library functions **rfmtdec()**, **rfmtdate()**, **dttofmtasc()**, and **intofmtosc()** to format the DECIMAL and MONEY, DATE, DATETIME, and INTERVAL data types for display.

For the TEXT and BYTE data types, you can retrieve the value of the column with the following two-stage process, because the database server returns a locator structure rather than the data:

• The GET DESCRIPTOR statement (lines 333 and 361) retrieves the locator structure from the system-descriptor area and moves it to the **ifx_loc_t** host variable.
• The **disp_data()** function obtains the address of the data buffer from the locator structure, in **loc_buffer**, and retrieves the data from there.

In the case of the BYTE data type, for the sake of brevity **disp_data()** retrieves the locator structure but does not display the data. For an example of the type of logic required to display a BYTE column, see "Guide to the dispcat_pic.ec File" on page 7-33.

```
========================================================================
381.   free_stuff()
382.   {
383.     EXEC SQL free sel_id;     /* free resources for statement */
384.     EXEC SQL free sel_curs;    /* free resources for cursor */
385.     /* free system descriptor area */
386.     EXEC SQL deallocate descriptor 'selcat';
387.   }
388.   /*
389.    * The inpfuncs.c file contains the following functions used in
        * this
390.    * program:
391.    *    more_to_do() - asks the user to enter 'y' or 'n' to indicate
392.    *                   whether to run the main program loop again.
393.    *
394.    *    getans(ans, len) - accepts user input, up to 'len' number of
395.    *                   characters and puts it in 'ans'
396.    */
397.   #include "inpfuncs.c"
398.   /*
399.    * The exp_chk.ec file contains the exception handling functions to
400.    * check the SQLSTATE status variable to see if an error has
        * occurred
401.    * following an SQL statement. If a warning or an error has
402.    * occurred, exp_chk() executes the GET DIAGNOSTICS statement and
403.    * displays the detail for each exception that is returned.
404.    */
405.   EXEC SQL include exp_chk.ec;
```

==

Lines 381 to 387

The **free_stuff()** function frees resources that were allocated to process the dynamic statement. Line 383 frees resources that were allocated by the application when it prepared the dynamic SELECT statement. Line 384 releases resources allocated by the database server to process the **sel_curs** cursor. The DEALLOCATE DESCRIPTOR statement releases the memory allocated for the **selcat** system-descriptor area and its associated data areas.

Lines 388 to 397

Several of the Informix ESQL/C demonstration programs also call the **more_to_do()** and **getans()** functions. Therefore, these functions are also broken out into a separate C source file and included in the appropriate demonstration program. Neither of these functions contain Informix ESQL/C, so the program can use the C **#include** preprocessor statement to include the file. For a description of these functions, see "Guide to the inpfuncs.c File" on page 7-42.

Lines 398 to 405

As a result of the WHENEVER statement on line 37, the **whenexp_chk()** function is called if an error occurs during the execution of an SQL statement. The **whenexp_chk()** function examines the SQLSTATE status variable to determine the outcome of an SQL statement. Because several demonstration programs use this function with the WHENEVER statement for exception handling, the **whenexp_chk()** function and its supporting functions have been broken out into a separate **exp_chk.ec** source file. The **dyn_sql** program must include this file with the Informix ESQL/C **include** directive because the exception-handling functions use Informix ESQL/C statements. The **exp_chk.ec** source file is described in Chapter 11, "Exception handling," on page 11-1.

Tip: In a production environment, you would put functions such as **more_to_do()**, **getans()**, and **whenexp_chk()** into a library and include them on the command line when you compile the Informix ESQL/C program.

Chapter 17. Using an sqlda structure

An **sqlda** structure is a dynamic-management structure that can hold data that is either returned from or sent to the database server by a prepared statement. It is a C structure defined in the **sqlda.h** header file.

Important: The **sqlda** structure does not conform to the X/Open standards. It is an Informix extension to Informix ESQL/C.

These topics describe the following information about how to use an **sqlda** structure:

- Using an **sqlda** structure to hold unknown values
- Managing an **sqlda** structure
- Using an **sqlda** structure to handle unknown values in dynamic SQL statements

For information about the fields of the **sqlda** structure, see "An sqlda Structure" on page 15-5.

Managing an sqlda structure

Your Informix ESQL/C program can manipulate an **sqlda** structure with the SQL statements that the following table summarizes.

Table 17-1. SQL Statements That Can Be Used to Manipulate an sqlda Structure

SQL Statement	Purpose	See
DESCRIBE...INTO	Allocates an **sqlda** structure and initializes this structure with information about column-list columns	17-3 17-4
For SELECT and EXECUTE FUNCTION statements that use cursors:		
OPEN...USING DESCRIPTOR	Takes any input parameters from the specified **sqlda** structure	17-8 17-9
FETCH...USING DESCRIPTOR	Puts the contents of the row into the **sqlda** structure	
For SELECT and EXECUTE FUNCTION statements that return *only one row*:		
EXECUTE...INTO DESCRIPTOR	Puts the contents of the singleton row into the **sqlda** structure	17-9
For non-SELECT statements:		
EXECUTE...USING DESCRIPTOR	Takes any input parameters from the specified **sqlda** structure	17-8
For an INSERT statement that uses an insert cursor:		
PUT...USING DESCRIPTOR	Puts a row into the insert buffer after it obtains the column values from the specified **sqlda** structure	17-18

In addition, your Informix ESQL/C program can manage an **sqlda** structure in the following ways:

- Declare a variable pointer to an **sqlda** structure.

- Assign values to the **sqlda** fields to provide the database server with missing column information.
- Obtain information from the **sqlda** fields to access column information that is received from the database server.
- Free the memory allocated to the **sqlda** structure when your program is finished with it.

Defining an sqlda Structure

The Informix ESQL/C **sqlda.h** header file defines the **sqlda** structure. To define an **sqlda** structure, the Informix ESQL/C program must take the following actions:

- Include the **sqlda.h** header file to provide the declaration for **sqlda** in your program

  The Informix ESQL/C preprocessor automatically includes the **sqlhdr.h** file, which includes the **sqlda.h** header file.

- Declare a variable name as a pointer to the **sqlda** structure

  The following line of code declares the **da_ptr** variable as an **sqlda** pointer:

  ```
  struct sqlda *da_ptr;
  ```

Important: The pointer to an **sqlda** structure is not an Informix ESQL/C host variable. Therefore, you do not need to precede the statement declaration with either the keywords EXEC SQL or a dollar ($) symbol. Furthermore, in the program blocks you do not precede any references to the pointer with a colon (:) or a dollar ($) symbol.

Allocating Memory for the sqlda Structure

Once you define a host variable as a pointer to an **sqlda** structure, you must ensure that memory is allocated for all parts of this structure, as follows:

- To allocate memory for the **sqlda** structure itself, use the DESCRIBE...INTO statement.

  The following DESCRIBE statement obtains information about the prepared statement **st_id**, allocates memory for an **sqlda** structure, and puts the address of the **sqlda** structure in the pointer **da_ptr**:

  ```
  EXEC SQL describe st_id into da_ptr;
  ```

- To allocate memory for the **sqlvar_struct** structures, take the following actions:
  - If the prepared statement is a SELECT (with no INTO TEMP clause), INSERT, or EXECUTE FUNCTION statement, the DESCRIBE...INTO statement can allocate space for **sqlvar_struct** structures.
  - If some other SQL statement was prepared and you need to send or receive columns in the database server, your program must allocate space for the **sqlvar_struct** structures.

  For more information, see "Initializing the sqlda Structure" on page 17-4.

- To allocate memory for the data of the **sqldata** fields, make sure you align the data types with proper word boundaries. For more information, see "Allocating Memory for Column Data" on page 17-6.

If you use the **sqlda** structure to define input parameters, you cannot use a DESCRIBE statement. Therefore, your program must explicitly allocate memory for both the **sqlda** structure and the **sqlvar_struct** structures. (See "Specifying Input Parameter Values" on page 17-8.)

Initializing the sqlda Structure

To send or receive column values in the database, your Informix ESQL/C program must initialize the **sqlda** structure so that it describes the unknown columns of the prepared statement. To initialize the **sqlda** structure, you must perform the following steps:

- Set the **sqlvar** field to the address of the initialized **sqlvar_struct** structures.
- Set the **sqld** field to indicate the number of unknown columns (and associated **sqlvar_struct** structures).

In addition to allocating memory for the **sqlda** structure (see "Allocating Memory for the sqlda Structure" on page 17-3), the DESCRIBE...INTO statement also initializes this structure with information about the prepared statement. The information that DESCRIBE...INTO can provide depends on which SQL statement it has described.

If the prepared statement is a SELECT (with no INTO TEMP clause), INSERT, or EXECUTE FUNCTION statement, the DESCRIBE...INTO statement can determine information about columns in the column list. Therefore, the DESCRIBE...INTO statement takes the following actions to initialize an **sqlda** structure:

- It allocates memory for the **sqlda** structure. For more information, see "Allocating Memory for the sqlda Structure" on page 17-3.
- It sets the **sqlda.sqld** field, which contains the number of **sqlvar_struct** structures initialized with data. This value is the number of columns and expressions in the column list (SELECT and INSERT) or the number of returned values (EXECUTE FUNCTION).
- It allocates memory for component **sqlvar_struct** structures, one **sqlvar_struct** structure for each column or expression in the column list (SELECT and INSERT) or for each of the returned values (EXECUTE FUNCTION).
- It sets the **sqlda.sqlvar** field to the initial address of the memory that DESCRIBE allocates for the **sqlvar_struct** structures.
- It describes each unknown column in the prepared SELECT (without an INTO TEMP), EXECUTE FUNCTION, or INSERT statement. The DESCRIBE...INTO statement initializes the fields of the **sqlvar_struct** structure for each column, as follows:
  - It initializes the **sqltype**, **sqllen**, and **sqlname** fields (for CHAR type data or for the qualifier of DATETIME or INTERVAL data) to provide information from the database about the column.

    For most data types, the **sqllen** field holds the length, in bytes, of the column. If the column is a collection type (SET, MULTISET, or LIST), a row type (named or unnamed), or an opaque type, the **sqllen** field is zero.
  - It initializes the **sqldata** and **sqlind** fields to null.

For descriptions of these fields, see Table 15-5 on page 15-7.

Important: Unlike with a system-descriptor area, DESCRIBE with an **sqlda** pointer does not allocate memory for the column data (the **sqldata** fields). Before your program receives column values from the database server, it must allocate this data space.

For more information, see "Allocating Memory for Column Data" on page 17-6.

The DESCRIBE statement provides information about the columns of a column list. Therefore, you usually use DESCRIBE...INTO *after* a SELECT (without an INTO TEMP clause), INSERT, or EXECUTE FUNCTION statement was prepared. The

DESCRIBE...INTO statement not only initializes the **sqlda** structure, but also returns the type of SQL statement prepared. For more information, see "Determining the Statement Type" on page 15-9.

The following DESCRIBE statement also allocates memory for an **sqlda** structure and for two **sqlvar_struct** data structures (one for the **customer_num** column and another for the **company** column) and then initializes the pointer **da_ptr->sqlvar** with the initial address of the memory that is allocated to the **sqlvar_struct** structure:

```
EXEC SQL prepare st_id
   'select customer_num, company from customer
   where customer_num = ?';
EXEC SQL describe st_id into da_ptr;
```

The preceding DESCRIBE...INTO statement returns an SQLCODE value of zero (0) to indicate that the prepared statement was a SELECT statement.

Figure 17-1 shows a sample **sqlda** structure that this DESCRIBE...INTO statement might initialize.
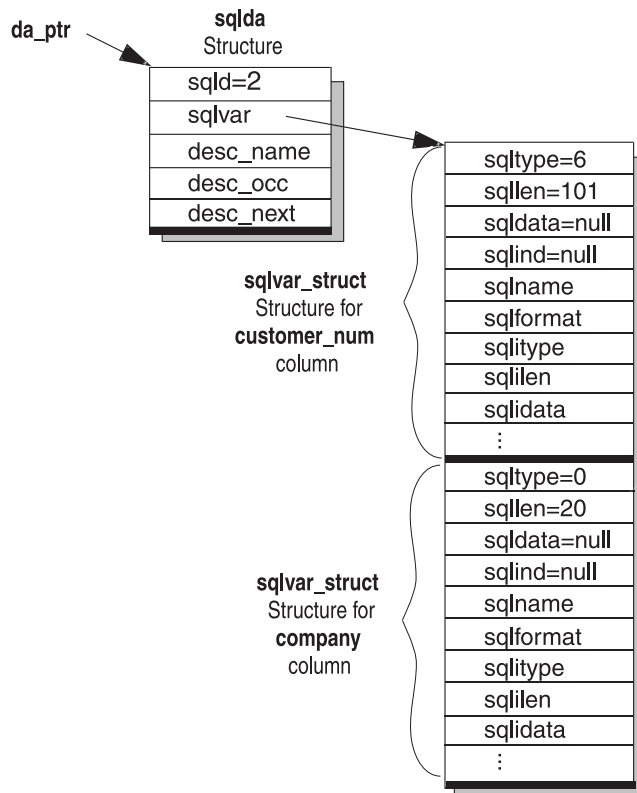
Figure 17-1. Sample sqlda Structure for Two Columns

If some other SQL statement was prepared, the DESCRIBE...INTO statement cannot initialize the **sqlda** structure. To send or receive column values in the database, your program must perform this initialization explicitly, as follows:

- Allocate memory for component **sqlvar_struct** structures, one **sqlvar_struct** structure for each column.

  You can use system memory-allocation functions such as **malloc()** or **calloc()** and assign the address to **sqlvar**, as follows:

```
da_ptr->sqlvar = (struct sqlvar_struct *)
    calloc(count, sizeof(struct sqlvar_struct));
```

- Perform the following tasks to describe each unknown column:
  - Set the **sqlda.sqld** field, which contains the number of **sqlvar_struct** structures initialized with data. This value is the number of unknown columns in the prepared statement.
  - Initialize the fields of each **sqlvar_struct** structure.

    Set the **sqltype**, **sqllen**, and **sqlname** fields (for CHAR type data or for the qualifier for DATETIME or INTERVAL data) to provide information about a column to the database server.

  To provide the column data, your program must also allocate space for this data and set the **sqldata** field of each **sqlvar_struct** structure to the appropriate location within this space. For more information, see Allocating Memory for Column Data. If you send column data to the database server, be sure to set the **sqlind** field appropriately.

If you use the **sqlda** structure to define input parameters, you cannot use a DESCRIBE statement to initialize the **sqlda** structure. Your code must explicitly set the appropriate fields of the **sqlda** structure to define the input parameters. (See "Specifying Input Parameter Values" on page 17-8.)

Allocating Memory for Column Data

The **sqlda** structure stores a pointer to the data for each column in the **sqldata** field of an **sqlvar_struct** structure. Unlike the DESCRIBE...USING SQL DESCRIPTOR statement, the DESCRIBE...INTO statement does not allocate memory for this data. When the DESCRIBE...INTO statement allocates memory for the **sqlda** pointer, it initializes the **sqldata** fields of each **sqlvar_struct** structure to null.

To send or receive column data in the database, your IBM Informix ESQL/C program must perform the following tasks:

- Allocate memory for the column data.
- Set the **sqldata** field for the **sqlvar_struct** structure associated with the column to the address of the memory allocated for the column data.

To allocate memory for the **sqldata** fields, you can use a system memory-allocation function such as **malloc()** or **calloc()**. As an alternative to the **malloc()** system memory-allocation function, your program can declare a static character buffer for the data buffer. Figure 17-2 shows a code fragment that allocates column data from a static character buffer called **data_buff**.

```
static char data_buff[1024];
struct sqlda *sql_descp;
struct sqlvar_struct * col_ptr;
short cnt, pos;
int size;
:
:

for(col_ptr=sql_descp->sqlvar, cnt=pos=0; cnt < sql_descp->sqld;
              cnt++, col_ptr++)
   {
   pos = (short)rtypalign(pos, col_ptr->sqltype);
   col_ptr->sqldata = &data_buf[pos];
   size = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);
   pos += size;
   }
```

Figure 17-2. Allocating Column Data from a Static Character Buffer

You can replace the code fragment in Figure 17-2 with a series of system memory-allocation calls within the **for** loop. However, system memory-allocation calls can be expensive so it is often more efficient to have a single memory allocation and then align pointers into that memory area.

When you allocate the column data, make sure that the allocated memory is formatted for the column data type. This data type is one of the Informix ESQL/C or SQL data types defined in the **sqltypes.h** header file. (See "Determining the Data Type of a Column" on page 15-13.) Make the allocated memory large enough to accommodate the maximum size of the data in the column.

You must also ensure that the data for each column begins on a proper word boundary in memory. On many hardware platforms, integer and other numeric data types must begin on a word boundary. The C language memory-allocation routines allocate memory that is suitably aligned for any data type, including structures, but the routines do not perform alignment for the constituent components of the structure.

Using the proper word boundaries assures that data types are machine independent. To assist you in this task, Informix ESQL/C provides the following memory-management functions:

- The **rtypalign()** function returns the position of the next proper word boundary for a specified data type.

  This function accepts two arguments: the current position in the data buffer and the integer Informix ESQL/C or SQL data type for which you want to allocate space.

- The **rtypmsize()** function returns the number of bytes of memory that you must allocate for the specified Informix ESQL/C or SQL data type.

  This function accepts two arguments: the integer Informix ESQL/C or SQL data type (in **sqltype**) and the length (in **sqllen**) for each column value.

These Informix ESQL/C library functions are described in more detail in Chapter 3, "Informix ESQL/C data types," on page 3-1.

When you allocate memory for the DATETIME or INTERVAL data types, you can take any of the following actions to set the qualifiers in the **dtime_t** and **intrvl_t** structures:

- Use the value that is in the associated **sqllen** field of **sqlda**.

- Compose a different qualifier with the values and macros that the **datatime.h** header file defines.
- Set the data-type qualifier to zero (0) and have the database server set this qualifier during the fetch. For DATETIME values, the data-type qualifier is the **dt_qual** field of the **dtime_t** structure. For INTERVAL values, the data-type qualifier is the **in_qual** field of the **intrvl_t** structure**.**

For examples that allocate memory for the sqldata fields, see the **demo3.ec** ("Executing a SELECT That Returns Multiple Rows" on page 17-11) and **unload.ec** demonstration programs that are supplied with Informix ESQL/C.

Assigning and Obtaining Values from an sqlda Structure

When you use the **sqlda** structure with dynamic SQL, you must transfer information in and out of it with C-language statements.

Assigning Values

To assign values to fields in the **sqlda** and **sqlvar_struct** structures, use regular C-language assignment to fields of the appropriate structure. For example:

```
da_ptr->sqld = 1;
da_ptr->sqlvar[0].sqldata = compny_data;
da_ptr->sqlvar[0].sqltype = SQLCHAR;   /* CHAR data type */
da_ptr->sqlvar[0].sqllen = 21;              /* column is CHAR(20) */
```

Set **sqlda** fields to provide values for input parameters in a WHERE clause ("Specifying Input Parameter Values" on page 17-8) or to modify the contents of a field after you use the DESCRIBE...INTO statement to fill the **sqlda** structure ("Allocating Memory for Column Data" on page 17-6).

Obtaining Values

To obtain values from the **sqlda** fields, you must also use regular C-language assignment from fields of the structure. For example:

```
count = da_ptr->sqld;
if(da_ptr->sqlvar[0].sqltype = SQLCHAR)
   size = da_ptr->sqlvar[0].sqllen + 1;
```

Typically, you obtain **sqlda** field values to examine descriptions of columns in a SELECT, INSERT, or EXECUTE FUNCTION statement. You might also need to access these fields to copy a column value that is returned by the database server from the **sqlda** structure into a host variable ("Putting Column Values into an sqlda Structure" on page 17-9).

The data type of the host variable must be compatible with the type of the associated field in the **sqlda** structure. When you interpret the **sqltype** field, make sure you use the data-type values that match your environment. For some data types, X/Open values differ from Informix values. For more information, see "Determining the Data Type of a Column" on page 15-13.

Specifying Input Parameter Values

Since the DESCRIBE...INTO statement does not analyze the WHERE clause, your program must explicitly allocate an **sqlda** structure and the **sqlvar_struct** structures (see "Allocating Memory for the sqlda Structure" on page 17-3). To describe the input parameters you must determine the number of input parameters and their data types and store this information in the allocated **sqlda** structure. For general information about how to define input parameters dynamically, see "Determining Unknown Input Parameters" on page 15-17.

When you execute a parameterized statement, you must include the USING DESCRIPTOR clause to specify the **sqlda** structure as the location of input parameter values, as follows:

- For input parameters in the WHERE clause of a SELECT statement, use the OPEN...USING DESCRIPTOR statement. This statement handles a sequential, scrolling, hold, or update cursor. If you are certain that the SELECT will return only *one* row, you can use the EXECUTE...INTO...USING SQL DESCRIPTOR statement instead of a cursor. For more information, see "Handling a Parameterized SELECT Statement" on page 17-20.

- For input parameters in the WHERE clause of a non-SELECT statement such as DELETE or UPDATE, use the EXECUTE...USING DESCRIPTOR statement. For more information, see "Handling a Parameterized UPDATE or DELETE Statement" on page 17-27.

- For input parameters in the VALUES clause of an INSERT statement, use the EXECUTE...USING SQL DESCRIPTOR statement. If the INSERT is associated with an insert cursor, use the PUT...USING DESCRIPTOR statement. For more information, see "Handling an unknown column list" on page 17-18.

Putting Column Values into an sqlda Structure

When you create a SELECT statement dynamically, you cannot use the INTO *host_var* clause of FETCH because you cannot name the host variables in the prepared statement. To fetch column values into an **sqlda** structure, use the USING DESCRIPTOR clause of FETCH instead of the INTO clause. The FETCH...USING DESCRIPTOR statement puts each column value into the **sqldata** field of its **sqlvar_struct** structure.

Using the FETCH...USING DESCRIPTOR statement assumes that a cursor is associated with the prepared statement. You must *always* use a cursor for SELECT statements and cursor functions (EXECUTE FUNCTION statements that return multiple rows). However, if either of these statements returns only one row, you can omit the cursor and retrieve the column values into an **sqlda** structure with the EXECUTE...INTO DESCRIPTOR statement.

Warning: If you execute a SELECT statement or user-defined function that returns more than one row and do not associate the statement with a cursor, your program generates a runtime error. When you associate a singleton SELECT (or EXECUTE FUNCTION) statement with a cursor, Informix ESQL/C does not generate an error. Therefore, it is a good practice always to associate a dynamic SELECT or EXECUTE FUNCTION statement with a cursor and to use a FETCH...USING DESCRIPTOR statement to retrieve the column values from this cursor into the **sqlda** structure.

Once the column values are in the **sqlda** structure, you can transfer the values from the **sqldata** fields to the appropriate host variables. You must use the **sqllen** and **sqltype** fields to determine, at runtime, the data types for the host variables. You might need to perform data type or length conversions between the SQL data types in the **sqltype** fields and the Informix ESQL/C data types that are needed for host variables that hold the returned value.

For more information about how to execute SELECT statements dynamically, see "Handling an Unknown Select List" on page 17-11. For more information about how to execute a user-defined function dynamically, see 17-16.

Freeing Memory Allocated to an sqlda Structure

Once you finish with an **sqlda** structure, free the associated memory. If you execute multiple DESCRIBE statements and you neglect to free the memory allocated by these statements, your application might run into memory constraints and the database server might exit.

If your application runs on a Windows operating system and uses the multi-threading library, use the Informix ESQL/C function **The SqlFreeMem() function** to release the memory that the **sqlda** structure occupies as in this example:

```
SqlFreeMem(sqlda_ptr, SQLDA_FREE);
```

Otherwise, use the standard C library **free()** function, as shown in this example:

```
free(sqlda_ptr);
```

If your Informix ESQL/C program executes a DESCRIBE statement multiple times for the same prepared statement and allocates a separate **sqlda** structure for each DESCRIBE, it must explicitly deallocate each **sqlda** structure. Figure 17-3 shows an example.

```
EXEC SQL prepare qid from 'select * from customer';
EXEC SQL describe qid into sqldaptr1;
EXEC SQL describe qid into sqldaptr2;
EXEC SQL describe qid into sqldaptr3;
    :
    ¹
free(sqldaptr1);
free(sqldaptr2);
free(sqldaptr3);
```

Figure 17-3. Deallocating Multiple sqlda Structures for the Same Prepared Statement

If your program allocated space for column data (see "Allocating Memory for Column Data" on page 17-6), you must also deallocate the memory allocated to the **sqldata** fields. For information about how to free other program resources, see "Freeing Resources" on page 14-7.

Using an sqlda Structure

Use an SQL descriptor-area (**sqlda**) structure to execute SQL statements that contain unknown values. Table 17-2 summarizes the types of dynamic statements that the remaining sections of this chapter cover.

Table 17-2. Using an sqlda Structure to Execute Dynamic SQL Statements

Purpose of the sqlda Structure	See
Holds select-list column values retrieved by a SELECT	17-11
Holds returned values from user-defined functions	17-16
Describes unknown columns in an INSERT	17-18
Describes input parameters in the WHERE clause of a SELECT	17-20
Describes input parameters in the WHERE clause of a DELETE or UPDATE	17-27

Handling an Unknown Select List

For an introduction on how to handle unknown columns in an unknown select list, see 15-16. This section describes how to use an **sqlda** structure to handle a SELECT statement.

To use an sqlda structure to handle unknown select-list columns:

1. Declare a variable to hold the address of an **sqlda** structure. For more information, see "Defining an sqlda Structure" on page 17-3.
2. Prepare the SELECT statement (with the PREPARE statement) to give it a statement identifier. The SELECT statement cannot include an INTO TEMP clause. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.
3. Use the DESCRIBE...INTO statement to perform two tasks:
   a. Allocate an **sqlda** structure. The address of the allocated structure is stored in the **sqlda** pointer that you declare. For more information, see "Allocating Memory for the sqlda Structure" on page 17-3.
   b. Determine the number and data types of select-list columns. The DESCRIBE statement fills an **sqlvar_struct** structure for each column of the select list. For more information, see "Initializing the sqlda Structure" on page 17-4.
4. Examine the **sqltype** and **sqllen** fields of **sqlda** for each select-list column to determine the amount of memory that you need to allocate for the data. For more information, see "Allocating Memory for Column Data" on page 17-6.
5. Save the number of select-list columns stored in the **sqld** field in a host variable.
6. Declare and open a cursor and then use the FETCH...USING DESCRIPTOR statement to fetch column values, one row at a time, into an allocated **sqlda** structure. See "Putting Column Values into an sqlda Structure" on page 17-9.
7. Retrieve the row data from the **sqlda** structure into host variables with C-language statements that access the **sqldata** field for each select-list column. For more information, see "Assigning and Obtaining Values from an sqlda Structure" on page 17-8.
8. Release memory allocated to the **sqldata** fields and the **sqlda** structure. For more information, see "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

Important: If the SELECT statement has input parameters of an unknown number and type in the WHERE clause, your program must also handle these input parameters with an **sqlda** structure.

For more information, see "Handling a Parameterized SELECT Statement" on page 17-20.

The following sections demonstrate how to use an **sqlda** structure to handle a SELECT statement that returns many rows and one that returns only one row.

Executing a SELECT That Returns Multiple Rows

The **demo3.ec** sample program, in this section, executes a dynamic SELECT statement with the following conditions:

- The SELECT returns more than one row.

The SELECT must be associated with a cursor, executed with the OPEN statement, and have its return values retrieved with the FETCH...USING DESCRIPTOR statement.

- The SELECT has either no input parameters or no WHERE clause.

  The OPEN statement does not need to include the USING clause.

- The SELECT has unknown columns in its select list.

  The FETCH statement includes the USING DESCRIPTOR clause to store the return values in an **sqlda** structure.

The **demo4** sample program ("Executing a SELECT That Returns Multiple Rows" on page 16-9) assumes these same conditions. While **demo4** uses a system-descriptor area to define the select-list columns, **demo3** uses an **sqlda** structure. The **demo3** program does *not* perform exception handling.

```
========================================================================
 1. #include <stdio.h>
 2. EXEC SQL include sqlda;
 3. EXEC SQL include sqltypes;
 4. main()
 5. {
 6. struct sqlda *demo3_ptr;
 7. struct sqlvar_struct *col_ptr;
 8. static char data_buff[1024];
 9. int pos, cnt, size;
10. EXEC SQL BEGIN DECLARE SECTION;
11.    int2 i, desc_count;
12.    char demoquery[80];
13. EXEC SQL END DECLARE SECTION;
14.    printf("DEMO3 Sample ESQL program running.\n\n");
15.    EXEC SQL connect to 'stores7';
========================================================================
```

Line 2

The program must include the Informix ESQL/C **sqlda.h** header file to provide a definition for the **sqlda** structure.

Lines 6 to 13

Lines 6 and 7 declare **sqlda** variables that are needed by the program. The **demo3_ptr** variable points to the **sqlda** structure that will hold the data that is fetched from the database. The **col_ptr** variable points to an **sqlvar_struct** structure so that the code can step through each of the **sqlvar_struct** structures in the variable-length portion of **sqlda**. Neither of these variables is declared as an Informix ESQL/C host variable. Lines 10 to 13 declare host variables to hold the data that is obtained from the user and the data that is retrieved from the **sqlda** structure.

```
========================================================================
16.    /* These next four lines have hard-wired both the query and
17.     * the value for the parameter. This information could have
18.     * been entered from the terminal and placed into the strings
19.     * demoquery and a query value string (queryvalue), respectively.
20.     */
21.    sprintf(demoquery, "%s %s",
22.       "select fname, lname from customer",
23.       "where lname < 'C' ");
24.    EXEC SQL prepare demo3id from :demoquery;
25.    EXEC SQL declare demo3cursor cursor for demo3id;
```

```
26.     EXEC SQL describe demo3id into demo3_ptr;
```
==

Lines 16 to 24

These lines assemble the character string for the SELECT statement (in **demoquery**) and prepare it as the **demo3id** statement identifier. For more information about these steps, see "Assembling and Preparing the SQL Statement" on page 14-2.

Line 25

This line declares the **demo3cursor** for the prepared statement identifier, **demo3id**.

Line 26

The DESCRIBE statement describes the select-list columns for the prepared statement that is in the **demo3id** statement identifier. For this reason, you must prepare the statement *before* you use DESCRIBE. This DESCRIBE includes the INTO clause to specify the **sqlda** structure to which **demo3_ptr** points as the location for these column descriptions. The DESCRIBE...INTO statement also allocates memory for an **sqlda** structure and stores the address of this structure in the **demo3_ptr** variable.

The **demo3** program assumes that the following SELECT statement is assembled at runtime and stored in the **demoquery** string:

```
SELECT fname, lname FROM customer WHERE lname < 'C'
```

After the DESCRIBE statement in line 26, the components of the **sqlda** structure contain the following:

- The **sqlda** component, **demo3_ptr->sqld**, has the value 2, since two columns were selected from the **customer** table.
- The component **demo3_ptr->sqlvar[0]**, an **sqlvar_struct** structure, contains information about the **fname** column of the **customer** table. The **demo3_ptr->sqlvar[0].sqlname** component, for example, gives the name of the first column (**fname**).
- The component **demo3_ptr->sqlvar[1]**, an **sqlvar_struct** structure, contains information about the **lname** column of the **customer** table.

==
```
27.     desc_count = demo3_ptr->sqld;
28.     printf("There are %d returned columns:\n", desc_count);
29.     /* Print out what DESCRIBE returns */
30.     for (i = 1; i <= desc_count; i++)
31.        prsqlda(i, demo3_ptr->sqlvar[i-1]);
32.     printf("\n\n");
```
==

Lines 27 and 28

Line 27 assigns the number of select-list columns that are found by the DESCRIBE statement to the **desc_count** host variable. Line 28 displays this information to the user.

Lines 29 to 32

This **for** loop goes through the **sqlvar_struct** structures for the columns of the select list. It uses the **desc_count** host variable to determine the number of these structures that are initialized by the DESCRIBE statement. For each **sqlvar_struct** structure, the **prsqlda()** function (line 31) displays information such as the data type, length, and name. For a description of **prsqlda()**, see the description of lines 75 to 81.

```
========================================================================
33.    for(col_ptr=demo3_ptr->sqlvar, cnt=pos=0; cnt < desc_count;
34.       cnt++, col_ptr++)
35.       {
36.       /* Allow for the trailing null character in C
37.          character arrays */
38.       if(col_ptr->sqltype==SQLCHAR)
39.          col_ptr->sqllen += 1;
40.       /* Get next word boundary for column data and
41.          assign buffer position to sqldata */
42.       pos = (int)rtypalign(pos, col_ptr->sqltype);
43.       col_ptr->sqldata = &data_buff[pos];
44.       /* Determine size used by column data and increment
45.          buffer position */
46.       size = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);
47.       pos += size;
48.       }
========================================================================
```

Lines 33 to 48

This second **for** loop allocates memory for the **sqldata** fields and sets the **sqldata** fields to point to this memory.

Lines 40 to 47 examine the **sqltype** and **sqllen** fields of **sqlda** for each select-list column to determine the amount of memory you need to allocate for the data. The program does not use **malloc()** to allocate space dynamically. Instead, it uses a static data buffer (the **data_buff** variable defined on line 8) to hold the column data. The Informix ESQL/C **rtypalign()** function (line 42) returns the position of the next word boundary for the column data type (in **col_ptr->sqltype**). Line 43 then assigns the address of this position within the **data_buff** data buffer to the **sqldata** field (for columns that receive values returned by the query).

The Informix ESQL/C **rtypmsize()** function (line 46) returns the number of bytes required for the SQL data type that is specified by the **sqltype** and **sqllen** fields. Line 47 then increments the data buffer pointer (**pos**) by the size required for the data. For more information, see "Allocating Memory for Column Data" on page 17-6.

```
========================================================================
49.    EXEC SQL open demo3cursor;
50.    for (;;)
51.       {
52.       EXEC SQL fetch demo3cursor using descriptor demo3_ptr;
53.       if (strncmp(SQLSTATE, "00", 2) != 0)
54.          break;
55.       /* Print out the returned values */
56.       for (i=0; i<desc_count; i++)
57.          printf("Column: %s\tValue:%s\n", demo3_ptr-
                  >sqlvar[i].sqlname,
58.             demo3_ptr->sqlvar[i].sqldata);
59.       printf("\n");
60.       }
========================================================================
```

Line 49

The database server executes the SELECT statement when it opens the **demo3cursor** cursor. If the WHERE clause of your SELECT statement contains

input parameters, you also need to specify the USING DESCRIPTOR clause of OPEN (see "Handling an unknown column list" on page 17-18).

Lines 50 to 60

This inner **for** loop executes for each row that is fetched from the database. The FETCH statement (line 52) includes the USING DESCRIPTOR clause to specify the **sqlda** structure to which **demo3_ptr** points as the location of the column values. After this FETCH, the column values are stored in the specified **sqlda** structure.

The **if** statement (lines 53 and 54) tests the value of the SQLSTATE variable to determine the success of the FETCH. If SQLSTATE indicates any status other than success, line 54 executes and ends the **for** loop. Lines 56 to 60 display the contents of the **sqlname** and **sqldata** fields for each column of the select list.

Important: The **demo3** program assumes that the returned columns are of character data type. If the program did not make this assumption, it would need to check the **sqltype** and **sqllen** fields to determine the appropriate data type for the host variable that holds the **sqldata** value.

```
=========================================================================
61.    if (strncmp(SQLSTATE, "02", 2) != 0)
62.        printf("SQLSTATE after fetch is %s\n", SQLSTATE);
63.    EXEC SQL close demo3cursor;

=========================================================================
```

Lines 61 and 62

Outside the **for** loop, the program tests the SQLSTATE variable again so that it can notify the user in the event of a successful execution, a runtime error, or a warning (class code not equal to "02").

Line 63

After all the rows are fetched, the CLOSE statement closes the **demo3cursor** cursor.

```
=========================================================================
64.    EXEC SQL free demo3id;
65.    EXEC SQL free demo3cursor;
66.    /* No need to explicitly free data buffer in this case because
67.     * it wasn't allocated with malloc(). Instead, it is a static char
68.     * buffer
69.     */
70.    /* Free memory assigned to sqlda pointer. */
71.    free(demo3_ptr);
72.    EXEC SQL disconnect current;
73.    printf("\nDEMO3 Sample Program Over.\n\n");
74. }
75. prsqlda(index, sp)
76. int2 index;
77. register struct sqlvar_struct *sp;
78. {
79.    printf("    Column %d: type = %d, len = %d, data = %s\n",
80.      index, sp->sqltype,  sp->sqllen, sp->sqldata, sp->sqlname);
81. }

=========================================================================
```

Lines 64 and 65

These FREE statements release the resources that are allocated for the **demo3id** prepared statement and the **demo3cursor** database cursor.

Lines 66 to 71

At the end of the program, free the memory allocated to the **sqlda** structure. Because this program does not use **malloc()** to allocate the data buffer, it does not use the **free()** system call to free the **sqldata** pointers. Although the allocation of memory from a static buffer is straightforward, it has the disadvantage that this buffer remains allocated until the program ends. For more information, see "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

The **free()** system call (line 71) frees the **sqlda** structure to which **demo3_ptr** points.

Lines 75 to 81

The **prsqlda()** function displays information about a select-list column. It reads this information from the **sqlvar_struct** structure whose address is passed into the function (**sp**).

Tip: The Informix ESQL/C demonstration programs **unload.ec** and **dyn_sql.ec** (described in "The dyn_sql Program" on page 16-28) also use **sqlda** to describe columns of a select list. Also see the PREPARE statement in the *IBM Informix Guide to SQL: Syntax*.

Executing a Singleton SELECT

The **demo3** program, described in the previous section, assumes that the SELECT statement returns more than one row and therefore the program associates the statement with a cursor. If you know at the time that you write the program that the dynamic SELECT *always* returns just one row, you can omit the cursor and use the EXECUTE...INTO DESCRIPTOR statement instead of the FETCH...USING DESCRIPTOR. You will still need to use the DESCRIBE statement to define the select-list columns. For more information, see "Putting Column Values into an sqlda Structure" on page 17-9.

Handling Unknown Return Values

For an introduction on how to handle unknown return values from a user-defined function, see "Determining Return Values Dynamically" on page 15-18. This section describes how to use an **sqlda** structure to save values that a dynamically executed user-defined function returns.

To use an sqlda structure to handle unknown-function return values:

1. Declare a variable to hold the address of an **sqlda** structure.

   For more information, see "Defining an sqlda Structure" on page 17-3.

2. Assemble and prepare an EXECUTE FUNCTION statement.

   The EXECUTE FUNCTION statement cannot contain the INTO clause. For more information, see "Assembling and Preparing the SQL Statement" on page 14-2.

3. Use the DESCRIBE...INTO statement to perform two tasks:

   a. Allocate an **sqlda** structure. The address of the allocated structure is stored in the **sqlda** pointer that you declare. For more information, see "Allocating Memory for the sqlda Structure" on page 17-3.

   b. Determine the number and data types of function return values. The DESCRIBE statement fills an **sqlvar_struct** structure for each return value. For more information, see "Initializing the sqlda Structure" on page 17-4.

4. After the DESCRIBE statement, you can test the SQLCODE variable (**sqlca.sqlcode**) for the defined constant SQ_EXECPROC to check for a prepared EXECUTE FUNCTION statement.

   The SQ_EXECPROC constant is defined in the **sqlstype.h** header file. For more information, see "Determining the Statement Type" on page 15-9.

5. Examine the **sqltype** and **sqllen** fields of **sqlda** for each return value to determine the amount of memory that you need to allocate for the data. For more information, see "Allocating Memory for Column Data" on page 17-6.

6. Execute the EXECUTE FUNCTION statement and store the return values in the **sqlda** structure.

   The statement you use to execute a user-defined function depends on whether the function is a noncursor function or a cursor function. The following sections discuss how to execute each type of function.

7. Deallocate any memory you allocated to the **sqlda** structure.

   See "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

Executing a Noncursor Function

A noncursor function returns only one row of return values to the application. Use the EXECUTE...INTO DESCRIPTOR statement to execute the function and save the return value(s) in an **sqlda** structure.

An external function that is not explicitly defined as an iterator function returns only a single row of data. Therefore, you can use EXECUTE...INTO DESCRIPTOR to execute most external functions dynamically and save their return values into an **sqlda** structure. This single row of data consists of only one value because an external function can only return a single value. The **sqlda** structure contains only one item descriptor with the single return value.

An SPL function whose RETURN statement does not include the WITH RESUME keywords returns only a single row of data. Therefore, you can use EXECUTE...INTO DESCRIPTOR to execute most SPL functions dynamically and save their return values into an **sqlda** structure. An SPL function can return one or more values at one time so the **sqlda** structure contains one or more item descriptors.

Warning: Because you usually do not know the number of returned rows that a user-defined function returns, you cannot guarantee that only one row will be returned. If you do not use a cursor to execute cursor function, Informix ESQL/C generates a runtime error. Therefore, it is a good practice to always associate a user-defined function with a function cursor.

Executing a Cursor Function

A cursor function can return one or more rows of return values to the application. To execute a cursor function, you must associate the EXECUTE FUNCTION statement with a function cursor and use the FETCH...INTO DESCRIPTOR statement to save the return value(s) in an **sqlda** structure.

To use an sqlda structure to hold cursor-function return values:

1. Declare a function cursor for the user-defined function.

   Use the DECLARE statement to associate the EXECUTE FUNCTION statement with a function cursor.

2. Use the OPEN statement to execute the function and open the cursor.

3. Use the FETCH...USING DESCRIPTOR statement to retrieve the return values from the cursor into the **sqlda** structure.

   For more information, see "Putting Column Values into an sqlda Structure" on page 17-9.
4. Retrieve the row data from the **sqlda** structure into host variables with C-language statements that access the **sqldata** field for each select-list column.

   For more information, see "Assigning and Obtaining Values from an sqlda Structure" on page 17-8.
5. Release memory allocated to the **sqldata** fields and the **sqlda** structure.

   For more information, see "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

Only an external function that is defined as an iterator function can return more than one row of data. Therefore, you must define a function cursor to execute an iterator function dynamically. Each row of data consists of only one value because an external function can only return a single value. For each row, the **sqlda** structure contains only one **sqlvar_struct** structure with the single return value.

An SPL function whose RETURN statement includes the WITH RESUME keywords returns can return one or more row of data. Therefore, you must define a function cursor to execute these SPL functions dynamically. Each row of data can consist of one or more values because an SPL function can return one or more values at one time. For each row, the **sqlda** structure contains an **sqlvar_struct** structure for each return value.

Handling an unknown column list

For an introduction on how to handle columns in a VALUES clause of an INSERT, see "Handling an Unknown Column List" on page 15-17. This section describes how to use an **sqlda** structure to handle the INSERT...VALUES statement.

To use an sqlda structure to handle input parameters in an INSERT:
1. Declare a variable to hold the address of an **sqlda** structure. For more information, see "Defining an sqlda Structure" on page 17-3.
2. Prepare the INSERT statement (with the PREPARE statement) and give it a statement identifier. See "Assembling and Preparing the SQL Statement" on page 14-2.
3. Use the DESCRIBE...INTO statement to perform two tasks:
   a. Allocate an **sqlda** structure. The address of the allocated structure is stored in the **sqlda** pointer that you declare. For more information, see "Allocating Memory for the sqlda Structure" on page 17-3.
   b. Determine the number and data types of columns in the table with the DESCRIBE...INTO statement. The DESCRIBE statement fills an **sqlvar_struct** structure for each item of the column list. For more information, see "Initializing the sqlda Structure" on page 17-4.
4. Examine the **sqltype** and **sqllen** fields of **sqlda** for each column to determine the amount of memory that you need to allocate for the data. For more information, see "Allocating Memory for Column Data" on page 17-6.
5. Save the number of columns stored in the **sqld** field in a host variable.
6. Set the columns to their values with C-language statements that set the appropriate **sqldata** fields in the **sqlvar_struct** structures of **sqlda**. A column value must be compatible with the data type of its associated column. If you

insert a null value, make sure to set the appropriate **sqlind** field to the address of an indicator variable that contains **-1**.

7. Execute the INSERT statement to insert the values into the database.

   The following sections demonstrate how to execute a simple INSERT statement that inserts only one row and an INSERT statement that uses an insert cursor to insert several rows from an insert buffer.

8. Release the memory that is allocated to the **sqldata** fields and the **sqlda** structure. For more information, see "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

Executing a Simple Insert

The following steps outline how to execute a simple INSERT statement with an **sqlda** structure:

1. Prepare the INSERT statement (with the PREPARE statement) and give it a statement identifier.

2. Set the columns to their values with C-language statements that set the appropriate **sqldata** fields in the **sqlvar_struct** structures of **sqlda**.

3. Execute the INSERT statement with the EXECUTE...USING DESCRIPTOR statement.

These steps are basically the same as those that handle an unknown select list of a SELECT statement ("Executing a Singleton SELECT" on page 17-16). The major difference is that because the statement is a not a SELECT statement, the INSERT does not require a cursor.

Executing an INSERT That Is Associated with a Cursor

You can also use an **sqlda** structure to handle an INSERT that is associated with an insert cursor. In this case, you do not execute the statement with the EXECUTE...USING DESCRIPTOR statement. Instead, you must declare and open an insert cursor and execute the insert cursor with the PUT...USING DESCRIPTOR statement, as follows:

1. Prepare the INSERT statement and associate it with an insert cursor with the DECLARE statement. All multirow INSERT statements *must* have a declared insert cursor.

2. Create the cursor for the INSERT statement with the OPEN statement.

3. Insert the first set of column values into the insert buffer with a PUT statement and its USING DESCRIPTOR clause. After this PUT statement, the column values stored in the specified **sqlda** structure are stored in the insert buffer. Repeat the PUT statement within a loop until there are no more rows to insert.

4. After all the rows are inserted, exit the loop and flush the insert buffer with the FLUSH statement.

5. Close the insert cursor with the CLOSE statement.

You handle the insert cursor in much the same way as you handle the cursor associated with a SELECT statement ("Executing a SELECT That Returns Multiple Rows" on page 17-11). For more information about how to use an insert cursor, see the PUT statement in the *IBM Informix Guide to SQL: Syntax*.

Handling a Parameterized SELECT Statement

For an introduction on how to determine input parameters, see "Determining Unknown Input Parameters" on page 15-17. This section describes how to handle a parameterized SELECT statement with an **sqlda** structure. If a prepared SELECT statement has a WHERE clause with input parameters of unknown number and data type, your Informix ESQL/C program must use an **sqlda** structure to define the input parameters.

To use an sqlda structure to define input parameters for a WHERE clause:

1. Declare a variable to hold the address of an **sqlda** structure. For more information, see "Defining an sqlda Structure" on page 17-3.

2. Determine the number and data types of the input parameters of the SELECT statement. For more information, see "Determining Unknown Input Parameters" on page 15-17.

3. Allocate an **sqlda** structure with a system memory-allocation function such as **malloc()**. For more information, see "Specifying Input Parameter Values" on page 17-8 and "Allocating Memory for the sqlda Structure" on page 17-3.

4. Indicate the number of input parameters in the WHERE clause with C-language statements that set the **sqld** field of the **sqlda** structure.

5. Store the definitions and values of each input parameter with C-language statements that set the **sqltype**, **sqllen**, and **sqldata** fields in the appropriate **sqlvar_struct** of the **sqlda** structure:

   a. The **sqltype** field uses the Informix ESQL/C data-type constants, which the **sqltypes.h** header file defines, to represent the data type of the input parameter. For more information, see "Determining the Data Type of a Column" on page 15-13.

   b. For a CHAR or VARCHAR value, **sqllen** is the size, in bytes, of the character array. For a DATETIME or INTERVAL value, this field stores the encoded qualifiers.

   c. The **sqldata** field of each **sqlvar_struct** structure contains the address of the memory allocated for the input parameter value. You might need to use the **sqltype** and **sqllen** fields for each input parameter to determine the amount of memory you need to allocate. For more information, see "Allocating Memory for Column Data" on page 17-6.

   If you use an indicator variable, also set the **sqlind** field and perhaps the **sqlidata**, **sqlilen**, and **sqlitype** fields (for non-X/Open applications only).

   Use an index into the **sqlda.sqlvar** array to identify the **sqlvar_struct** structure. For more information, see "Assigning and Obtaining Values from an sqlda Structure" on page 17-8.

6. Pass the defined input parameters from the **sqlda** structure to the database server with the USING DESCRIPTOR clause.

   The statement that provides the input parameters depends on how many rows the SELECT statement returns. The following sections discuss how to execute each type of SELECT statement.

7. Release the memory that you allocated for the **sqlvar_struct** fields, the **sqldata** fields, and the **sqlda** structure itself with the **free()** system call. For more information, see "Freeing Memory Allocated to an sqlda Structure" on page 17-10.

Important: If the SELECT statement has unknown columns in the select list, your program must also handle these columns with an **sqlda** structure. For more information, see "Handling an Unknown Select List" on page 17-11.

The following sections demonstrate how to use an **sqlda** structure to handle a parameterized SELECT statement that returns many rows and one that returns only a single row.

Executing a Parameterized SELECT That Returns Multiple Rows

The sample program described on the following pages is a modified version of the **demo4.ec** example program. It shows how to use a dynamic SELECT statement with the following conditions:

- The SELECT returns more than row.

  The SELECT must be associated with a cursor, executed with the OPEN statement, and have its return values retrieved with the FETCH...USING DESCRIPTOR statement.

- The SELECT has input parameters in its WHERE clause.

  The OPEN statement includes the USING DESCRIPTOR clause to provide the parameter values in an **sqlda** structure.

- The SELECT has unknown columns in the select list.

  The FETCH statement includes the USING DESCRIPTOR clause to store the return values in an **sqlda** structure.

The program illustrates how to use an **sqlda** structure to handle *both* input parameters of a WHERE clause and the columns in the select list.

```
========================================================================
1.  #include <stdio.h>
2.  EXEC SQL include sqlda;
3.  EXEC SQL include sqltypes;
4.  #define FNAME 15
5.  #define LNAME 15
6.  #define PHONE 18
========================================================================
```

Line 2
The program must include the Informix ESQL/C **sqlda.h** header file to use an **sqlda** structure.

```
========================================================================
 7.  main()
 8.  {
 9.      char fname[ FNAME + 1 ];
10.      char lname[ LNAME + 1 ];
11.      char phone[ PHONE + 1 ];
12.      int count, customer_num, i;
13.      struct sqlvar_struct *pos;
14.      struct sqlda *sqlda_ptr;
15.      printf("Sample ESQL program running.\n\n");
16.      EXEC SQL connect to 'stores7';
17.      stcopy("Carole", fname);
18.      stcopy("Sadler", lname);
19.      EXEC SQL prepare sql_id from
```

```
20.         'select * from customer where fname=? and lname=?';
21.     EXEC SQL declare slct_cursor cursor for sql_id;
```
==

Lines 9 to 14

Lines 9 to 11 declare variables to hold the data that is obtained from the user. The
sqlda_ptr variable (line 14) is the pointer to an **sqlda** structure. The **pos** variable
(line 13) points to an **sqlvar_struct** structure so that the code can proceed through
each of the **sqlvar_struct** structures in the variable-length portion of **sqlda**. Neither
of these variables is defined as an Informix ESQL/C host variable.

Lines 17 to 20

These lines assemble the character string for the SELECT statement and prepare
the SELECT string. This program assumes the number and data types of the input
parameters. Therefore, no C code needs to determine this information at runtime.
The question mark (?) indicates the input parameters in the WHERE clause. For
more information about these steps, see "Assembling and Preparing the SQL
Statement" on page 14-2.

Line 21

This line declares the **slct_cursor** cursor for the prepared statement identifier,
sql_id.

==
```
22.     count=2;
23.     whereClauseMem(&sqlda_ptr, count, fname, lname);
24.     EXEC SQL open slct_cursor using descriptor sqlda_ptr;
25.     free(sqlda_ptr->sqlvar);
26.     free(sqlda_ptr);
```
==

Lines 22 and 23

These lines initialize the **sqlda** structure with the input parameter information. The
program assumes two input parameters (line 22). If the number of input
parameters is unknown, the program needs to parse the SELECT character string
(not the prepared version) and count the number of "?" characters that it contains.

The program then calls the **whereClauseMem()** function to allocate and initialize
the **sqlda** structure. For more information, see lines 69 to 77.

Line 24

The database server executes the SELECT statement when it opens the cursor. You
must include the USING DESCRIPTOR clause of OPEN to specify the **sqlda**
structure as the location of the input parameter values.

Lines 25 and 26

Once the OPEN...USING DESCRIPTOR statement has executed, these input
parameter values have been used. Deallocate this **sqlda** structure because it is no
longer needed and so that it does not conflict with the **sqlda** that contains the
retrieved values. Keep in mind that this second **sqlda** must have memory allocated
before it can be used.

==
```
27.     EXEC SQL describe sql_id into sqlda_ptr;
28.     selectListMem(sqlda_ptr);
29.     while(1)
30.         {
31.         EXEC SQL fetch slct_cursor using descriptor sqlda_ptr;
```

```
32.      if(SQLCODE != 0)
33.         {
34.         printf("fetch SQLCODE %d\n", SQLCODE);
35.         break;
36.          }
```
==

Line 27

For demonstration purposes, this program assumes that the number and data
types of the select-list columns are also unknown at compile time. It uses the
DESCRIBE...INTO statement (line 27) to allocate an **sqlda** structure, and puts
information about the select-list columns into the structure to which **sqlda_ptr**
points.

Lines 28

The **selectListMem()** function handles the allocation of memory for column values.
For more information about **selectListMem()**, see lines 85 to 102.

Lines 29 to 31

The **while** loop executes for each row fetched from the database. The FETCH
statement (line 31) includes the USING DESCRIPTOR clause to specify an **sqlda**
structure as the location for the returned column values. For more information
about how to handle unknown select-list columns, see "Handling an Unknown
Select List" on page 17-11.

Lines 32 to 36

These lines test the value of the SQLCODE variable to determine if the FETCH
was successful. If SQLCODE contains a nonzero value, then the FETCH generates
the NOT FOUND condition (100) or an error (< 0). In any of these cases, line 34
prints out the SQLCODE value. To determine if the FETCH statement generated
warnings, you need to examine the **sqlca.sqlwarn** structure.

==
```
37.      for(i=0; i<sqlda_ptr->sqld; i++)
38.         {
39.         printf("\ni=%d\n", i);
40.         prsqlda(sqlda_ptr->sqlvar[i]);
41.         switch (i)
42.            {
43.          case 0:
44.             customer_num = *(int *)(sqlda_ptr->sqlvar[i].sqldata);
45.             break;
46.          case 1:
47.             stcopy(sqlda_ptr->sqlvar[i].sqldata, fname);
48.             break;
49.          case 2:
50.             stcopy(sqlda_ptr->sqlvar[i].sqldata, lname);
51.             break;
52.          case 9:
53.             stcopy(sqlda_ptr->sqlvar[i].sqldata, phone);
54.             break;
55.            }
56.         }
57.      printf("%d ==> |%s|, |%s|, |%s|\n",
58.         customer_num, fname, lname, phone);
59.      }
60.    EXEC SQL close slct_cursor;
```

```
61.    EXEC SQL free slct_cursor;
62.    EXEC SQL free sql_id;
```

==

Lines 37 to 59

These lines access the fields of the **sqlvar_struct** structure for each column in the
select list. The **prsqlda()** function (see lines 75 to 81) displays the column name
(from **sqlvar_struct.sqlname**) and its value (from the **sqlvar_struct.sqldata** field).
The **switch** (lines 41 to 55) transfers the column values from the **sqlda** structure
into host variables of the appropriate lengths and data types.

Lines 60 to 62

These lines free resources after all the rows are fetched. Line 60 closes the
slct_cursor cursor and line 61 frees it. Line 62 frees the **sql_id** statement ID.

==

```
63.    free(sqlda_ptr->sqlvar);
64.    free(sqlda_ptr);
65.    EXEC SQL close database;
66.    EXEC SQL disconnect current;
67.    printf("\nProgram Over.\n");
68. }
69. whereClauseMem(descp, count, fname, lname)
70.    struct sqlda **descp;
71.    int count;
72.    char *fname, *lname;
73. {
74.    (*descp)=(struct sqlda *) malloc(sizeof(struct sqlda));
75.    (*descp)->sqld=count;
76.    (*descp)->sqlvar=(struct sqlvar_struct *)
77.       calloc(count, sizeof(struct sqlvar_struct));
```

==

Lines 63 and 64

These **free()** system calls release the memory that is associated with the **sqlda**
structure. Line 63 releases the memory allocated to the **sqlvar_struct** structures.
Line 64 releases the memory allocated for the **sqlda** structure. The program does
not need to deallocate memory associated with the **sqldata** fields because these
fields have used space that is in a data buffer. For more information, see "Freeing
Memory Allocated to an sqlda Structure" on page 17-10.

Lines 69 to 77

The **whereClauseMem()** function initializes the **sqlda** structure with the
input-parameter definitions. Line 74 allocates memory for an **sqlda** structure to
hold the input parameters in the WHERE clause. Use of a DESCRIBE...INTO
statement to allocate an **sqlda** results in an **sqlda** that holds information about the
select-list columns of the SELECT. Because you want to describe the input
parameters in the WHERE clause, do not use DESCRIBE here.

Line 75 sets the **sqld** field of the **sqlda** structure to the value of **count** (2) to
indicate the number of parameters that are in the WHERE clause. Lines 76 and 77
use the **calloc()** system function to allocate the memory so that each input
parameter in the WHERE clause has an **sqlvar_struct** structure. These lines then
set the **sqlvar** field of the **sqlda** structure so that it points to this **sqlvar_struct**
memory.

==

```
78.    (*descp)->sqlvar[0].sqltype = CCHARTYPE;
```

```
79.     (*descp)->sqlvar[0].sqllen = FNAME + 1;
80.     (*descp)->sqlvar[0].sqldata = fname;
81.     (*descp)->sqlvar[1].sqltype = CCHARTYPE;
82.     (*descp)->sqlvar[1].sqllen = LNAME + 1;
83.     (*descp)->sqlvar[1].sqldata = lname;
84.  }
85.  selectListMem(descp)
86.     struct sqlda *descp;
87.  {
88.     struct sqlvar_struct *col_ptr;
89.     static char buf[1024];
90.     int pos, cnt, size;
91.     printf("\nWITHIN selectListMem: \n");
92.     printf("number of parms: %d\n", descp->sqld);
93.     for(col_ptr=descp->sqlvar, cnt=pos=0; cnt < descp->sqld;
94.        cnt++, col_ptr++)
95.        {
96.        prsqlda(col_ptr);
97.        pos = rtypalign(pos, col_ptr->sqltype);
98.        col_ptr->sqldata = &buf[pos];
99.        size = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);
100.        pos += size;
101.         }
102. }
```

==

Lines 78 to 84

Lines 78 to 80 set the **sqltype**, **sqllen**, and **sqldata** fields of the **sqlvar_struct** structure to describe the *first* input parameter: a character (CCHARTYPE) host variable of length 16 (FNAME + 1) whose data is stored in the **fname** buffer. The **fname** buffer is a character buffer declared in the **main()** program and passed as an argument to **whereClauseMem()**.

Lines 81 to 83 set the **sqltype**, **sqllen**, and **sqldata** fields of the **sqlvar_struct** structure to describe the *second* input parameter. This parameter is for the **lname** column. It is defined in the same way as the **fname** column (lines 78 to 80) but it receives its data from the **lname** buffer [also passed from **main()** to **whereClauseMem()**].

Lines 85 to 102

The **selectListMem()** function allocates the memory and initializes the **sqlda** structure for the unknown select-list columns of the parameterized SELECT statement. For more information about how to use an **sqlda** structure for select-list columns, see "Handling a Parameterized SELECT Statement" on page 17-20.

Executing a Parameterized Singleton SELECT

The instructions in the previous section assume that the parameterized SELECT statement returns more than one row and, therefore, is associated with a cursor. If you know at the time that you write the program that the parameterized SELECT statement *always* returns just one row, you can omit the cursor and use the EXECUTE...USING DESCRIPTOR...INTO statement instead of the OPEN...USING DESCRIPTOR statement to specify parameters values from an **sqlda** structure. For more information, see "Specifying Input Parameter Values" on page 17-8.

Handling a Parameterized User-Defined Routine

For an introduction on how to determine input parameters, see "Determining Unknown Input Parameters" on page 15-17. This section describes how to handle a parameterized user-defined routine with an **sqlda** structure. The following statements execute user-defined routines:

- The EXECUTE FUNCTION statement executes a user-defined function (external and SPL).
- The EXECUTE PROCEDURE statement executes a user-defined procedure (external and SPL).

If a prepared EXECUTE PROCEDURE or EXECUTE FUNCTION statement has arguments specified as input parameters of unknown number and data type, your Informix ESQL/C program can use an **sqlda** structure to define the input parameters.

Executing a Parameterized Function

You handle the input parameters of a user-defined function in the same way that you handle input parameters in the WHERE clause of a SELECT statement, as follows:

- Execute a *noncursor* function in the same way as a singleton SELECT statement.

  If you know at the time that you write the program that the dynamic user-defined function *always* returns just one row, you can use the EXECUTE...USING DESCRIPTOR...INTO statement to provide the argument values from an **sqlda** structure and to execute the function. For more information, see "Executing a Parameterized Singleton SELECT" on page 17-25.

- Execute a *cursor* function in the same way as a SELECT statement that returns one or more rows.

  If you are not sure at the time that you write the program that the dynamic user-defined function *always* returns just one row, define a function cursor and use the OPEN...USING DESCRIPTOR statement to provide the argument values from an **sqlda** structure. For more information, see "Executing a Parameterized SELECT That Returns Multiple Rows" on page 17-21.

The only difference between the execution of these EXECUTE FUNCTION and SELECT statements is that you prepare the EXECUTE FUNCTION statement for the noncursor function, instead of the SELECT statement.

Executing a Parameterized Procedure

To execute a parameterized user-defined procedure, you can use the EXECUTE...USING DESCRIPTOR statement to provide the argument values from an **sqlda** structure and to execute the procedure. You handle the input parameters of a user-defined procedure in the same way that you handle input parameters in a noncursor function. The only difference between the execution of the EXECUTE PROCEDURE statement and the EXECUTE FUNCTION statement (for a noncursor function) is that you do not need to specify the INTO clause of the EXECUTE...USING DESCRIPTOR statement for the user-defined procedure. For more information, see "Executing a Parameterized Function."

Handling a Parameterized UPDATE or DELETE Statement

The way to determine the input parameters in the WHERE clause of a DELETE or UPDATE statement is very similar to the way to determine them in the WHERE clause of a SELECT statement ("Handling a Parameterized SELECT Statement" on page 17-20). The major differences between these two types of dynamic parameterized statements are as follows:

- You do not need to use a cursor to handle a DELETE or UPDATE statement. You provide the parameter values from an **sqlda** structure with the USING DESCRIPTOR clause of the EXECUTE statement instead of with the OPEN statement.

- You can use the DESCRIBE...INTO statement to determine if the DELETE or UPDATE statement has a WHERE clause. For more information, see "Checking for a WHERE Clause" on page 15-15.

Appendix A. The ESQL/C example programs

Your IBM Informix software includes demonstration databases. IBM Informix ESQL/C also includes source files for many of the demonstration programs and examples in this publication, some of which access the demonstration databases.

In Windows environments, you can find the source files for Informix ESQL/C example programs in the **%INFORMIXDIR%\demo\esqldemo** directory.

On UNIX operating systems, you can find the source files for Informix ESQL/C example programs in the **$INFORMIXDIR/demo/esqlc** directory. The **esqldemo** script, which is included with Informix ESQL/C, copies the source files from the **$INFORMIXDIR/demo/esqlc** directory into the current directory.

For information about creating demonstration databases, see the *IBM Informix DB–Access User's Guide*.

Appendix B. ESQL/C function library

These topics describe the syntax and behavior of all the library functions provided with IBM Informix ESQL/C.

IBM Informix ESQL/C library functions

The following table lists the IBM Informix ESQL/C library functions in alphabetical order.

Function Name	Description	See
bigintcvasc()	Converts a C char type value to a BIGINT type number.	B-5
bigintcvdbl()	Converts a double type number to a BIGINT type number.	B-6
bigintcvdec()	Converts a decimal type number to a BIGINT type number.	B-6
bigintcvflt()	Converts a float type number to a BIGINT type number.	B-6
bigintcvifx_int8()	Converts and int8 type number to a BIGINT type number.	B-7
bigintcvint2()	Converts an int2 type number to a BIGINT type number.	B-7
bigintcvint4()	Converts an int4 type number to a BIGINT type number.	B-7
biginttoasc()	Converts a BIGINT type value to a C char type value.	B-8
biginttodbl()	Converts a BIGINT type number to a double type number.	B-8
biginttodec()	Converts a BIGINT type number to a decimal type number.	B-8
biginttoflt()	Converts a BIGINT type number to a float type number.	B-9
biginttoifx_int8()	Converts a BIGINT type number to an int8 type number.	B-9
biginttoint2()	Converts a BIGINT type number to an int2 type number.	B-9
biginttoint4()	Converts a BIGINT type number to an int4 type number.	B-10
bycmpr()	Compares two groups of contiguous bytes	B-10
bycopy()	Copies bytes from one area to another	B-12
byfill()	Fills the specified area with a character	B-13
byleng()	Counts the number of bytes in a string	B-14
decadd()	Adds two **decimal** numbers	B-15
deccmp()	Compares two **decimal** numbers	B-17
deccopy()	Copies a **decimal** number	B-18
deccvasc()	Converts a C **char** type to a **decimal** type	B-19
deccvdbl()	Converts a C **double** type to a **decimal** type	B-21
deccvint()	Converts a C **int2** type to a **decimal** type	B-24
deccvlong()	Converts a C **int4** type to a **decimal** type	B-25
decdiv()	Divides two **decimal** numbers	B-27
dececvt()	Converts a **decimal** value to an ASCII string	B-28
decfcvt()	Converts a **decimal** value to an ASCII string	B-28
decmul()	Multiplies two **decimal** numbers	B-32
decround()	Rounds a **decimal** number	B-34
decsub()	Subtracts two **decimal** numbers	B-35

Function Name	Description	See
dectoasc()	Converts a **decimal** type to an ASCII string	B-36
dectodbl()	Converts a **decimal** type to a C **double** type	B-39
dectoint()	Converts a **decimal** type to a C **int** type	B-40
dectolong()	Converts a **decimal** type to a C **long** type	B-42
dectrunc()	Truncates a **decimal** number	B-43
dtaddinv()	Adds an **interval** value to a **datetime** value	B-45
dtcurrent()	Gets current date and time	B-46
dtcvasc()	Converts an ANSI-compliant character string to **datetime**	B-47
dtcvfmtasc()	Converts a character string to a **datetime** value	B-49
dtextend()	Changes the qualifier of a **datetime**	B-52
dtsub()	Subtracts one **datetime** value from another	B-53
dtsubinv()	Subtracts an **interval** value from a **datetime** value	B-55
dttoasc()	Converts a **datetime** value to an ANSI-compliant character string	B-56
dttofmtasc()	Converts a **datetime** value to a character string	B-58
GetConnect()	Requests an explicit connection and returns a pointer to the connection information	B-60
ifx_cl_card()	Returns the cardinality of the specified collection type host variable	B-62
ifx_dececvt()	Converts a decimal value to an ASCII string (thread-safe version)	B-63
ifx_decfcvt()	Converts a decimal value to an ASCII string (thread-safe version)	B-63
ifx_getcur_conn_name()	Returns the name of the current connection	B-72
ifx_getenv()	Retrieves the value of an environment variable	B-71
ifx_getserial8()	Returns an inserted SERIAL8 value	B-72
ifx_int8add()	Adds two **int8** numbers	B-73
ifx_int8cmp()	Compares two **int8** numbers	B-75
ifx_int8copy()	Copies an **int8** number	B-76
ifx_int8cvasc()	Converts a C **char** type value to an **int8** type value	B-78
ifx_int8cvdbl()	Converts a C **double** type value to an **int8** type value	B-80
ifx_int8cvdec()	Converts a C **decimal** type value to an **int8** type value	B-81
ifx_int8cvflt()	Converts a C **float** type value to an **int8** type value	B-83
ifx_int8cvint()	Converts a C **int2** type value to an **int8** type value	B-84
ifx_int8cvlong()	Converts a C **int4** type value to an **int8** type value	B-86
ifx_int8div()	Divides two **int8** numbers	B-87
ifx_int8mul()	Multiplies two **int8** numbers	B-89
ifx_int8sub()	Subtracts two **int8** numbers	B-90
ifx_int8toasc()	Converts **an int8** type value to a text string	B-92
ifx_int8todbl()	Converts an **int8** type value to a C **double** type value	B-94
ifx_int8todec()	Converts an **int8** type value to a **decimal** type value	B-96
ifx_int8toflt()	Converts an **int8** type value to a C **float** type value	B-99
ifx_int8toint()	Converts an **int8** type value to a C **int2** type value	B-101
ifx_int8tolong()	Converts an **int8** type value to a C **int4** type value	B-103
ifx_lo_alter()	Alters the storage characteristics of an existing smart large object	B-106

Function Name	Description	See
ifx_lo_close()	Closes an open smart large object	B-107
ifx_lo_col_info()	Obtains column-level storage characteristics into an LO-specification structure	B-107
ifx_lo_copy_to_file()	Copies a smart large object to an operating-system file	B-108
ifx_lo_copy_to_lo()	Copies an operating-system file to an open smart large object	B-110
ifx_lo_create()	Creates an LO descriptor for a smart large object	B-111
ifx_lo_def_create_spec()	Allocates an LO-specification structure and initializes its fields to null values	B-112
ifx_lo_filename()	Returns the generated file name, given an LO descriptor and a file specification	B-113
ifx_lo_from_buffer()	Copies bytes from a user-defined buffer to a smart large object	B-114
ifx_lo_open()	Opens an existing smart large object	B-116
ifx_lo_read()	Reads a specified number of bytes from an open smart large object	B-118
ifx_lo_readwithseek()	Seeks to a specified position in an open smart large object and reads a specified number of bytes	B-119
ifx_lo_release()	Releases resources associated with a temporary smart large object	B-120
ifx_lo_seek()	Sets the seek position for the next read or write on an open smart large object	B-121
ifx_lo_spec_free()	Frees the resources allocated to an LO-specification structure	B-122
ifx_lo_specget_estbytes()	Gets the estimated number of bytes from the LO-specification structure	B-123
ifx_lo_specget_extsz()	Gets the allocation extent size from the LO-specification structure	B-124
ifx_lo_specget_flags()	Gets the create-time flags from the LO-specification structure	B-125
ifx_lo_specget_maxbytes()	Gets the maximum number of bytes from the LO-specification structure	B-126
ifx_lo_specget_sbspace()	Gets the name of the sbspace from the LO-specification structure	B-127
ifx_lo_specset_estbytes()	Sets the estimated number of bytes from the LO-specification structure	B-129
ifx_lo_specset_extsz()	Sets the allocation extent size in the LO-specification structure	B-130
ifx_lo_specset_flags()	Sets the create-time flags in the LO-specification structure	B-131
ifx_lo_specset_maxbytes()	Sets the maximum number of bytes in the LO-specification structure	B-132
ifx_lo_specset_sbspace()	Sets the name of the sbspace in the LO-specification structure	B-132
ifx_lo_stat()	Returns status information about an open smart large object	B-133
ifx_lo_stat_atime()	Returns the last access time for a smart large object	B-134
ifx_lo_stat_cspec()	Returns the storage characteristics into the LO-specification structure for a specified smart large object	B-135
ifx_lo_stat_ctime()	Returns the last change-in-status time for the smart large object	B-136
ifx_lo_stat_free()	Frees the resources allocated to an LO-status structure	B-137
ifx_lo_stat_mtime_sec()	Returns the last modification time, in seconds, for the smart large object	B-138
ifx_lo_stat_refcnt()	Returns the reference count for the smart large object	B-138
ifx_lo_stat_size()	Returns the size of the smart large object	B-139
ifx_lo_tell()	Returns the current seek position of an open smart large object	B-140
ifx_lo_to_buffer	Copies bytes from a smart large object into a user-defined buffer	B-141
ifx_lo_truncate()	Truncates a smart large object to a specific offset	B-142
ifx_lo_write()	Writes a specified number of bytes to an open smart large object	B-143

Function Name	Description	See
ifx_lo_writewithseek()	Seeks to a specified position in an open smart large object and writes a specified number of bytes	B-144
ifx_lvar_alloc()	Specifies whether to allocate memory when fetching **lvarchar** data	B-146
ifx_putenv()	Modifies or removes an existing environment variable or creates a new one	B-146
ifx_var_alloc()	Allocates memory for the data buffer	B-148
ifx_var_dealloc()	Deallocates memory for the data buffer	B-149
ifx_var_flag()	Determines whether Informix ESQL/C or the application handles memory allocation for the data buffer	B-150
ifx_var_getdata()	Returns the contents of the data buffer	B-151
ifx_var_getlen()	Returns the length of the data buffer	B-152
ifx_var_isnull()	Checks whether the data in the data buffer is null	B-153
ifx_var_setdata()	Sets the data for the data buffer	B-154
ifx_var_setlen()	Sets the length of the data buffer	B-154
ifx_var_setnull()	Sets the data in the data buffer to a null value	B-155
incvasc()	Converts an ANSI-compliant character string to an **interval** value	B-156
incvfmtasc()	Converts a character string to an **interval** value	B-158
intoasc()	Converts an **interval** value to an ANSI-compliant character string	B-160
intofmtasc()	Converts an **interval** value to a string	B-161
invdivdbl()	Divides an **interval** value by a numeric value	B-163
invdivinv()	Divides an **interval** value by an **interval** value	B-165
invextend()	Copies an **interval** value under a different qualifier	B-167
invmuldbl()	Multiplies an **interval** value by a numeric value	B-168
ldchar()	Copies a fixed-length string to a null-terminated string	B-170
rdatestr()	Converts an internal format to string	B-171
rdayofweek()	Returns the day of the week	B-172
rdefmtdate()	Converts a string to an internal format	B-174
rdownshift()	Converts all letters to lowercase	B-176
ReleaseConnect()	Closes an established explicit connection	B-177
rfmtdate()	Converts an internal format to a string	B-178
rfmtdec()	Converts a **decimal** type to a formatted string	B-181
rfmtdouble()	Converts a **double** type to a string	B-183
rfmtlong()	Converts an **int4** to a formatted string	B-185
rgetlmsg()	Retrieves the error message for a large error number	B-188
rgetmsg()	Retrieves the error message for an error number	B-189
risnull()	Checks whether a C variable is null	B-191
rjulmdy()	Returns month, day, and year from an internal format	B-193
rleapyear()	Determines whether a specified year is a leap year	B-195
rmdyjul()	Returns an internal format from month, day, and year	B-196
rsetnull()	Sets a C variable to null	B-197
rstod()	Converts a string to a **double** type	B-200
rstoi()	Converts a null-terminated string to an **int2**	B-201

Function Name	Description	See
rstol()	Converts a string to an **int4**	B-202
rstrdate()	Converts a string to an internal format	B-204
rtoday()	Returns a system date in internal format	B-205
rtypalign()	Aligns data on a proper type boundary	B-206
rtypmsize()	Gives the byte size of SQL data types	B-209
rtypname()	Returns the name of a specified SQL type	B-211
rtypwidth()	Gives minimum conversion byte size	B-213
rupshift()	Converts all letters to uppercase	B-209
SetConnect()	Switches the connection to an established (dormant) explicit connection	B-216
sqgetdbs()	Returns the names of the databases that a database server can access	B-217
sqlbreak()	Sends the database server a request to stop processing	B-220
sqlbreakcallback()	Provides a method of returning control to the application while it is waiting for the database server to process an SQL request	B-221
sqldetach()	Detaches a child process from a parent process	B-223
sqldone()	Determines whether the database server is currently processing an SQL request	B-228
sqlexit()	Terminates a database server process	B-228
SqlFreeMem()	Used in some cases to free memory allocated by the Informix ESQL/C libraries.	B-229
sqlsignal()	Performs signal handling and child-processes cleanup	B-229
sqlstart()	Starts a database server process	B-231
stcat()	Concatenates one string to another	B-232
stchar()	Copies a null-terminated string to a fixed-length string	B-233
stcmpr()	Compares two strings	B-234
stcopy()	Copies one string to another string	B-235
stleng()	Counts the number of bytes in a string	B-235

The bigintcvasc() function

The **bigintcvasc()** function converts a C char type value to a BIGINT type number.

Syntax:

```
mint bigintcvasc(strng_val, len, bigintp)
 const char *strng_val
 mint len
 bigint *bigintp
```

strng_val   A pointer to a string.

len   The length of the *strng_val* string.

bigintp   A pointer to a bigint variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bigintcvdbl() function

The **bigintcvdbl()** function converts a double type number to a BIGINT type number.

Syntax:

```
mint bigintcvdbl(dbl, bigintp)
 const double dbl
 bigint *bigintp
```

dbl   The double value to convert to bigint.

bigintp   A pointer to a bigint variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bigintcvdec() function

The **bigintcvdec()** function converts a decimal type number to a BIGINT type number.

Syntax:

```
mint bigintcvdec(decp, bigintp)
 const dec_t *decp
 bigint *bigintp
```

decp   A pointer to the decimal structure that contains the value to convert to a bigint value.

bigintp   A pointer to a bigint variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bigintcvflt() function

The **bigintcvflt()** function converts a float type number to a BIGINT type number.

Syntax:

```
mint bigintcvflt(dbl, bigintp)
 const double dbl
 bigint *bigintp
```

dbl   The float value to convert to bigint.

bigintp   A pointer to a bigint value to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bigintcvifx_int8() function

The **bigintcvifx_int8()** function converts and int8 type number to a BIGINT type number.

Syntax:
```
mint bigintcvifx_int8(int8p, bigintp)
 const ifx_int8_t *int8p
 bigint *bigintp
```

int8p   The int8 value to convert to a bigint value.

bigintp   A pointer to a bigint variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bigintcvint2() function

The **bigintcvint2()** function converts an int2 type number to a BIGINT type number.

Syntax:
```
mint bigintcvint2(int2v, bigintp)
 const int2 int2v
 bigint *bigintp
```

int2v   The int2 value to convert to a bigint value.

bigintp   A pointer to a bigint variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bigintcvint4() function

The **bigintcvint4()** function converts an int4 type number to a BIGINT type number.

Syntax:
```
mint bigintcvint4(int4v, bigintp)
 const int4 int4v
 bigint *bigintp
```

int4v   The int4 value to convert to a bigint value.

bigintp   A pointer to a bigint variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The biginttoasc() function

The **biginttoasc()** function converts a BIGINT type value to a C char type value.

Syntax:
```
mint biginttoasc(bigintv, strng_val, len, base)
 const bigint bigintv
 char *strng_val
 mint len
  mint base
```

bigintv	A bigint value to convert to a text string.
strng_val	A pointer to the first byte of the character buffer to contain the text string.
len	The size of *strng_val*, in bytes, minus 1 for the null terminator.
base	The numeric base of the output. Base 10 and 16 are supported. Other values result in base 10.

Return codes:

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

The biginttodbl() function

The **biginttodbl()** function converts a BIGINT type number to a double type number.

Syntax:
```
mint biginttodbl(bigintv, dbl)
 const bigint bigintv
 double *dbl
```

bigintv	A bigint value to convert to double.
dbl	A pointer to a double variable to contain the result of the conversion.

Return codes:

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

The biginttodec() function

The **biginttodec()** function converts a BIGINT type number to a decimal type number.

Syntax:
```
mint biginttodec(bigintv, decp)
 const bigint bigintv
 dec_t *decp
```

bigintv	A bigint value to convert to decimal.
decp	A pointer to a decimal variable to contain the result of the conversion.

Return codes:

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

The biginttoflt() function

The **biginttoflt()** function converts a BIGINT type number to a float type number.

Syntax:
```
mint biginttoflt(bigintv, fltp)
 const bigint bigintv
 float *fltp
```

bigintv   is a bigint value to convert to float.

fltp   is a pointer to a float variable to contain the result of the conversion.

Return codes:

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

The biginttoifx_int8() function

The **biginttoifx_int8()** function converts a BIGINT type number to an int8 type number.

Syntax:
```
void biginttoifx_int8(bigintv, int8p)
 const bigint bigintv
 ifx_int8_t *int8p
```

bigintv   A bigint value to convert to int8.

int8p   A pointer to an int8 structure to contain the result of the conversion.

The biginttoint2() function

The **biginttoint2()** function converts a BIGINT type number to an int2 type number.

Syntax:
```
mint biginttoint2(bigintv, int2p)
 const bigint bigintv
 int2 *int2p
```

bigintv   A bigint value to convert to an int2 integer value.

int2p   A pointer to an int variable to contain the result of the conversion.

Return codes:

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

The biginttoint4() function

The **biginttoint4()** function converts a BIGINT type number to an int4 type number.

Syntax:
```
mint biginttoint4(bigintv, int4p)
 const bigint bigintv
 int4 *int4p
```

bigintv   A bigint value to convert to an int4 integer value.

int4p   A pointer to an int4 variable to contain the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

The bycmpr() function

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

Syntax:
```
mint bycmpr(byte1, byte2, length)
   char *byte1;
   char *byte2;
   mint length;
```

byte1   A pointer to the location at which the first group of contiguous bytes starts.

byte2   A pointer to the location at which the second group of contiguous bytes starts.

length   The number of bytes that you want **bycmpr()** to compare.

Usage:

The **bycmpr()** function performs a byte-by-byte comparison of the two groups of contiguous bytes until it finds a difference or until it compares *length* number of bytes. The **bycmpr()** function returns an integer whose value (0, **-1**, or **+1**) indicates the result of the comparison between the two groups of bytes.

The **bycmpr()** function subtracts the bytes of the *byte2* group from those of the *byte1* group to accomplish the comparison.

Return codes:

0   The two groups are identical.

-1   The *byte1* group is less than the *byte2* group.

+1   The *byte1* group is greater than the *byte2* group.

Examples:

This sample program is in the **bycmpr.ec** file in the **demo** directory.

```
/*
 * bycmpr.ec *

   The following program performs three different byte comparisons with
   bycmpr() and displays the results.
*/

#include <stdio.h>

main()
{
    mint x;

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

    static mint number1 = 12345;
    static mint number2 = 12367;

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

    printf("BYCMPR Sample ESQL Program running.\n\n");

                    /* strings */
    printf("Comparing strings: String 1=%s\tString 2=%s\n", string1, string2);
    printf("  Executing: bycmpr(string1, string2, sizeof(string1))\n");
    x = bycmpr(string1, string2, sizeof(string1));
    printf("  Result = %d\n", x);

                   /* ints */
    printf("Comparing numbers: Number 1=%d\tNumber 2=%d\n", number1, number2);
    printf("  Executing: bycmpr( (char *) &number1, (char *) &number2, ");
    printf("sizeof(number1))\n");
    x = bycmpr( (char *) &number1, (char *) &number2, sizeof(number1));
    printf("  Result = %d\n", x);

                   /* non printable */
    printf("Comparing strings with non-printable characters:\n");
    printf("  Octal string 1=%o\tOctal string 2=%o\n", string3, string4);
    printf("  Executing: bycmpr(string3, string4, sizeof(string3))\n");
    x = bycmpr(string3, string4, sizeof(string3));
    printf("  Result = %d\n", x);

                 /* bytes */
    printf("Comparing bytes: Byte string 1=%c%c\tByte string 2=%c%c\n");
    printf("  Executing: bycmpr(&string1[2], &string2[2], 2)\n");
    x = bycmpr(&string1[2], &string2[2], 2);
    printf("  Result = %d\n", x);

    printf("\nBYCMPR Sample ESQL Program over.\n\n");
}
```

Output:

```
BYCMPR Sample ESQL Program running.

Comparing strings: String1=abcdef       String 2=abcdeg
  Executing: bycmpr(string1, string2, sizeof(string1))
  Result = -1
Comparing numbers: Number 1=12345       Number 2=12367
  Executing: bycmpr( (char *) &number1,  (char *) &number2, sizeof(number1)
  Result = -1
Comparing strings with non-printable characters:
```

```
    Octal string 1=40300      Octal string 2=40310
    Executing: bycmpr(string3, string4, sizeof(string3))
    Result = 1
Comparing bytes: Byte string 1=cd       Byte string 2=cd
    Executing: bycmpr(&string1[2], &string2[2], 2)
    Result = 0

BYCMPR Sample ESQL Program over.
```

The bycopy() function

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

Syntax:
```
void bycopy(from, to, length)
    char *from;
    char *to;
    mint length;
```

from   A pointer to the first byte of the group of bytes that you want
                bycopy() to copy.

to   A pointer to the first byte of the destination group of bytes. The
                memory area to which *to* points can overlap the area to which the
                from argument points. In this case, Informix ESQL/C does not
                preserve the value to which *from* points.

length   The number of bytes that you want **bycopy()** to copy.

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

Examples:

This sample program is in the **bycopy.ec** file in the **demo** directory.
```
/*
 *  * bycopy.ec *

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

#include <stdio.h>

char dest[20];

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

    printf("BYCOPY Sample ESQL Program running.\n\n");

    printf("String 1=%s\tString 2=%s\n", string1, string2);
    printf("  Copying String 1 to destination string:\n");
    bycopy(string1, dest, strlen(string1));
    printf("  Result = %s\n\n", dest);

    printf("  Copying String 2 to destination string:\n");
```

```
        bycopy(string2, dest, strlen(string2));
        printf("  Result = %s\n\n", dest);

        printf("Number 1=%d\tNumber 2=%d\n", number1, number2);
        printf("  Copying Number 1 to Number 2:\n");
        bycopy( (char *) &number1, (char *) &number2, sizeof(int));
        printf("  Result = number1(hex) %08x, number2(hex) %08x\n",
                number1, number2);

        printf("\nBYCOPY Sample Program over.\n\n");
}
```

Output:

```
BYCOPY Sample ESQL Program running.

String 1=abcdef  String2=abcdefghijklmn
  Copying String 1 to destination string:
  Result  = abcdef

  Copying String 2 to destination string:
  Result = abcdefghijklmn

Number 1=12345  Number2=0
  Copying Number 1 to Number 2:
  Result = number1(hex) 00003039, number2(hex) 00003039

BYCOPY Sample Program over.
```

The byfill() function

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

Syntax:

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

to	A pointer to the first byte of the memory area that you want **byfill()** to fill.
length	The number of times that you want **byfill()** to repeat the character within the area.
ch	The character that you want **byfill()** to use to fill the area.
	Important: Take care not to overwrite areas of memory adjacent to the area that you want **byfill()** to fill.

Examples:

This sample program is in the **byfill.ec** file in the **demo** directory.

```
/*
 * 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";

    printf("BYFILL Sample ESQL Program running.\n\n");

    printf("String = %s\n", area);

    printf("\nFilling string with five 's' characters:\n");
    byfill(area, 5, 's');
    printf("Result = %s\n", area);

    printf("\nFilling string with two 's' characters starting at ");
    printf("position 16:\n");
    byfill(&area[16], 2, 's');
    printf("Result = %s\n", area);

    printf("Filling entire string with 'b' characters:\n");
    byfill(area, sizeof(area)-1, 'b');
    printf("Result = %s\n", area);

    printf("\nBYFILL Sample Program over.\n\n");
}
```

Output:
```
BYFILL Sample ESQL Program running.

String = xxxxxxxxxxxxxxxxxxxx

Filling string with five 's' characters:
Result = sssssxxxxxxxxxxxxxxx

Filling string with two 's' characters starting at position 16:
Result = sssssxxxxxxxxxxxssx

Filling entire string with 'b' characters:
Result = bbbbbbbbbbbbbbbbbbb

BYFILL Sample Program over.
```

The byleng() fucntion

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

Syntax:
```
mint byleng(from, count)
    char *from;
    mint count;
```

from   A pointer to a fixed-length string (not null-terminated).

count   The number of bytes in the fixed-length string. This does not include trailing blanks.

Examples:

This sample program is in the **byleng.ec** file in the **demo** directory.
```
/*
 * byleng.ec *

The following program uses byleng() to count the significant characters
```

```
    in an area.
*/

#include <stdio.h>

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


    printf("BYLENG Sample Program running.\n\n");

            /* initial length */
    printf("Initial string:\n");
    x = byleng(area, 15);
    printf("  Length =  %d, String = '%s'\n", x, area);

            /* after copy */
    printf("\nAfter copying two 's' characters starting ");
    printf("at position 16:\n");
    bycopy("ss", &area[16], 2);
    x = byleng(area, 19);
    printf("  Length = %d, String = '%s'\n", x, area);

    printf("\nBYLENG Sample Program over.\n\n");
}
```

Output:

```
BYLENG Sample Program running.

Initial string:
  Length = 10, String = 'xxxxxxxxxx         '

After copying two 's' characters starting at position 16:
  Length = 18, String =  'xxxxxxxxxx      ss '

BYLENG Sample Program over.
```

The decadd() function

The **decadd()** function adds two **decimal** type values.

Syntax:

```
mint decadd(n1, n2, sum)
    dec_t *n1;
    dec_t *n2;
    dec_t *sum;
```

n1   A pointer to the **decimal** structure of the first operand.

n2   A pointer to the **decimal** structure of the second operand.

sum   A pointer to the **decimal** structure that contains the sum *(n1 + n2)*.

Usage:

The *sum* 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.

Examples:

The file **decadd.ec** in the **demo** directory contains the following sample program.

```
/*
   * decadd.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "  1000.6789"; /* leading spaces will be ignored */
char string2[] = "80";
char result[41];

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

    printf("DECADD Sample ESQL Program running.\n\n");

    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);
      }
    result[40] = '\0';
    printf("\t%s + %s = %s\n", string1, string2, result); /* display result */

    printf("\nDECADD Sample Program over.\n\n");
    exit(0);
}
```

Output:
```
DECADD Sample ESQL Program running.

    1000.6789 + 80 = 1080.6789

DECADD Sample Program over.
```

The deccmp() function

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

Syntax:
```
mint deccmp(n1, n2)
   dec_t *n1;
   dec_t *n2;
```

n1   A pointer to a **decimal** structure of the first number to compare.

n2   A pointer to a **decimal** structure of the second number to compare.

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.

Examples:

The file **deccmp.ec** in the **demo** directory contains the following sample program.

```
/*
   * deccmp.ec *

   The following program compares DECIMAL numbers and displays the results.
*/

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "-12345.6789"; /* leading spaces will be ignored */
char string2[] = "12345.6789";
char string3[] = "-12345.6789";
char string4[] = "-12345.6780";


main()
{
    mint x;
    dec_t num1, num2, num3, num4;

    printf("DECCOPY Sample ESQL Program running.\n\n");

    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("Number 1 = %s\tNumber 2 = %s\n", string1, string2);
        printf("Number 3 = %s\tNumber 4 = %s\n",string3, string4);
        printf("\nExecuting: deccmp(&num1, &num2)\n");
        printf("   Result = %d\n", deccmp(&num1, &num2));
        printf("Executing: deccmp(&num2, &num3)\n");
        printf("   Result = %d\n", deccmp(&num2, &num3));
        printf("Executing: deccmp(&num1, &num3)\n");
        printf("   Result = %d\n", deccmp(&num1, &num3));
        printf("Executing: deccmp(&num3, &num4)\n");
        printf("   Result = %d\n", deccmp(&num3, &num4));

        printf("\nDECCMP Sample Program over.\n\n");
        exit(0);
}
```

Output:
```
DECCMP Sample ESQL Program running.

Number 1 = -12345.6789       Number 2 = 12345.6789
Number 3 = -12345.6789       Number 4 = -12345.6780

Executing: deccmp(&num1, &num2)
  Result = -1
Executing: deccmp(&num2, &num3)
  Result = 1
Executing: deccmp(&num1, &num3)
  Result = 0
Executing: deccmp(&num3, &num4)
  Result = -1

DECCMP Sample Program over.
```

The deccopy() function

The **deccopy()** function copies one **decimal** structure to another.

Syntax:
```
void deccopy(source, target)
   dec_t *source;
   dec_t *target;
```

source   A pointer to the value held in the source **decimal** structure.

target   A pointer to the target **decimal** structure.

The **deccopy()** function does not return a status value. To determine the success of the copy operation, look at the contents of the **decimal** structure to which the *target* argument points.

Examples:

The file **deccopy.ec** in the **demo** directory contains the following sample program.

```
/*
 * deccopy.ec *

   The following program copies one DECIMAL number to another.
*/

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "12345.6789";
char result[41];

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

    printf("DECCOPY Sample ESQL Program running.\n\n");

    printf("String = %s\n", string1);
    if (x = deccvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to DECIMAL\n", x);
        exit(1);
        }
    printf("Executing: deccopy(&num1, &num2)\n");
    deccopy(&num1, &num2);
    if (x = dectoasc(&num2, result, sizeof(result), -1))
        {
        printf("Error %d in converting num2 to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("Destination = %s\n", result);

    printf("\nDECCOPY Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
DECCOPY Sample ESQL Program running.

String = 12345.6789
Executing: deccopy(&num1, &num2)
Destination = 12345.6789

DECCOPY Sample Program over.
```

The deccvasc() function

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

Syntax:

```
mint deccvasc(strng_val, len, dec_val)
    char *strng_val;
    mint len;
    dec_t *dec_val;
```

strng_val   A pointer to a string whose value **deccvasc()** converts to a **decimal** value.

len	The length of the strng_val string.
dec_val	A pointer to the **decimal** structure where **deccvasc()** places the result of the conversion.

Usage:

The character string, *strng_val*, can contain the following symbols:

- A leading sign, either a plus (+) or minus (-)
- A decimal point, and digits to the right of the decimal point
- An exponent that is preceded by either e or E. You can precede the exponent by a sign, either a plus (+) or minus (-).

The **deccvasc()** function ignores leading spaces in the character string.

Return codes:

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

Examples:

The **deccvasc.ec** file in the **demo** directory contains the following sample program.

```
/*
   * 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>

EXEC SQL include decimal;

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

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

    printf("DECCVASC Sample ESQL Program running.\n\n");

    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);

        printf("\nDECCVASC Sample Program over.\n\n");
        exit(0);
}


disp_dec(s, num)
char *s;
dec_t *num;
{
    mint n;

    printf("%s dec_t structure:\n", s);
    printf("\tdec_exp = %d, dec_pos = %d, dec_ndgts = %d, dec_dgts: ",
      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");
}
```

Output:

```
DECCVASC Sample ESQL Program running.

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

string2 =   480
num2 dec_t structure:
        dec_exp = 2, dec_pos =  1, dec_ndgts = 2, dec_dgts: 04 80

DECCVASC Sample Program over.
```

The deccvdbl() function

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

Syntax:

```
mint deccvdbl(dbl_val, np)
   double dbl_val;
   dec_t *dec_val;
```

dbl_val   The **double** value that **deccvdbl()** converts to a **decimal** type value.

dec_val   A pointer to a **decimal** structure where **deccvdbl()** places the result
                 of the conversion.

Return codes:

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

Examples:

The **deccvdbl.ec** file in the **demo** directory contains the following sample program.

```
/*
   * deccvdbl.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

char result[41];


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

    printf("DECCVDBL Sample ESQL Program running.\n\n");

    printf("Number 1 (double) = 1234.5678901234\n");
    if (x = deccvdbl((double)1234.5678901234, &num))
      {
      printf("Error %d in converting double1 to DECIMAL\n", x);
      exit(1);
      }
    if (x = dectoasc(&num, result, sizeof(result), -1))
      {
      printf("Error %d in converting DECIMAL1 to string\n", x);
      exit(1);
      }
    result[40] = '\0';
    printf("  String Value = %s\n", result);

    printf("Number 2 (double) = $.1f\n", d);
    if (x = deccvdbl(d, &num))
      {
      printf("Error %d in converting double2 to DECIMAL\n", x);
      exit(1);
      }
    if (x = dectoasc(&num, result, sizeof(result), -1))
      {
      printf("Error %d in converting DECIMAL2 to string\n", x);
      exit(1);
      }
    result[40] = '\0';
    printf("  String Value = %s\n", result);

    printf("\nDECCVDBL Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
DECCVDBL Sample ESQL Program running.

Number 1 (double) = 1234.5678901234
   String Value = 1234.5678901234
Number 2 (double) = 2147483647.0
   String Value =  2147483647.0

DECCVDBL Sample Program over.
```

The deccvflt() function

The **deccvflt()** function converts a C float type number into an ESQL/C decimal type number.

Syntax:
```
int deccvflt(flt_val, dec_val)

float flt_val;

dec_t *dec_val;
```

flt_val   The float value that **deccvflt()** converts to a decimal type value.

dec_val   A pointer to a decimal structure where **deccvflt()** places the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The following example program converts two float type number to DECIMAL numbers and displays the results.

```
#include <stdio.h>
EXEC SQL include decimal;
char result[41];
main()
{
   int x;
dec_t num;
float f = 2147483674;
printf("DECCVFLT Sample ESQL Program Running.\n\n);
if (x = deccvflt((float)1234.5678901234, &num))
   {
   printf("Error %d in converting double1 to DECIMAL\n", x);
   exit(1);
   }
if (x = dectoasc(&num, result, sizeof(result), -1))
   {
   printf("Error %d in converting DECIMAL1 to string\n", x);
   exit(1);
   }
result[40] = '\0';
printf(" String Value = %s\n", result);
printf(" Number 2 (float) = %.1f\n", f);
if (x = deccvflt(f, &num))
   {
   printf("Error %d in converting float2 to DECIMAL\n", x);
   exit(1);
   }
if (x = dectoasc(&num, result, sizeof(result), -1))
```

```
                  {
                  printf("Error %d in converting DECIMAL2 to string\n", x);
                  exit(1);
                  }
          result[40] = '\0';
          printf(" String Value = %s\n", result);
          printf("\n DECCVFLT Sample Program Over.\n\n");
          exit(0);
          }
```

Output:

```
DECCVFLT Sample ESQL Program running.

Number 1 (float) = 1234.5678901234
  String Value = 1234.56787
Number 2 (float) = 2147483647.0
  String Value =  2147483647.0

DECCVFLT Sample Program over.
```

The deccvint() function

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

Syntax:

```
mint deccvint(int_val, dec_val)
    mint int_val;
    dec_t *dec_val;
```

int_val   The **mint** value that **deccvint()** converts to a **decimal** type value.

dec_val   A pointer to a **decimal** structure where **deccvint()** places the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The **deccvint.ec** file in the **demo** directory contains the following sample program.

```
/*
   * deccvint.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

char result[41];

main()
{
    mint x;
    dec_t num;

    printf("DECCVINT Sample ESQL Program running.\n\n");
```

```
    printf("Integer 1 = 129449233\n");
    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);
       }
    result[40] = '\0';
    printf("  String for Decimal Value = %s\n", result);

    printf("Integer 2 = 33\n");
    if (x = deccvint(33, &num))
       {
       printf("Error %d in converting int2 to DECIMAL\n", x);
       exit(1);
       }
    result[40] = '\0';
    printf("  String for Decimal Value = %s\n", result);

    printf("\nDECCVINT Sample Program over.\n\n");
    exit(0);
}
```

Output:
```
DECCVINT Sample ESQL Program running.

Integer 1 = 129449233
  String for Decimal Value = 129449233.0
Integer 2 = 33
  String for Decimal Value = 33.0

DECCVINT Sample Program over.
```

The deccvlong() function

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

Syntax:
```
mint deccvlong(lng_val, dec_val)
    int4 lng_val;
    dec_t *dec_val;
```

lng_val   The **int4** value that **deccvlong()** converts to a **decimal** type value.

dec_val   A pointer to a **decimal** structure where **deccvint()** places the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **deccvlong.ec** in the **demo** directory contains the following sample program.

```
/*
 * deccvlong.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

char result[41];
main()
{
    mint x;
    dec_t num;

  int4 n;

   printf("DECCVLONG Sample ESQL Program running.\n\n");

   printf("Long Integer 1 = 129449233\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);
      }
   result[40] = '\0';
   printf("  String for Decimal Value = %s\n", result);

   n = 2147483646;                                  /* set n */
   printf("Long Integer 2 = %d\n", 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);
      }
   result[40] = '\0';
   printf("  String for Decimal Value = %s\n", result);

   printf("\nDECCVLONG Sample Program over.\n\n");
   exit(0);
}
```

Output:

```
DECCVLONG Sample ESQL Program running.

Long Integer 1 = 129449233
  String for Decimal Value =  129449233.0
Long Integer 2 = 2147483646
  String for Decimal Value = 2147483646.0

DECCVLONG Sample Program over.
```

The decdiv() function

The **decdiv()** function divides two **decimal** type values.

Syntax:
```
mint decdiv(n1, n2, result)    /* result = n1 / n2 */
   dec_t *n1;
   dec_t *n2;
   dec_t *result;
```

n1   A pointer to the **decimal** structure of the first operand.

n2   A pointer to the **decimal** structure of the second operand.

quotient   A pointer to the **decimal** structure that contains the quotient of *n1* divided by *n2*.

Usage:

The *quotient* 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.

Examples:

The file **decdiv.ec** in the **demo** directory contains the following sample program.

```
/*
   * decdiv.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "480";
char string2[] = "80";
char result[41];

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

    printf("DECDIV Sample ESQL Program running.\n\n");

    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 = dectoasc(&dvd, result, sizeof(result), -1))
        {
        printf("Error %d in converting dividend to string\n", x);
        exit(1);
        }
     result[40] = '\0';
     printf("\t%s / %s = %s\n", string1, string2, result);

     printf("\nDECDIV Sample Program over.\n\n");
     exit(0);
}
```

Output:
```
DECDIV Sample ESQL Program running.

    480 / 80 = 6.0

DECDIV Sample Program over.
```

The dececvt() and decfcvt() functions

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 the **ecvt(3) function**, and the **decfcvt()** function works in the same fashion as the **fcvt(3) function**. They both convert a **decimal** type number to a C **char** type value.

Syntax:
```
char *dececvt(dec_val, ndigit, decpt, sign)
   dec_t *dec_val;
   mint ndigit;
   mint *decpt;
   mint *sign;

char *decfcvt(dec_val, ndigit, decpt, sign)
   dec_t *dec_val;
   mint ndigit;
   mint *decpt;
   mint *sign;
```

dec_val	A pointer to a **decimal** structure that contains the **decimal** value you want these functions to convert.
ndigit	The length of the ASCII string for **dececvt()**. It is the number of digits to the right of the decimal point for **decfcvt()**.
decpt	A pointer to an integer that is the position of the decimal point relative to the start of the string. A negative or zero value for *decpt means to the left of the returned digits.
sign	A pointer to the sign of the result. If the sign of the result is negative, *sign is nonzero; otherwise, *sign* is zero.

Usage:

The **dececvt()** function converts the **decimal** value to which *np* points into a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string. A subsequent call to this function overwrites the string.

The **dececvt()** function rounds low-order digits.

The **decfcvt()** function is identical to **dececvt()**, except that *ndigit* specifies the number of digits to the right of the decimal point instead of the total number of digits.

Let *dec_val* point to a **decimal** value of 12345.67 and suppress all arguments except *ndigit*. The following table shows the values that the **dececvt()** function returns for four different *ndigit* values.

ndigit value	Return string	*decpt value	*sign
4	"1235"	5	0
10	"1234567000"	5	0
1	"1"	5	0
3	"123"	5	0

For more examples of *dec_val* and *ndigit* values, see the sample output of the **dececvt.ec** demonstration program on B-29.

Warning: When you write thread-safe Informix ESQL/C applications, do not use the **dececvt()** or **decfcvt()** library functions. Instead, use their thread-safe equivalents, "The ifx_dececvt() and ifx_decfcvt() function" on page B-63 For more information, see Chapter 13, "Using Informix libraries," on page 13-1

Example of dececvt():

The file **dececvt.ec** in the **demo** directory contains the following sample program.

```
/*
    * dececvt.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.
*/

#include <stdio.h>

EXEC SQL include decimal;


char *strings[] =
    {
    "210203.204",
    "4894",
    "443.334899312",
    "-12344455",
    "12345.67",
    ".001234",
    0
    };

char result[40];
```

```
main()
{
    mint x;
    mint i = 0, f, sign;
    dec_t num;
    char *dp, *dececvt();

    printf("DECECVT Sample ESQL Program running.\n\n");
    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;
            }
        printf("\Input string[%d]: %s\n", i, strings[i]);

        dp = dececvt(&num, 20, &f, &sign);     /* to 20-char ASCII string */
        printf(" Output of dececvt(&num, 20, ...): %c%s  decpt: %d  sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvt(&num, 10, &f, &sign);     /* to 10-char ASCII string */
        /* display result */
        printf(" Output of dececvt(&num, 10, ...): %c%s  decpt: %d  sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvt(&num, 4, &f, &sign);     /* to 4-char ASCII string */
        /* display result */
        printf(" Output of dececvt(&num, 4, ...): %c%s  decpt: %d  sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvt(&num, 3, &f, &sign);     /* to 3-char ASCII string */
        /* display result */
        printf(" Output of dececvt(&num, 3, ...): %c%s  decpt: %d  sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);
        dp = dececvt(&num, 1, &f, &sign);     /* to 1-char ASCII string */
        /* display result */
        printf(" Output of dececvt(&num, 1, ...): %c%s  decpt: %d  sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        ++i;                                         /* next string */
        }

    printf("\nDECECVT Sample Program over.\n\n");
}
```

Output of dececvt():

```
DECECVT Sample ESQL Program running.


Input string[0]: 210203.204
 Output of dececvt: +2102  decpt: 6  sign: 0
 Output of dececvt: +2102032040  decpt: 6  sign: 0
 Output of dececvt: +2  decpt: 6  sign: 0
 Output of dececvt: +210  decpt: 6  sign: 0

Input string[1]: 4894
 Output of dececvt: +4894  decpt: 4  sign: 0
 Output of dececvt: +4894000000  decpt: 4  sign: 0
 Output of dececvt: +5  decpt: 4  sign: 0
 Output of dececvt: +489  decpt: 4  sign: 0
```

```
Input string[2]: 443.334899312
 Output of dececvt: +4433   decpt: 3   sign: 0
 Output of dececvt: +4433348993   decpt: 3   sign: 0
 Output of dececvt: +4   decpt: 3   sign: 0
 Output of dececvt: +443   decpt: 3   sign: 0

Input string[3]: -12344455
 Output of dececvt: -1234   decpt: 8   sign: 1
 Output of dececvt: -1234445500   decpt: 8   sign: 1
 Output of dececvt: -1   decpt: 8   sign: 1
 Output of dececvt: -123   decpt: 8   sign: 1

Input string[4]: 12345.67
 Output of dececvt: +1235   decpt: 5   sign: 0
 Output of dececvt: +1234567000   decpt: 5   sign: 0
 Output of dececvt: +1   decpt: 5   sign: 0
 Output of dececvt: +123   decpt: 5   sign: 0

Input string[5]: .001234
 Output of dececvt: +1234   decpt: -2   sign: 0
 Output of dececvt: +1234000000   decpt: -2   sign: 0
 Output of dececvt: +1   decpt: -2   sign: 0
 Output of dececvt: +123   decpt: -2   sign: 0

DECECVT Sample Program over.
```

Example of decfcvt():

The file **decfcvt.ec** in the **demo** directory contains the following sample program.

```
/*
    * 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.
*/

#include <stdio.h>

EXEC SQL include decimal;

char *strings[] =
    {
    "210203.204",
    "4894",
    "443.334899312",
    "-12344455",
    0
    };


main()
{
    mint x;
    dec_t num;
    mint i = 0, f, sign;
    char *dp, *decfcvt();


    printf("DECFCVT Sample ESQL Program running.\n\n");

    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("Input string[%d]: %s\n", i, strings[i]);
          printf("  Output of decfcvt: %c%*.*s.%s  decpt: %d  sign: %d\n\n",
            (sign ? '-' : '+'), f, f, dp, dp+f, f, sign);
          ++i;                                             /* next string */
          }

      printf("\nDECFCVT Sample Program over.\n\n");
    }
```

Output of decfcvt():

```
DECFCVT Sample ESQL Program running.

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

DECFCVT Sample Program over.
```

The decmul() function

The **decmul()** function multiplies two **decimal** type values.

Syntax:

```
mint decmul(n1, n2, product)
    dec_t *n1;
    dec_t *n2;
    dec_t *product;
```

n1   A pointer to the **decimal** structure of the first operand.

n2   A pointer to the **decimal** structure of the second operand.

product   A pointer to the **decimal** structure that contains the product of *n1* times *n2*.

Usage:

The *product* 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.

Examples:

The file **decmul.ec** file in the **demo** directory contains the following sample program.

```
/*
   * decmul.ec *

   This program multiplies two DECIMAL numbers and displays the result.
*/

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "80.2";
char string2[] = "6.0";
char result[41];

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

    printf("DECMUL Sample ESQL Program running.\n\n");

    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 = dectoasc(&mpx, result, sizeof(result), -1))
      {
      printf("Error %d in converting mpx to display string\n", x);
      exit(1);
      }
    result[40] = '\0';
    printf("\t%s * %s = %s\n", string1, string2, result);

    printf("\nDECMUL Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
DECMUL Sample ESQL Program running.

   80.2 * 6.0 = 481.2

DECMUL Sample Program over.
```

The decround() function

The **decround()** function rounds a **decimal** type number to fractional digits.

Syntax:

```
void decround(d, s)
   dec_t *d;
   mint s;
```

d	A pointer to a **decimal** structure whose value the **decround()** function rounds.
s	The number of fractional digits to which **decround()** rounds *d*. Use a positive number for the *s* argument.

Usage:

The rounding factor is $5 \times 10^{-s-1}$. To round a value, the **decround()** function adds the rounding factor to a positive number or subtracts this factor from a negative number. It then truncates to *s* digits, as the following table shows.

Value Before Round	Value of *s*	Rounded Value
1.4	0	1.0
1.5	0	2.0
1.684	2	1.68
1.685	2	1.69
1.685	1	1.7
1.685	0	2.0

Return codes:

The file **decround.ec** in the **demo** directory contains the following sample program.

```
/*
   * decround.ec *

   The following program rounds a DECIMAL type number six times and displays
   the result of each operation.
*/

#include <stdio.h>

EXEC SQL include decimal;

char string[] = "-12345.038572";
char result[41];

main()
{
   mint x;
   mint i = 6;            /* number of decimal places to start with */
   dec_t num1;

   printf("DECROUND Sample ESQL Program running.\n\n");

   printf("String = %s\n", string);
   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;
           }
        result[40] = '\0';
        printf("  Rounded to %d Fractional Digits: %s\n", i--, result);
        }
     printf("\nDECROUND Sample Program over.\n\n");
}
```

Output:

```
DECROUND Sample ESQL Program running.

String = -12345.038572
   Rounded to 6 Fractional Digits: -12345.038572
   Rounded to 5 Fractional Digits: -12345.03857
   Rounded to 4 Fractional Digits: -12345.0386
   Rounded to 3 Fractional Digits: -12345.039
   Rounded to 2 Fractional Digits: -12345.04
   Rounded to 1 Fractional Digits: -12345.

DECROUND Sample Program over.
```

The decsub() function

The **decsub()** function subtracts two **decimal** type values.

Syntax:

```
mint decsub(n1, n2, difference)
   dec_t *n1;
   dec_t *n2;
   dec_t *difference;
```

n1   A pointer to the **decimal** structure of the first operand.

n2   A pointer to the **decimal** structure of the second operand.

difference A pointer to the **decimal** structure that contains the difference of *n1* minus *n2*.

Usage:

The *difference* 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.

Examples:

The file **decsub.ec** in the **demo** directory contains the following sample program.

```
/*
 * decsub.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "1000.038782";
char string2[] = "480";
char result[41];

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

    printf("DECSUB Sample ESQL Program running.\n\n");

    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 = dectoasc(&diff, result, sizeof(result), -1))
      {
      printf("Error %d in converting result to string\n", x);
      exit(1);
      }
    result[40] = '\0';
    printf("\t%s - %s = %s\n", string1, string2, result);

    printf("\nDECSUB Sample Program over.\n\n");
    exit(0);
}
```

Output:
```
DECSUB Sample ESQL Program running.

   1000.038782 - 480 = 520.038782

DECSUB Sample Program over.
```

The dectoasc() function

The **dectoasc()** function converts a **decimal** type number to a C **char** type value.

Syntax:

```
mint dectoasc(dec_val, strng_val, len, right)
   dec_t *dec_val;
   char  *strng_val;
   mint   len;
   mint   right;
```

dec_val   A pointer to the **decimal** structure whose value **dectoasc()** converts
              to a text string.

strng_val   A pointer to the first byte of the character buffer where the
              dectoasc() function places the text string.

len   The size of strng_val, in bytes, minus 1 for the null terminator.

right   An integer that indicates the number of decimal places to the right
              of the decimal point.

Usage:

If *right* = -1, the decimal value of *dec_val* determines the number of decimal places.

If the **decimal** 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,
dectoasc() fills the string with asterisks. If the number is shorter than the string,
dectoasc() left-justifies the number and pads it on the right with blanks.

Because the character string that **dectoasc()** returns is not null terminated, your
program must add a null character to the string before you print it.

Return codes:

0   The conversion was successful.

-1   The conversion failed.

Examples:

The file **dectoasc.ec** in the **demo** directory contains the following sample program.

```
/*
   * dectoasc.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

#define  END  sizeof(result)

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

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

    printf("DECTOASC Sample ESQL Program running.\n\n");

    printf("String Decimal Value 1 = %s\n", string1);
```

```
                    if (x = deccvasc(string1, strlen(string1), &num1))
                      {
                      printf("Error %d in converting string1 to DECIMAL\n", x);
                      exit(1);
                      }
                    printf("String Decimal Value 2 = %s\n", string2);
                    if (x = deccvasc(string2, strlen(string2), &num2))
                      {
                      printf("Error %d in converting string2 to DECIMAL\n", x);
                      exit(1);
                      }

                    printf("\nConverting Decimal back to ASCII\n");
                    printf("  Executing: dectoasc(&num1, result, 5, -1)\n");
                    if (x = dectoasc(&num1, result, 5, -1))
                      printf("\tError %d in converting DECIMAL1 to string\n", x);
                    else
                      {
                      result[5] = '\0';                         /* null terminate */
                      printf("\tResult ='%s'\n", result);
                      }

                    printf("Executing: dectoasc(&num1, result, 10, -1)\n");
                    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 = '%s'\n", result);
                      }

                    printf("Executing: dectoasc(&num2, result, END, 3)\n");
                    if (x = dectoasc(&num2, result, END, 3))
                      printf("\tError %d in converting DECIMAL2 to string\n", x);
                    else
                      {
                      result[END-1] = '\0';                     /* null terminate */
                      printf("\tResult = '%s'\n", result);
                      }

                    printf("\nDECTOASC Sample Program over.\n\n")
                  }
```

Output:

```
DECTOASC Sample ESQL Program running.

String Decimal Value 1 = -12345.038782
String Decimal Value 2 = 480

Converting Decimal back to ASCII
  Executing: dectoasc(&num1, result, 5, -1)
   Error -1 in converting decimal1 to string
  Executing: dectoasc(&num1, result, 10, -1)
   Result = '-12345.039'
  Executing: dectoasc(&num2, result, END, 3)
   Result = '480.000                           '

DECTOASC Sample Program over.
```

The dectodbl() function

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

Syntax:

```
mint dectodbl(dec_val, dbl_val)
   dec_t *dec_val;
   double *dbl_val;
```

dec_val   A pointer to a **decimal** structure whose value **dectodbl()** converts to a **double** type value.

dbl_val   A pointer to a **double** type where **dectodbl()** places the result of the conversion.

Usage:

The floating-point format of the host computer can result in loss of precision in the conversion of a **decimal** type number to a **double type number**.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **dectodbl.ec** file in the **demo** directory contains the following sample program.

```
/*
   * dectodbl.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

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

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

    printf("DECTODBL Sample ESQL Program running.\n\n");

    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("String 1 = %s\n", string1);
    printf("Double value = %.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("String 2 = %s\n", string2);
    printf("Double value = %f\n", d);

    printf("\nDECTODBL Sample Program over.\n\n");
    exit(0);
}
```

Output:
```
DECTODBL Sample ESQL Program running.

String 1 = 2949.3829398204382
Double value = 2949.382939820438423

String 2 = 3238299493
Double value = 3238299493.000000

DECTODBL Sample Program over.
```

The dectoint() function

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

Syntax:
```
mint dectoint(dec_val, int_val)
    dec_t *dec_val;
    mint *int_val;
```

dec_val   A pointer to a **decimal** structure whose value **dectoint()** converts to a **mint** type value.

int_val   A pointer to a **mint** value where **dectoint()** places the result of the conversion.

Usage:

The **dectoint()** library function converts a **decimal** value to a C integer. The size of a C integer depends on the hardware and operating system of the computer you are using. Therefore, the **dectoint()** function equates an integer value with the SQL SMALLINT data type. The valid range of a SMALLINT is between 32767 and -32767. To convert larger **decimal** values to larger integers, use the **dectolong()** library function.

Return codes:

0   The conversion was successful.

<0	The conversion failed.
-1200	The magnitude of the **decimal** type number is greater than 32767.

Examples:

The file **dectoint.ec** in the **demo** directory contains the following sample program.

```
/*
   * dectoint.ec *

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

#include <stdio.h>

EXEC SQL include decimal;

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


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

    printf("DECTOINT Sample ESQL Program running.\n\n)";

    printf("String 1 = %s\n", string1);
    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);
    else
        printf("  Result = %d\n", n);

    printf("\nString 2 = %s\n", string2);
    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);
    else
        printf("  Result = %d\n", n);

    printf("\nDECTOINT Sample Program over.\n\n");
    exit(0);
}
```

Output:
```
DECTOINT Sample ESQL Program running.

String 1 = 32767
  Result = 32767
```

```
String 2 = 32768
  Error -1200 in converting decimal to int

DECTOINT Sample Program over.
```

The dectolong() function

The **dectolong()** function converts a **decimal** type number into an **int4** type number.

Syntax:

```
mint dectolong(dec_val, lng_val)
   dec_t *dec_val;
   int4  *lng_val;
```

dec_val   A pointer to a **decimal** structure whose value **dectolong()** converts to an **int4** integer.

lng_val   A pointer to an **int4** integer where **dectolong()** places the result of the conversion.

Return codes:

0   The conversion was successful.

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

Examples:

The file **dectolong.ec** in the **demo** directory contains the following sample program.

```
/*
   * 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>

EXEC SQL include decimal;

char string1[] = "2147483647";
char string2[] = "2147483648";


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

    printf("DECTOLONG Sample ESQL Program running.\n\n");

    printf("String 1 = %s\n", string1);
    if (x = deccvasc(string1, strlen(string1), &num))
       {
       printf("  Error %d in converting string1 to DECIMAL\n", x);
       exit(1);
       }
    if (x = dectolong(&num, &n))
```

```
          printf("  Error %d in converting DECIMAL1 to long\n", x);
      else
          printf("  Result = %ld\n", n);

      printf("\nString 2 = %s\n", string2);
      if (x = deccvasc(string2, strlen(string2), &num))
        {
        printf("  Error %d in converting string2 to DECIMAL\n", x);
        exit(1);
        }
      if (x = dectolong(&num, &n))
        printf("  Error %d in converting DECIMAL2 to long\n", x);
      else
          printf("  Result = %ld\n", n);

      printf("\nDECTOLONG Sample Program over.\n\n");
      exit(0);
}
```

Output:

```
DECTOLONG Sample ESQL Program running.

String 1 = 2147483647
  Result = 2147483647

String 2 = 2147483648
  Error -1200 in converting DECIMAL2 to long

DECTOLONG Sample Program over.
```

The dectrunc() function

The **dectrunc()** function truncates a rounded **decimal** type number to fractional digits.

Syntax:

```
void dectrunc(d, s)
   dec_t *d;
   mint s;
```

d   A pointer to a **decimal** structure for a rounded number whose value **dectrunc()** truncates.

s   The number of fractional digits to which **dectrunc()** truncates the number. Use a positive number or zero for this argument.

Usage:

The following table shows the sample output from **dectrunc()** with various inputs.

Value before truncation	Value of *s*	Truncated value
1.4	0	1.0
1.5	0	1.0
1.684	2	1.68
1.685	2	1.68
1.685	1	1.6
1.685	0	1.0

Examples:

The file **dectrunc.ec** in the **demo** directory contains the following sample program.

```
/*
   * dectrunc.ec *

   The following program truncates a DECIMAL number six times and displays
   each result.
*/

#include <stdio.h>

EXEC SQL include decimal;

char string[] = "-12345.038572";
char result[41];

main()
{
    mint x;
    mint i = 6;              /* number of decimal places to start with */
    dec_t num1;

    printf("DECTRUNC Sample ESQL Program running.\n\n");

    printf("String = %s\n", string);
    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;
          }
      result[40] = '\0';
      printf("  Truncated to %d Fractional Digits: %s\n", i--, result);
      }

    printf("\nDECTRUNC Sample Program over.\n\n");
}
```

Output:

```
DECTRUNC Sample ESQL Program running.

String = -12345.038572
  Truncated to 6 Fractional Digits: -12345.038572
  Truncated to 5 Fractional Digits: -12345.03857
  Truncated to 4 Fractional Digits: -12345.0385
  Truncated to 3 Fractional Digits: -12345.038
  Truncated to 2 Fractional Digits: -12345.03
  Truncated to 1 Fractional Digits: -12345.0

DECTRUNC Sample Program over.
```

The dtaddinv() function

The **dtaddinv()** function adds an **interval** value to a **datetime** value. The result is a **datetime** value.

Syntax:

```
mint dtaddinv(dt, inv, res)
    dtime_t *dt;
    intrvl_t *inv;
    dtime_t *res;
```

dt	A pointer to the initialized **datetime** host variable.
inv	A pointer to the initialized **interval** host variable.
res	A pointer to the **datetime** host variable that contains the result.

Usage:

The **dtaddinv()** function adds the **interval** value in *inv* to the **datetime** value in *dt* and stores the **datetime** value in *res*. This result inherits the qualifier of *dt*.

The **interval** value must be in either the **year to month** or **day to fraction(5)** ranges.

The **datetime** value must include all the fields present in the **interval** value.

If you do not initialize the variables *dt* and *inv*, the function might return an unpredictable result.

Return codes:

0	The addition was successful.
<0	Error in addition.

Examples:

The **demo** directory contains this sample program in the **dtaddinv.ec** file.

```
/*
   * dtaddinv.ec *

   The following program adds an INTERVAL value to a DATETIME value and
   displays the result.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    char out_str[16];

    EXEC SQL BEGIN DECLARE SECTION;
      datetime year to minute dt_var, result;
      interval day to minute intvl;
    EXEC SQL END DECLARE SECTION;

    printf("DTADDINV Sample ESQL Program running.\n\n");
```

```
            printf("datetime year to minute value=2006-11-28 11:40\n");
            dtcvasc("2006-11-28 11:40", &dt_var);
            printf("interval day to minute value = 50 10:20\n");
            incvasc("50 10:20", &intvl);

            dtaddinv(&dt_var, &intvl, &result);

            /* Convert to ASCII for displaying */
            dttoasc(&result, out_str);
            printf("----------------------------------------------------------\n");
            printf("                                          Sum=%s\n", out_str);

            printf("\nDTADDINV Sample Program over.\n\n");
}
```

Output:

```
DTADDINV Sample ESQL Program running.

datetime year to minute value=2006-11-28 11:40
interval day to minute value =      50 10:20
----------------------------------------------
                              Sum=2007-01-17 22:00

DTADDINV Sample Program over.
```

The dtcurrent() function

The **dtcurrent()** function assigns the current date and time to a **datetime** variable.

Syntax:

```
void dtcurrent(d)
    dtime_t *d;
```

d   A pointer to the initialized **datetime** host variable.

Usage:

When the variable qualifier is set to zero (or any invalid qualifier), the **dtcurrent()** function initializes it with the **year to fraction(3)** qualifier.

When the variable contains a valid qualifier, the **dtcurrent()** function extends the current date and time to agree with the qualifier.

Example calls:

The following statements set the host variable **timewarp** to the current date:

```
EXEC SQL BEGIN DECLARE SECTION;
    datetime year to day timewarp;
EXEC SQL END DECLARE SECTION;

dtcurrent(&timewarp);
```

The following statements set the variable **now** to the current time, to the nearest millisecond:

```
now.dt_qual = TU_DTENCODE(TU_HOUR,TU_F3);
dtcurrent(&now);
```

Examples:

The **demo** directory contains this sample program in the **dtcurrent.ec** file.

```
/*
   * dtcurrent.ec *

   The following program obtains the current date from the system, converts
   it to ASCII and prints it.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    mint x;
    char out_str[20];

    EXEC SQL BEGIN DECLARE SECTION;
      datetime year to hour dt1;
    EXEC SQL END DECLARE SECTION;

    printf("DTCURRENT Sample ESQL Program running.\n\n");

    /* Get today's date */
    dtcurrent(&dt1);

    /* Convert to ASCII for displaying */
    dttoasc(&dt1, out_str);
    printf("\tToday's datetime (year to minute) value is %s\n", out_str);

    printf("\nDTCURRENT Sample Program over.\n\n");
}
```

Output:

```
DTCURRENT Sample ESQL Program running.

    Today's datetime (year to minute) value is 2007-09-16 14:49

DTCURRENT Sample Program over.
```

The dtcvasc() function

The **dtcvasc()** function converts a string that conforms to ANSI SQL standard for a DATETIME value to a **datetime** value. For information about the ANSI SQL DATETIME standard, see "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

Syntax:

```
mint dtcvasc(inbuf, dtvalue)
   char *inbuf;
   dtime_t *dtvalue;
```

inbuf	A pointer to the buffer that contains an ANSI-standard DATETIME string.
dtvalue	A pointer to an initialized **datetime** variable.

Usage:

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want this variable to have.

The character string in *inbuf* must have values that conform to the **year to second** qualifier in the ANSI SQL format. The *inbuf* string can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can only contain characters that are digits and delimiters that conform to the ANSI SQL standard for DATETIME values.

If you specify a year value as one or two digits, the **dtcvasc()** function assumes that the year is in the present century. You can set the **DBCENTURY** environment variable to determine which century **dtcvasc()** uses when you omit a century from the date.

If the character string is an empty string, the **dtcvasc()** function sets to null the value to which *dtvalue* points. If the character string is acceptable, the function sets the value in the **datetime** variable and returns zero. Otherwise, the function leaves the variable unchanged and returns a negative error code.

Return codes:

0	Conversion was successful.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of **datetime** or **interval**.
-1262	Non-numeric character in **datetime** or **interval**.
-1263	A field in a **datetime** or **interval** value is out of range or incorrect.
-1264	Extra characters exist at the end of a **datetime** or **interval**.
-1265	Overflow occurred on a **datetime** or **interval** operation.
-1266	A **datetime** or **interval** value is incompatible with the operation.
-1267	The result of a **datetime** computation is out of range.
-1268	A parameter contains an invalid **datetime** qualifier.

Examples:

The **demo** directory contains this sample program in the **dtcvasc.ec** file.

```
/*
 * dtcvasc.ec *

 The following program converts ASCII datetime strings in ANSI SQL format
 into datetime (dtime_t) structure.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    mint x;

    EXEC SQL BEGIN DECLARE SECTION;
      datetime year to second dt1;
    EXEC SQL END DECLARE SECTION;

    printf("DTCVASC Sample ESQL Program running.\n\n");

    printf("Datetime string #1 = 2007-02-11 3:10:35\n");
```

```
        if (x = dtcvasc("2007-02-11 3:10:35", &dt1))
          printf("Result =  failed with conversion error: %d\n", x);
        else
          printf("Result = successful conversion\n");

        /*
         * Note that the following literal string has a 26 in the hours place
         */
        printf("\nDatetime string #2 = 2007-02-04 26:10:35\n");
        if (x = dtcvasc("2007-02-04 26:10:35", &dt1))
          printf("Result =  failed with conversion error: %d\n", x);
        else
          printf("Result = successful conversion\n");

        printf("\nDTCVASC Sample Program over.\n\n");
}
```

Output:

```
DTCVASC Sample ESQL Program running.

Datetime string #1 = 2007-02-11 3:10:35
Result =  successful conversion

Datetime string #2 = 2007-02-04 26:10:35
Result = failed with conversion error:-1263

DTCVASC Sample Program over.
```

The dtcvfmtasc() function

The **dtcvfmtasc()** function uses a formatting mask to convert a character string to a **datetime** value.

Syntax:

```
mint dtcvfmtasc(inbuf, fmtstring, dtvalue)
    char *inbuf;
    char *fmtstring;
    dtime_t *dtvalue;
```

inbuf	A pointer to the buffer that contains the string to convert.
fmtstring	A pointer to the buffer that contains the formatting mask to use for the *inbuf* string. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*).
dtvalue	A pointer to the initialized **datetime** variable.

Usage:

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want this variable to have. The **datetime** variable does not need to specify the same qualifier that the formatting mask implies. When the **datetime** qualifier is different from the implied formatting-mask qualifier, **dtcvfmtasc()** extends the **datetime** value (as if it had called the **dtextend** function).

All qualifier fields in the character string in *inbuf* must be contiguous. In other words, if the qualifier is **hour to second**, you must specify all values for **hour**, **minute**, and **second** somewhere in the string, or the **dtcvfmtasc()** function returns an error.

The *inbuf* character string can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can contain only digits and delimiters that are appropriate for the qualifier fields that the formatting mask implies. For more information about acceptable digits and delimiters for a DATETIME value, see the "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

The **dtcvfmtasc()** function returns an error if the formatting mask, *fmtstring*, is an empty string. If *fmtstring* is a null pointer, the **dtcvfmtasc()** function must determine the format to use when it reads the character string in *inbuf*. When you use the default locale, the function uses the following precedence:

1. The format that the **DBTIME** environment variable specifies (if **DBTIME** is set). For more information about **DBTIME**, see the *IBM Informix Guide to SQL: Reference*.
2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set). For more information about **GL_DATETIME**, see the *IBM Informix GLS User's Guide*.
3. The default date format conforms to the standard ANSI SQL format:

   ```
   %iY-%m-%d %H:%M:%S
   ```

The ANSI SQL format specifies a qualifier of **year to second** for the output. You can express the year as four digits (2007) or as two digits (07). When you use a two-digit year (**%y**) in a formatting mask, the **dtcvfmtasc()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **dtcvfmtasc()** assumes the present century for two-digit years. For information about how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **dtcvfmtasc()** uses the default DATETIME format that the locale defines. For more information, see the *IBM Informix GLS User's Guide*.

When the character string and the formatting mask are acceptable, the **dtcvfmtasc()** function sets the **datetime** variable in *dtvalue* and returns zero. Otherwise, it returns an error code and the **datetime** variable contains an unpredictable value.

Return codes:

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

Examples:

The **demo** directory contains this sample program in the file **dtcvfmtasc.ec**. The code initializes the variable **birthday** to a fictitious birthday.

```
/* *dtcvfmtasc.ec*
 The following program illustrates the conversion of several ascii strings
 into datetime values.
*/
```

```
#include <stdio.h>

EXEC SQL include datetime;

main()
{
   char out_str[17],  out_str2[17],  out_str3[17]; mint x;

   EXEC SQL BEGIN DECLARE SECTION;
       datetime month to minute birthday;
       datetime year to minute birthday2;
       datetime year to minute birthday3;
   EXEC SQL END DECLARE SECTION;

   printf("DTCVFMTASC Sample ESQL Program running.\n\n");

   /* Initialize birthday to "09-06 13:30" */
   printf("Birthday #1 = September 6 at 01:30 pm\n");
   x = dtcvfmtasc("September 6 at 01:30 pm", "%B %d at %I:%M %p",
       &birthday);

   /*Convert the internal format to ascii in ANSI format, for displaying. */
   x = dttoasc(&birthday, out_str);
   printf("Datetime (month to minute) value = %s\n\n", out_str);
 /* Initialize birthday2 to "07-14-88 09:15" */
   printf("Birthday #2 = July 14, 1988. Time: 9:15 am\n");
   x = dtcvfmtasc("July 14, 1988. Time: 9:15am",
       "%B %d, %Y. Time: %I:38p",  &birthday2);

   /*Convert the internal format to ascii in ANSI format, for displaying. */
x = dttoasc(&birthday2, out_str2);
   printf("Datetime (year to minute) value = %s\n\n", out_str2);
 /* Initialize birthday3 to "07-14-XX 09:15" where XX is current year.
   * Note that birthday3 is year to minute but this initialization only
   * provides month to minute. dtcvfmtasc provides current information
   * for the missing year.
 */
   printf("Birthday #3 = July 14. Time: 9:15 am\n");
   x = dtcvfmtasc("July 14. Time: 9:15am", "%B %d. Time: %I:%M %p",
       &birthday3);

 /* Convert the internal format to ascii in ANSI format, for displaying. */
   x = dttoasc(&birthday3, out_str3);
   printf("Datetime (year to minute) value with current year = %s\n",
       out_str3);

    printf("\nDTCVFMTASC Sample Program over.\n\n");

}
```

Output:

```
DTCVFMTASC Sample ESQL Program running.

Birthday #1 = September 6 at 01:30 pm
Datetime (month to minute) value = 09-06 13:30

Birthday #2 = July 14, 1988 Time: 9:15 am
Datetime (year to minute) value =  2007-07-14 09:15

Birthday #3 = July 14. Time: 9:15 am
Datetime  (year to minute) value with current year = 2007-07-14 09:15

DTCVFMTASC Sample Program over.
```

The dtextend() function

The **dtextend()** function extends a **datetime** value to a different qualifier. Extending is the operation of adding or dropping fields of a DATETIME value to make it match a given qualifier.

Syntax:

```
mint dtextend(in_dt, out_dt)
   dtime_t *in_dt, *out_dt;
```

in_dt   A pointer to the **datetime** variable to extend.

out_dt   A pointer to the **datetime** variable with a valid qualifier to use for the extension.

Usage:

The **dtextend()** function copies the qualifier-field digits of the *in_dt* **datetime** variable to the *out_dt* **datetime** variable. The qualifier of the *out_dt* variable controls the copy.

The function discards any fields in *in_dt* that the *out_dt* variable does not include. The function fills in any fields in *out_dt* that are not present in *in_dt*, as follows:

- It fills in fields to the left of the most-significant field in *in_dt* from the current time and date.
- It fills in fields to the right of the least-significant field in *in_dt* with zeros.

In the following example, a variable **fiscal_start** is set up with the first day of a fiscal year that begins on June 1. The **dtextend()** function generates the current year.

```
EXEC SQL BEGIN DECLARE SECTION;
    datetime work, fiscal_start;
EXEC SQL END DECLARE SECTION;

work.dt_qual = TU_DTENCODE(TU_MONTH,TU_DAY);
dtcvasc("06-01",&work);
fiscal_start.dt_qual = TU_DTENCODE(TU_YEAR,TU_DAY);
dtextend(&work,&fiscal_start);
```

Return codes:

0   The operation was successful.

-1268   A parameter contains an invalid **datetime** qualifier.

Examples:

The **demo** directory contains this sample program in the file **dtextend.ec**.

```
/*
   * 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>

EXEC SQL include datetime;
```

```
main()
{
    mint x;
    char year_str[20];

    EXEC SQL BEGIN DECLARE SECTION;
      datetime month to day month_dt;
      datetime year to minute year_min;
    EXEC SQL END DECLARE SECTION;

    printf("DTEXTEND Sample ESQL Program running.\n\n");

    /* Assign value to month_dt and extend */
    printf("Datetime (month to day) value = 12-07\n");
    if(x = dtcvasc("12-07", &month_dt))
       printf("Result = Error %d in dtcvasc()\n", x);
else
      {
       if (x = dtextend(&month_dt, &year_min))
         printf("Result = Error %d in dtextend()\n", x);
       else
          {
           dttoasc(&year_min, year_str);
           printf("Datetime (year to minute) extended value =%s\n",
             year_str);
          }
      }

    printf("\nDTEXTEND Sample Program over.\n\n");
}
```

Output:

```
DTEXTEND Sample ESQL Program running.

Datetime (month to day) value = 12-07
Datetime (year to minute) extended value = 2006-12-07 00:00

DTEXTEND Sample Program over.
```

The dtsub() function

The **dtsub()** function subtracts one **datetime** value from another. The result is an **interval** value.

Syntax:

```
mint dtsub(d1, d2, inv)
   dtime_t *d1, *d2;
   intrvl_t *inv;
```

d1   A pointer to an initialized **datetime** host variable.

d2   A pointer to an initialized **datetime** host variable.

inv   A pointer to the **interval** host variable that contains the result.

Usage:

The **dtsub()** function subtracts the **datetime** value *d2* from *d1* and stores the **interval** result in *inv*. The result can be either a positive or a negative value. If necessary, the function extends *d2* to match the qualifier for *d1*, before the subtraction.

Initialize the qualifier for *inv* with a value in either the **year to month** or **day to fraction(5)** classes. When *d1* contains fields in the **day to fraction** class, the **interval** qualifier must also be in the **day to fraction** class.

Return codes:

0   The subtraction was successful.

<0   An error occurred while performing the subtraction.

Examples:

The **demo** directory contains this sample program in the file **dtsub.ec**. The program performs **datetime** subtraction that returns equivalent **interval** results in the range of **year to month** and **month to month** and attempts to return an **interval** result in the range **day to hour**.

```
/*
   * dtsub.ec *

   The following program subtracts one DATETIME value from another and
   displays  the resulting INTERVAL value or an error message.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    mint x;
    char out_str[16];

    EXEC SQL BEGIN DECLARE SECTION;
        datetime year to month dt_var1, dt_var2;
        interval year to month i_ytm;
        interval month to month i_mtm;
        interval day to hour i_dth;
    EXEC SQL END DECLARE SECTION;


    printf("DTSUB Sample ESQL Program running.\n\n");

    printf("Datetime (year to month) value #1 = 2007-10\n");
    dtcvasc("2007-10", &dt_var1);
    printf("Datetime (year to month) value #2 = 2001-08\n");
    dtcvasc("2001-08", &dt_var2);

    printf("----------------------------------------------------------\n");

   /* Determine year-to-month difference */
    printf("Difference (year to month)          = ");
    if(x = dtsub(&dt_var1, &dt_var2, &i_ytm))
      printf("Error from dtsub(): %d\n", x);
    else
      {
      /* Convert to ASCII for displaying */
      intoasc(&i_ytm, out_str);
      printf("%s\n", out_str);
      }

   /* Determine month-to-month difference */
    printf("Difference (month to month)        = ");
```

```
    if(x = dtsub(&dt_var1, &dt_var2, &i_mtm))
      printf("Error from dtsub(): %d\n", x);
    else
       {
       /* Convert to ASCII for displaying */
       intoasc(&i_mtm, out_str);
       printf("%s\n", out_str);
       }

   /* Determine day-to-hour difference: Error - Can't convert
    * year-to-month to day-to-hour
    */
   printf("Difference (day to hour)              = ");
   if(x = dtsub(&dt_var1, &dt_var2, &i_dth))
     printf("Error from dtsub(): %d\n", x);
   else
      {
       /* Convert to ASCII for displaying */
      intoasc(&i_dth, out_str);
      printf("%s\n", out_str);
      }

   printf("\nDTSUB Sample Program over.\n\n");
}
```

Output:

```
DTSUB Sample ESQL Program running.

Datetime (year to month) value #1 = 2007-10
Datetime (year to month) value #2 = 2001-08
-------------------------------------------
Difference (year to month)    = 0006-02
Difference (month to month)   = 86
Difference (day to hour)      = Error from dtsub(): -1266

DTSUB Sample Program over.
```

The dtsubinv() function

The **dtsubinv()** function subtracts an **interval** value from a **datetime** value. The result is a **datetime** value.

Syntax:

```
mint dtsubinv(dt, inv, res)
   dtime_t *dt;
   intrvl_t *inv;
   dtime_t *res;
```

dt   A pointer to an initialized **datetime** host variable.

inv   A pointer to an initialized **interval** host variable.

res   A pointer to the **datetime** host variable that contains the result.

Usage:

The **dtsubinv()** function subtracts the **interval** value in *inv* from the **datetime** value in *dt* and stores the **datetime** value in *res*. This result inherits the qualifier of *dt*.

The **datetime** value must include all the fields present in the **interval** value. When you do not initialize the variables *dt* and *inv*, the function might return an unpredictable result.

Return codes:

0   The subtraction was successful.

<0   An error occurred while performing the subtraction.

Examples:

The **demo** directory contains this sample program in the file **dtsubinv.ec**.

```
/*
   * dtsubinv.ec *

   The following program subtracts an INTERVAL value from a DATETIME value and
   displays the result.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    char out_str[16];

    EXEC SQL BEGIN DECLARE SECTION;
      datetime year to minute dt_var, result;
      interval day to minute intvl;
    EXEC SQL END DECLARE SECTION;

    printf("DTSUBINV Sample ESQL Program running.\n\n");

    printf("Datetime (year to month) value = 2007-11-28\n");
    dtcvasc("2007-11-28 11:40", &dt_var);
    printf("Interval (day to minute) value   =     50 10:20\n");
    incvasc("50 10:20", &intvl);

    printf("-----------------------------------------------\n");
    dtsubinv(&dt_var, &intvl, &result);

    /* Convert to ASCII for displaying */
    dttoasc(&result, out_str);
    printf("Difference (year to hour)            = %s\n", out_str);

    printf("\nDTSUBINV Sample Program over.\n\n");
}
```

Output:

```
DTSUBINV Sample ESQL Program running.

Datetime (year to month) value = 2007-11-28
Interval (day to minute) value =    50 10:20
----------------------------------------------------
Difference (year to hour)      = 2007-10-09  01:20

DTSUBINV Sample Program over.
```

The dttoasc() function

The **dttoasc()** function converts the field values of a **datetime** variable to an ASCII string that conforms to ANSI SQL standards. For information about the ANSI SQL DATETIME standard, see "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

Syntax:

```
mint dttoasc(dtvalue, outbuf)
   dtime_t *dtvalue;
   char *outbuf;
```

dtvalue   A pointer to the initialized **datetime** variable to convert.

outbuf   A pointer to the buffer that receives the ANSI-standard DATETIME string for the value in *dtvalue*.

Usage:

The **dttoasc()** function converts the digits of the fields in the **datetime** variable to their character equivalents and copies them to the *outbuf* character string with delimiters (hyphen, space, colon, or period) between them. You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want the character string to have.

The character string does not include the qualifier or the parentheses that SQL statements use to delimit a DATETIME literal. The *outbuf* string conforms to ANSI SQL standards. It includes one character for each delimiter, plus the fields, which are of the following sizes.

Field	Field Size
Year	Four digits
Fraction of DATETIME	As specified by precision
All other fields	Two digits

A **datetime** value with the **year to fraction(5)** qualifier produces the maximum length of output. The string equivalent contains 19 digits, 6 delimiters, and the null terminator, for a total of 26 bytes:

```
YYYY-MM-DD HH:MM:SS.FFFFF
```

If you do not initialize the qualifier of the **datetime** variable, the **dttoasc()** function returns an unpredictable value, but this value does not exceed 26 bytes.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The **demo** directory contains this sample program in the file **dttoasc.ec**.

```
/*
   * dttoasc.ec *

   The following program illustrates the conversion of a datetime value
   into an ASCII string in ANSI SQL format
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
```

```
      char out_str[16];

      EXEC SQL BEGIN DECLARE SECTION;
        datetime year to hour dt1;
      EXEC SQL END DECLARE SECTION;

    printf("DTTOASC Sample ESQL Program running.\n\n");

      /* Initialize dt1 */
      dtcurrent(&dt1);

      /* Convert the internal format to ascii for displaying */
      dttoasc(&dt1, out_str);

      /* Print it out*/
      printf("\tToday's datetime (year to hour)value  is %s\n", out_str);

    printf("\nDTTOASC Sample Program over.\n\n");
}
```

Output:

```
DTTOASC Sample ESQL Program running.

    Today's datetime (year to hour) value is 2007-09-19 08

DTTOASC Sample Program over.
```

The dttofmtasc() function

The **dttofmtasc()** function uses a formatting mask to convert a **datetime** variable to
a character string.

Syntax:

```
mint dttofmtasc(dtvalue, outbuf, buflen, fmtstring)
    dtime_t *dtvalue;
    char *outbuf;
    mint buflen;
    char *fmtstring;
```

dtvalue	A pointer to the initialized **datetime** variable to convert.
outbuf	A pointer to the buffer that receives the string for the value in *dtvalue*.
buflen	The length of the *outbuf* buffer.
fmtstring	A pointer to the buffer that contains the formatting mask to use for the *outbuf* string. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*).

Usage:

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want
the character string to have. If you do not initialize the **datetime** variable, the
function returns an unpredictable value. The character string in *outbuf* does *not*
include the qualifier or the parentheses that SQL statements use to delimit a
DATETIME literal.

The formatting mask, *fmtstring*, does not need to imply the same qualifiers as the **datetime** variable. When the implied formatting-mask qualifier is different from the **datetime** qualifier, **dttofmtasc()** extends the **datetime** value (as if it called the **dtextend()** function).

If the formatting mask is an empty string, the function sets character string, *outbuf*, to an empty string. If *fmtstring* is a null pointer, the **dttofmtasc()** function must determine the format to use for the character string in *outbuf.* When you use the default locale, the function uses the following precedence:

1. The format that the **DBTIME** environment variable specifies (if **DBTIME** is set). For more information about **DBTIME**, see the *IBM Informix Guide to SQL: Reference.*

2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set). For more information about **GL_DATETIME**, see the *IBM Informix GLS User's Guide.*

3. The default date format that conforms to the standard ANSI SQL format:

   %iY-%m-%d %H:%M:%S

When you use a two-digit year (**%y**) in a formatting mask, the **dttofmtasc()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **dttofmtasc()** assumes the present century for two-digit years. For information about how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference.*

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **dttofmtasc()** uses the default DATETIME format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide.*

Return codes:

0   The conversion was successful.

<0   The conversion failed. Check the text of the error message.

Examples:

The **demo** directory contains this sample program in the file **dttofmtasc.ec**.

```
/* *dttofmtasc.ec*
   The following program illustrates the conversion of a datetime
   value into strings of different formats.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
   char out_str1[25];
   char out_str2[25];
   char out_str3[30];
   mint x;

   EXEC SQL BEGIN DECLARE SECTION;
        datetime month to minute birthday;
   EXEC SQL END DECLARE SECTION;
```

```
               printf("DTTOFMTASC Sample ESQL Program running.\n\n");

               /* Initialize birthday to "09-06 13:30" */
               printf("Birthday datetime (month to minute) value = ");
               printf("September 6 at 01:30 pm\n");
               x = dtcvfmtasc("September 6 at 01:30 pm","%B %d at %I:%M %p",
                   &birthday);

               /* Convert the internal format to ascii for 3 given display formats.
                * Note that the second format does not include the minutes field and
                * that the last format includes a year field even though birthday was
                * not initialized as year to minute.
                */

               x = dttofmtasc(&birthday, out_str1, sizeof(out_str1),
                   "%d %B at %H:%M");
               x = dttofmtasc(&birthday, out_str2, sizeof(out_str2),
                   "%d %B at %H");
               x = dttofmtasc(&birthday, out_str3, sizeof(out_str3),
                   "%d %B, %Y at%H:%M");   /* Print out the three forms of the same date */
               printf("\tFormatted value (%%d %%B at %%H:%%M) = %s\n", out_str1);
               printf("\tFormatted value (%%d %%B at %%H) = %s\n", out_str2);
               printf("\tFormatted value (%%d %%B, %%Y at %%H:%%M) = %s\n", out_str3);

               printf("\nDTTOFMTASC Sample Program over.\n\n");
         }
```

Output:

```
DTTOFMTASC Sample ESQL Program running.

Birthday datetime (month to minute) value = September 6 at 01:30 pm
   Formatted value (%d %B at %H:%M)  = 06 September at 13:30
   Formatted value (%d %B at %H)) = 06 September at 13
   Formatted value (%d %B, %Y at %H:%M)) = 06 September, 2007 at 13:30

DTTOFMTASC Sample Program over.
```

The GetConnect() function (Windows)

The **GetConnect()** function is available only in Windows environments and establishes a new explicit connection to a database server.

Important: Informix ESQL/C supports the **GetConnect()** connection library function for compatibility with Version 5.01 Informix ESQL/C for Windows applications. When you write new Informix ESQL/C applications for Windows environments, use the SQL CONNECT statement to establish an explicit connection.

Syntax:

```
void *GetConnect ( )
```

Usage:

The **GetConnect()** function call *by itself* is equivalent to the following SQL statement:

```
EXEC SQL connect to '@dbservername' with concurrent transaction;
```

In this example, *dbservername* is the name of a defined database server. All database servers that the client application specifies must be defined in *at least one* of the following places:

- The **INFORMIXSERVER** environment variable in the Registry contains the name of the *default* database server. The **Setnet32** utility sets the Registry values.
- The **InfxServer** field in the **InetLogin** structure can contain the name of the *default* database server or a *specified* database server. The client application sets the **InetLogin** fields.

For more information about the default and specified database server, see "Sources of connection information in a Windows environment" on page 12-5

For example, the following code fragment uses **GetConnect()** to establish an explicit connection to the **stores7** database on the **mainsrvr** database server:

```
void *cnctHndl;
.
.
.
strcpy(InetLogin.InfxServer, "mainsrvr");
.
.
.

cnctHndl = GetConnect();
EXEC SQL database stores7;
```

In the preceding example, if you had omitted the assignment to the **InetLogin.InfxServer** field, Informix ESQL/C would establish an explicit connection to the **stores7** database in the default database server (the database server that the **INFORMIXSERVER** environment variable in the Registry indicates).

After any call to **GetConnect()**, use the SQL DATABASE statement (or some other SQL statement that opens a database) to open the desired database. In the previous code fragment, the combination of the **GetConnect()** function and the DATABASE statement is equivalent to the following CONNECT statement:

```
EXEC SQL connect to 'stores7@mainsrvr' with concurrent transaction;
```

Important: Because the **GetConnect()** function maps to a CONNECT statement, it sets the SQLCODE and SQLSTATE status codes to indicate the success or failure of the connection request. This behavior differs from that of **GetConnect()** in Version 5.01 Informix ESQL/C for Windows, in which this function did not set the SQLCODE and SQLSTATE values.

The following table shows the differences between the use of the **GetConnect()** function and the SQL CONNECT statement.

Situation	GetConnect() library function	SQL CONNECT statement
Connection name	Internally generated and stored in the connection handle structure for the connection	Internally generated unless CONNECT includes the AS clause; therefore, to switch to other connections, specify the AS clause when you create the connection.
Opening a database	Only establishes an explicit connection to a database server; therefore, the application must use DATABASE (or some other valid SQL statement) to open the database.	Can establish an explicit connection to a database server *and* open a database when provided with names of both the database server and the database

Important: Because the **GetConnect()** function maps to a CONNECT statement with the WITH CONCURRENT TRANSACTION clause, it allows an explicit connection with open transactions to become dormant. Your Informix ESQL/C application does not need to ensure that the current transaction was committed or rolled back before it calls the **SetConnect()** function to switch to another explicit connection.

For each connection that you establish with **GetConnect()**, call **ReleaseConnect()** to close the connection and deallocate resources.

Return codes:

CnctHndl   The call to **GetConnect()** was successful, and the function has returned a connection handle for the new connection.

null pointer   The call to **GetConnect()** was unsuccessful.

The ifx_cl_card() function

The **ifx_cl_card()** function returns the cardinality of the specified collection type host variable.

Syntax:
```
mint ifx_cl_card(collp, isnull)
   ifx_collection_t *collp;
   mint *isnull;
```

collp   A pointer to the name of the **collection** host variable in the application.

isnull   Set to 1 if the collection is null, 0 otherwise

Usage:

The **ifx_cl_card()** function enables you to determine the number of elements in a collection, whether the collection is empty, and whether the collection is null.

Return codes:

0   The collection is empty.

>0   The number of elements in the collection.

<0   An error occurred.

Examples:

This sample program is in the **ifx_cl_card.ec** file in the **demo** directory.

```
/*
 * Check the cardinality of the collection variable when
 * the data is returned from the server
 */

main()
{
        exec sql begin declare section;
        client collection myset;
        exec sql end declare section;
    mint numelems = 0;
        mint isnull = 0;
```

```
            exec sql allocate collection ::myset;
            exec sql create database newdb;
            exec sql create table tab (col set(int not null));
            exec sql insert into tab values ("set{}");
            exec sql select * into :myset from tab;
            if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 0)
                    printf("collection is empty\n");
            else if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 1)
                    printf("collection is null\n");
            else if ((numelems = ifx_cl_card(myset, &isnull))> 0)
                    printf("number of elements is %d\n", numelems);
            else
                    printf("error occurred\n");

            exec sql update tab set col = 'set{1,2,3}';
            exec sql select * into :myset from tab;
            if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 0)
                    printf("collection is empty\n");

            else if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 1)
                    printf("collection is null\n");
            else if ((numelems = ifx_cl_card(myset, &isnull))> 0)
                    printf("number of elements is %d\n", numelems);
            else
                    printf("error occurred\n");

            exec sql update tab set col = NULL;
            exec sql select * into :myset from tab;
            if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 0)
                    printf("collection is empty\n");
            else if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 1)
                    printf("collection is null\n");
            else if ((numelems = ifx_cl_card(myset, &isnull))> 0)
                    printf("number of elements is %d\n", numelems);
            else
                    printf("error occurred\n");
    }
```

Output:

```
collection is empty
number of elements is 3
collection is null
```

The ifx_dececvt() and ifx_decfcvt() function

The **ifx_dececvt()** and **ifx_decfcvt()** functions are the thread-safe versions of the **dececvt()** and **decfcvt()** Informix ESQL/C library functions.

Syntax:

```
mint ifx_dececvt(np, ndigit, decpt, sign, decstr, decstrlen)
   register dec_t *np;
   register mint ndigit;
   mint *decpt;
   mint *sign;
   char *decstr;
   mint decstrlen;

mint ifx_decfcvt(np, ndigit, decpt, sign, decstr, decstrlen)
   register dec_t *np;
   register mint ndigit;
```

```
mint *decpt;
mint *sign;
char *decstr;
mint decstrlen;
```

np	A pointer to a **decimal** structure that contains the **decimal** value to be converted.
ndigit	The length of the ASCII string for **ifx_dececvt()**. It is the number of digits to the right of the decimal point for **ifx_decfcvt()**.
decpt	A pointer to an integer that is the position of the decimal point relative to the beginning of the string. A negative or zero value for *decpt means the position is located to the left of the returned digits.
sign	A pointer to the sign of the result. If the sign of the result is negative, *sign is nonzero; otherwise, it is zero.
decstr	The user-defined buffer where the function returns the converted decimal value.
decstrlen	The length, in bytes, of the decstr buffer that the user defines.

Usage:

The **ifx_dececvt()** function is the thread-safe version of the **dececvt()** function. The **ifx_decfcvt()** function is the thread-safe version of **decfcvt()** function. Each function returns a character string that cannot be overwritten when two threads simultaneously call the function. For information about how to use **dececvt()** and **decfcvt()**, see B-28.

Return codes:

0	The conversion was successful.
<0	The conversion was not successful.
-1273	Output buffer is null or too small to hold the result.

The ifx_defmtdate() function

The **ifx_defmtdate()** function uses a formatting mask to convert a character string to an internal DATE format.

Syntax:
```
mint ifx_defmtdate(jdate, fmtstring, instring, dbcentury)
   int4 *jdate;
   char *fmtstring;
   char *instring;
   char dbcentury;
```

jdate	A pointer to an **int4** integer value that receives the internal DATE value for the *inbuf* string.
fmtstring	A pointer to the buffer that contains the formatting mask to use for the *inbuf* string.
instring	A pointer to the buffer that contains the date string to convert.
dbcentury	Can be one of the following characters, which determines which century to apply to the year portion of the date:

	R	Present. The function uses the two high-order digits of the current year to expand the year value.

R   Present. The function uses the two high-order digits of the
        current year to expand the year value.

P   Past. The function uses the present and past centuries to
        expand the year value. It compares these two dates against
        the current date and uses the century that is prior to the
        current century. If both dates are prior to the current date,
        the function uses the century closest to the current date.

F   Future. The function uses the present and next centuries to
        expand the year value. It compares these against the
        current date and uses the century that is later than the
        current date. If both dates are later than the current date,
        the function uses the date closest to the current date.

C   Closest. The function uses the present, past, and next
        centuries to expand the year value. It chooses the century
        that is closest to the current date.

Usage:

The *fmtstring* argument points to the date-formatting mask, which contains formats
that describe how to interpret the date string. For more information about these
date formats, see "Formatting date strings" on page 6-1.

The *input* string and the *fmtstring* must be in the same sequential order in terms of
month, day, and year. They need not, however, contain the same literals or the
same representation for month, day, and year.

You can include the weekday format (ww), in *fmtstring*, but the database server
ignores that format. Nothing from the *inbuf* corresponds to the weekday format.

The following combinations of *fmtstring* and *input* are valid.

Formatting mask	Input
mmddyy	Dec. 25th, 2007
mmddyyyy	Dec. 25th, 2007
mmm. dd. yyyy	dec 25 2007
mmm. dd. yyyy	DEC-25-2007
mmm. dd. yyyy	122507
mmm. dd. yyyy	12/25/07
yy/mm/dd	07/12/25
yy/mm/dd	2007, December 25th
yy/mm/dd	In the year 2007, the month of December, it is the 25th day
dd-mm-yy	This 25th day of December, 2007

If the value stored in *inbuf* is a four-digit year, the **ifx_defmtdate()** function uses
that value. If the value stored in *inbuf* is a two-digit year, the **ifx_defmtdate()**
function uses the value of the *dbcentury* argument to determine which century to
use. If you do not set the *dbcentury* argument, **ifx_defmtdate()** uses the
DBCENTURY environment variable to determine which century to use. If you do

not set **DBCENTURY**, **ifx_strdate()** assumes the current century for two-digit years. For information about how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *fmtstring* argument of **ifx_defmtdate()**. For more information, see the *IBM Informix GLS User's Guide*.

Return codes:

If you use an invalid date-string format, **ifx_defmtdate()** returns an error code and sets the internal DATE to the current date. The following are possible return codes.

0	The operation was successful.
-1204	The *input* parameter specifies an invalid year.
-1205	The *input* parameter specifies an invalid month.
-1206	The *input* parameter specifies an invalid day.
-1209	Because *input* does not contain delimiters between the year, month, and day, the length of *input* must be exactly six or eight bytes.
-1212	*fmtstring* does not specify a year, a month, and a day.

The ifx_dtcvasc() function

The **ifx_dtcvasc()** function converts a string that conforms to ANSI SQL standard for a DATETIME value to a **datetime** value. For information about the ANSI SQL DATETIME standard, see 6-7.

Syntax:
```
mint dtcvasc(str, d, dbcentury)
   char *str;
   dtime_t *d;
   char dbcentury;
```

str	A pointer to the buffer that contains an ANSI-standard DATETIME string.
d	A pointer to an initialized **datetime** variable.
dbcentury	Can be one of the following characters, which determines which century to apply to the year portion of the date:

	R	Present. The function uses the two high-order digits of the current year to expand the year value.
	P	Past. The function uses the past and present centuries to expand the year value. It compares these two dates against the current date and uses the century that is prior to the cur rent century. If both dates are prior to the current date, the function uses the century closest to the current date.
	F	Future. The function uses the present and the next centuries to expand the year value. It compares these against the current date and uses the century that is later than the current date. If both dates are later than the current date, the function uses the date closest to the current date.

C   Closest. The function uses the past, present, and next centuries to expand the year value. It chooses the century that is closest to the current date.

Usage:

You must initialize the **datetime** variable in *d* with the qualifier that you want this variable to have.

The character string in *str* must have values that conform to the **year to second** qualifier in the ANSI SQL format. The *str* string can have leading and trailing spaces. However, from the first significant digit to the last, *str* can only contain characters that are digits and delimiters that conform to the ANSI SQL standard for DATETIME values.

If you specify a year value as one or two digits, the **ifx_dtcvasc()** function uses the value of the *dbcentury* argument to determine which century to use. If you do not set the *dbcentury* argument, **ifx_dtcvasc()** uses the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **ifx_dtcvasc()** assumes the current century for two-digit years. For information about the **DBCENTURY** environment variable, see the *IBM Informix Guide to SQL: Reference*.

If the character string is an empty string, the **ifx_dtcvasc()** function sets to null the value to which *d* points. If the character string is acceptable, the function sets the value in the **datetime** variable and returns zero. Otherwise, the function leaves the variable unchanged and returns a negative error code.

Return codes:

0	Conversion was successful.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of **datetime** or **interval**.
-1262	Non-numeric character in **datetime** or **interval**.
-1263	A field in a **datetime** or **interval** value is out of range or incorrect.
-1264	Extra characters exist at the end of a **datetime** or **interval**.
-1265	Overflow occurred on a **datetime** or **interval** operation.
-1266	A **datetime** or **interval** value is incompatible with the operation.
-1267	The result of a **datetime** computation is out of range.
-1268	A parameter contains an invalid **datetime** qualifier.

The ifx_dtcvfmtasc() function

The **ifx_dtcvfmtasc()** function uses a formatting mask to convert a character string to a **datetime** value.

Syntax:

```
mint ifx_dtcvfmtasc(input, fmtstring, d, dbcentury)
   char *input;
   char *fmtstring;
   dtime_t *d;

   char dbcentury;
```

input	A pointer to the buffer that contains the string to convert.
fmtstring	A pointer to the buffer that contains the formatting mask to use for the input string. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*).
d	A pointer to the initialized **datetime** variable.
dbcentury	Can be one of the following characters, which determines which century to apply to the year portion of the date:

R	Present. The function uses the two high-order digits of the current year to expand the year value.
P	Past. The function uses the past and present centuries to expand the year value. It compares these two dates against the current date and uses the century that is prior to the current century. If both dates are prior to the current date, the function uses the century closest to the current date.
F	Future. The function uses the present and the next centuries to expand the year value. It compares these against the current date and uses the century that is later than the current date. If both dates are later than the current date, the function uses the date closest to the current date.
C	Closest. The function uses the past, present, and next centuries to expand the year value. It chooses the century that is closest to the current date.

Usage:

You must initialize the **datetime** variable in *d* with the qualifier that you want this variable to have. The **datetime** variable does not need to specify the same qualifier that the formatting mask implies. When the **datetime** qualifier is different from the implied formatting-mask qualifier, **ifx_dtcvfmtasc()** extends the **datetime** value (as if it had called the **dtextend()** function).

All qualifier fields in the character string in *input* must be contiguous. In other words, if the qualifier is **hour to second**, you must specify all values for **hour**, **minute**, and **second** somewhere in the string, or the **ifx_dtcvfmtasc()** function returns an error.

The *input* character string can have leading and trailing spaces. However, from the first significant digit to the last, *input* can contain only digits and delimiters that are appropriate for the qualifier fields that the formatting mask implies. For more information about acceptable digits and delimiters for a DATETIME value, see the "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

The **ifx_dtcvfmtasc()** function returns an error if the formatting mask, *fmtstring*, is an empty string. If *fmtstring* is a null pointer, the **ifx_dtcvfmtasc()** function must determine the format to use when it reads the character string in *input*. When you use the default locale, the function uses the following precedence:

1. The format that the **DBTIME** environment variable specifies (if **DBTIME** is set). For more information about **DBTIME**, see the *IBM Informix Guide to SQL: Reference*.

2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set). For more information about **GL_DATETIME**, see the *IBM Informix GLS User's Guide*.

3. The default date format conforms to the standard ANSI SQL format:

   ```
   %iY-%m-%d %H:%M:%S
   ```

The ANSI SQL format specifies a qualifier of **year to second** for the output. You can express the year as four digits (2007) or as two digits (07). When you use a two-digit year (**%y**) in a formatting mask, the **ifx_dtcvfmtasc()** function uses the value of the *dbcentury* argument to determine which century to use. If you do not set the *dbcentury* argument, **ifx_dtcvfmtasc()** uses the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **ifx_dtcvfmtasc()** assumes the current century for two-digit years. For information about the **DBCENTURY** environment variable, see the *IBM Informix Guide to SQL: Reference*.

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **ifx_dtcvfmtasc()** uses the default DATETIME format that the locale defines. For more information, see the *IBM Informix GLS User's Guide*.

When the character string and the formatting mask are acceptable, the **ifx_dtcvfmtasc()** function sets the **datetime** variable in *d* and returns zero. Otherwise, it returns an error code and the **datetime** variable contains an unpredictable value.

Return codes:

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

The ifx_dttofmtasc() function

The **ifx_dttofmtasc()** function uses a formatting mask to convert a **datetime** variable to a character string.

Syntax:

```
mint dttofmtasc(dtvalue, output, str_len, fmtstring, dbcentury)
   dtime_t *dtvalue;
   char *outbuf;
   mint buflen;
   char *fmtstring;
```

d	A pointer to the initialized **datetime** variable to convert.
output	A pointer to the buffer that receives the string for the value in d.
str_len	The length of the output buffer.
fmtstring	A pointer to the buffer that contains the formatting mask to use for the output string. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*).
dbcentury	Can be one of the following characters, which determines which century to apply to the year portion of the date:

R   Present. The function uses the two high-order digits of the current year to expand the year value.

P   Past. The function uses the past and present centuries to expand the year value. It compares these two dates against the current date and uses the century that is prior to the cur rent century. If both dates are prior to the current date, the function uses the century closest to the current date.

F   Future. The function uses the present and the next centuries to expand the year value. It compares these against the current date and uses the century that is later than the current date. If both dates are later than the current date, the function uses the date closest to the current date.

C   Closest. The function uses the past, present, and next centuries to expand the year value. It chooses the century that is closest to the current date.

Usage:

You must initialize the **datetime** variable in *dtvalue* with the qualifier that you want the character string to have. If you do not initialize the **datetime** variable, the function returns an unpredictable value. The character string in *outbuf* does not include the qualifier or the parentheses that SQL statements use to delimit a DATETIME literal.

The formatting mask, *fmtstring*, does not need to imply the same qualifiers as the **datetime** variable. When the implied formatting-mask qualifier is different from the **datetime** qualifier, **dttofmtasc()** extends the **datetime** value (as if it called the **dtextend()** function).

If the formatting mask is an empty string, the function sets character string, *outbuf*, to an empty string. If *fmtstring* is a null pointer, the **dttofmtasc()** function must determine the format to use for the character string in *outbuf*. When you use the default locale, the function uses the following precedence:

1. The format that the **DBTIME** environment variable specifies (if **DBTIME** is set). For more information about **DBTIME**, see the *IBM Informix Guide to SQL: Reference*.
2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set). For more information about **GL_DATETIME**, see the *IBM Informix GLS User's Guide*.
3. The default date format that conforms to the standard ANSI SQL format:
   ```
   %iY-%m-%d %H:%M:%S
   ```

When you use a two-digit year (**%y**) in a formatting mask, the **dttofmtasc()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **dttofmtasc()** assumes the present century for two-digit years. For information about how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **dttofmtasc()** uses the default DATETIME format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide*.

Return codes:

0	The conversion was successful.
<0	The conversion failed. Check the text of the error message.

The ifx_getenv() function

The **ifx_getenv()** function retrieves the value of a current environment variable.

Syntax:

```
char *ifx_getenv( varname );
    const char *varname;
```

varname   A pointer to a buffer that contains the name of an environment variable.

Usage:

The **ifx_getenv()** function searches for the environment variable in the following order:

1. Table of IBM Informix environment variables that the application has modified or defined with the **ifx_putenv()** function or directly (the **InetLogin** structure)
2. Table of Informix environment variables that the user has defined in the Registry with the **Setnet32** utility
3. Non-Informix environment variables retrieved from the C runtime environment variables
4. Table of defined defaults for Informix environment variables

The **ifx_getenv()** function is not case sensitive. You can specify the name of the environment variable in any case.

The **ifx_getenv()** function operates only on the data structures accessible to the C runtime library and not on the environment segment that the operating system creates for the process. Therefore, programs that use **ifx_getenv()** might retrieve invalid information.

The **ifx_putenv()** and **ifx_getenv()** functions use the copy of the environment to which the global variable **_environ** points to access the environment.

The following program fragment uses **ifx_getenv()** to retrieve the current value of the **INFORMIXDIR** environment variable:

```
char InformixDirVal[100];

/* Get current value of INFORMIXDIR */
InformixDirVal = ifx_getenv( "informixdir" );
/* Check if INFORMIXDIR is set */
If( InformixDirVal != NULL )
   printf( "Current INFORMIXDIR value is %\n", InformixDirVal );
```

Return codes:

The **ifx_getenv()** function returns a pointer to the Informix environment table entry that contains *varname*, or returns NULL if the function does not find *varname* in the table.

> **Restriction::** Do not use the returned pointer to modify the value of the environment variable. Use the **ifx_putenv()** function instead. If **ifx_getenv()** does not find "varname" in the Informix environment table, the return value is NULL.

The ifx_getcur_conn_name() function

The **ifx_getcur_conn_name()** function returns the name of the current connection.

Syntax:

```
char *ifx_getcur_conn_name(void);
```

Usage:

The current connection is the active database server connection that is currently sending SQL requests to the database server and possibly receiving data from the database server. In a callback function, the current connection is the current connection at the time when the callback was registered with a call to the **sqlbreakcallback()** function. The current connection name is the explicit name of the current connection. If the CONNECT statement that establishes a connection does not include the AS clause, the connection does not have an explicit name.

Return codes:

Name of current connection	Successfully obtained current connection name
Null pointer	Unable to obtain current connection name or current connection does not have an explicit name

The ifx_getserial8() function

The **ifx_getserial8()** function returns the SERIAL8 value of the last inserted row into an **int8** host variable.

Syntax:

```
void ifx_getserial8(serial8_val)
    ifx_int8_t *serial8_val;
```

serial8_val   A pointer to the **int8** structure where **ifx_getserial8()** places the newly-inserted SERIAL8 value.

Usage:

Use the **ifx_getserial8()** function after you insert a row that contains a SERIAL8 column. The function returns the new SERIAL8 value in the **int8** variable, *serial8_val*, which you declare. If the INSERT statement generated a new SERIAL8 value, the *serial8_val* points to a value greater than zero. A SERIAL8 value of zero or null indicates an invalid INSERT; the INSERT might have failed or might not have been performed.

Examples:

```
EXEC SQL BEGIN DECLARE SECTION;
    int8 order_num;
    int8 rep_num;
    char str[20];
EXEC SQL END DECLARE SECTION;

EXEC SQL create table order2
```

```
(
   order_number SERIAL8(1001),
   order_date DATE,
   customer_num INTEGER,
   backlog CHAR(1),
   po_num CHAR(10),
   paid_date DATE,
   sales_rep INT8
);
EXEC SQL insert into order2 (order_number, sales_rep)
   values (0, :rep_num);
if (SQLCODE == 0)
{
   ifx_getserial8(order_num);
   if (ifx_int8toasc(&order_num, str, 20) == 0)
      printf("New order number is %s\n", str);
}
```

The ifx_int8add() function

The **ifx_int8add()** function adds two **int8** type values.

Syntax:

```
mint ifx_int8add(n1, n2, sum)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *sum;
```

n1   A pointer to the **int8** structure that contains the first operand.

n2   A pointer to the **int8** structure that contains the second operand.

sum   A pointer to the **int8** structure that contains the sum of *n1 + n2*.

Usage:

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

Return codes:

0   The operation was successful.

-1284   The operation resulted in overflow or underflow.

Examples:

The file **int8add.ec** in the **demo** directory contains the following sample program.

```
*int8add.ec *

   The following program obtains the sum of two INT8 type values.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "6";
char string2[] = "9,223,372,036,854,775";
char string3[] = "999,999,999,999,999,9995";
char result[41];

main()
{
```

```
          mint x;
          ifx_int8_t num1, num2, num3, sum;

          printf("INT8 Sample ESQL Program running.\n\n");

          if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
              {
              printf("Error %d in converting string1 to INT8\n", x);
              exit(1);
              }
          if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
              {
              printf("Error %d in converting string2 to INT8\n", x);
              exit(1);
              }
        if (x = ifx_int8add(&num1, &num2, &sum))     /* adding the first two INT8s */
              {
              printf("Error %d in adding INT8s\n", x);
              exit(1);
              }
          if (x = ifx_int8toasc(&sum, result, sizeof(result)))
              {
              printf("Error %d in converting INT8 result to string\n", x);
              exit(1);
              }
          result[40] = '\0';
          printf("\t%s + %s = %s\n", string1, string2, result); /* display result */

    /* attempt to convert to INT8 value that is too large*/

          if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
              {
              printf("Error %d in converting string3 to INT8\n", x);
              exit(1);
              }
          if (x = ifx_int8add(&num2, &num3, &sum))
              {
              printf("Error %d in adding INT8s\n", x);
              exit (1);
              }
          if (x = ifx_int8toasc(&sum, result, sizeof(result)))
              {
              printf("Error %d in converting INT8 result to string\n", x);
              exit(1);
              }
          result[40] = '\0';
          printf("\t%s + %s = %s\n", string2, string3, result); /* display result */

          printf("\nINT8 Sample Program over.\n\n");
          exit(0);
}
```

Output:

```
INT8 Sample ESQL Program running.

   6 + 9,223,372,036,854,775 = 9223372036854781
Error -1284 in converting string3 to INT8
```

The ifx_int8cmp() function

The **ifx_int8cmp()** function compares two **int8** type numbers.

Syntax:

```
mint ifx_int8cmp(n1, n2)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
```

n1	A pointer to the **int8** structure that contains the first number to compare.
n2	A pointer to the **int8** structure that contains the second number to compare.

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.
INT8UNKNOWN	Either value is null.

Examples:

The file **int8cmp.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8cmp.ec *

    The following program compares INT8s types and displays
    the results.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-999,888,777,666";
char string2[] = "-12,345,678,956,546";
char string3[] = "123,456,780,555,224,456";
char string4[] = "123,456,780,555,224,456";
char string5[] = "";

main()
{
    mint x;
    ifx_int8_t num1, num2, num3, num4, num5;

    printf("IFX_INT8CMP Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to int8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to int8\n", x);
        exit(1);
        }
```

```
        if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
            {
            printf("Error %d in converting string3 to int8\n", x);
            exit(1);
            }
        if (x = ifx_int8cvasc(string4, strlen(string4), &num4))
            {
            printf("Error %d in converting string4 to int8\n", x);
            exit(1);
            }
        if (x = ifx_int8cvasc(string5, strlen(string5), &num5))
            {
            printf("Error %d in converting string5 to int8\n", x);
            exit(1);
            }
    printf("num1 = %s\nnum2 = %s\n", string1, string2);
    printf("num3 = %s\nnum4 = %s\n", string3, string4);
    printf("num5 = %s\n", "NULL");
    printf("\nExecuting: ifx_int8cmp(&num1, &num2)\n");
    printf("  Result = %d\n", ifx_int8cmp(&num1, &num2));
    printf("Executing: ifx_int8cmp(&num2, &num3)\n");
    printf("  Result = %d\n", ifx_int8cmp(&num2, &num3));
    printf("Executing: ifx_int8cmp(&num1, &num3)\n");
    printf("  Result = %d\n", ifx_int8cmp(&num1, &num3));
    printf("Executing: ifx_int8cmp(&num3, &num4)\n");
    printf("  Result = %d\n", ifx_int8cmp(&num3, &num4));
    printf("Executing: ifx_int8cmp(&num1, &num5)\n");
    x = ifx_int8cmp(&num1, &num5);
    if(x == INT8UNKNOWN)
        printf("RESULT is INT8UNKNOWN.  One of the INT8 values in null.\n");
    else
        printf("  Result = %d\n", x);
    printf("\nIFX_INT8CMP Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
IFX_INT8CMP Sample ESQL Program running.

Number 1 = -999,888,777,666    Number 2 = -12,345,678,956,546
Number 3 = 123,456,780,555,224,456    Number 4 = 123,456,780,555,224,456
Number 5 =

Executing: ifx_int8cmp(&num1, &num2)
  Result = 1
Executing: ifx_int8cmp(&num2, &num3)
  Result = -1
Executing: ifx_int8cmp(&num1, &num3)
  Result = -1
Executing: ifx_int8cmp(&num3, &num4)
  Result = 0
Executing: ifx_int8cmp(&num1, &num5)
RESULT is INT8UNKNOWN.  One of the INT8 values in null.

IFX_INT8CMP Sample Program over.
```

The ifx_int8copy() function

The **ifx_int8copy()** function copies one **int8** structure to another.

Syntax:

```
void ifx_int8copy(source, target)
   ifx_int8_t *source;
   ifx_int8_t *target;
```

source   A pointer to the **int8** structure that contains the source **int8** value
                  to copy.

target   A pointer to the target **int8** structure.

The **ifx_int8copy()** function does not return a status value. To determine the
success of the copy operation, look at the contents of the **int8** structure to which
the *target* argument points.

Examples:

The file **int8copy.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8copy.ec *

   The following program copies one INT8 number to another.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-12,888,999,555,333";
char result[41];

main()
{
    mint x;
    ifx_int8_t num1, num2;

    printf("IFX_INT8COPY Sample ESQL Program running.\n\n");

    printf("String = %s\n", string1);
    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    printf("Executing: ifx_int8copy(&num1, &num2)\n");
    ifx_int8copy(&num1, &num2);
    if (x = ifx_int8toasc(&num2, result, sizeof(result)))
        {
        printf("Error %d in converting num2 to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("Destination = %s\n", result);

    printf("\nIFX_INT8COPY Sample Program over.\n\n");
    exit(0);
}
```

Output:
```
IFX_INT8COPY Sample ESQL Program running.

String = -12,888,999,555,333
```

```
Executing: ifx_int8copy(&num1, &num2)
Destination = -12888999555333

IFX_INT8COPY Sample Program over
```

The ifx_int8cvasc() function

The **ifx_int8cvasc()** function converts a value held as printable characters in a C **char** type into an **int8** type number.

Syntax:
```
mint ifx_int8cvasc(strng_val, len, int8_val)
   char *strng_val
   mint len;
   ifx_int8_t *int8_val;
```

strng_val	A pointer to a string.
len	The length of the strng_val string.
int8_val	A pointer to the **int8** structure where **ifx_int8cvasc()** places the result of the conversion.

Usage:

The character string, *strng_val*, can contain the following symbols:
- A leading sign, either a plus (+) or minus (-).
- An exponent that is preceded by either e or E. You can precede the exponent by a sign, either a plus (+) or minus (-).

The *strng_val* character string should not contain a decimal separator or digits to the right of the decimal separator. The **ifx_int8svasc()** function truncates the decimal separator and any digits to the right of the decimal separator. The **ifx_int8cvasc()** function ignores leading spaces in the character string.

When you use a nondefault locale (one other than U.S. English), **ifx_int8cvasc()** supports non-ASCII characters in the *strng_val* character string. For more information, see the *IBM Informix GLS User's Guide*.

Return codes:

0	The conversion was successful.
-1213	The string has non-numeric characters.
-1284	The operation resulted in overflow or underflow.

Examples:

The file **int8cvasc.ec** in the **demo** directory contains the following sample program.
```
/*
   * ifx_in8cvasc.ec *

    The following program converts three strings to INT8
    types and displays the values stored in each field of
    the INT8 structures.
*/

#include <stdio.h>
```

```
EXEC SQL include "int8.h";

char string1[] = "-12,555,444,333,786,456";
char string2[] = "480";
char string3[] = "5.2";
main()
{
    mint x;
    ifx_int8_t num1, num2, num3;
    void nullterm(char *, mint);

    printf("IFX_INT8CVASC Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }

    /*  Display the exponent, sign value and number of digits in num1. */

    ifx_int8toasc(&num1, string1, sizeof(string1));
    nullterm(string1, sizeof(string1));
    printf("The value of the first INT8 is = %s\n", string1);

    /*  Display the exponent, sign value and number of digits in num2. */

    ifx_int8toasc(&num2, string2, sizeof(string2));
    nullterm(string2, sizeof(string2));
    printf("The value of the 2nd INT8 is = %s\n", string2);

    /*  Display the exponent, sign value and number of digits in num3. */
    /*  Note that the decimal is truncated */

    ifx_int8toasc(&num3, string3, sizeof(string3));
    nullterm(string3, sizeof(string3));
    printf("The value of the 3rd INT8 is = %s\n", string3);

    printf("\nIFX_INT8CVASC Sample Program over.\n\n");
    exit(0);
}
void nullterm(char *str, mint size)
{
    char *end;

    end = str + size;
    while(*str && *str > ' ' && str <= end)
        ++str;
    *str = '\0';
}
```

Output:

```
              IFX_INT8CVASC Sample ESQL Program running.

              The value of the first INT8 is = -12555444333786456
              The value of the 2nd INT8 is = 480
              The value of the 3rd INT8 is = 5

              IFX_INT8CVASC Sample Program over.
```

The ifx_int8cvdbl() function

The **ifx_int8cvdbl()** function converts a C **double** type number into an **int8** type number.

Syntax:
```
mint ifx_int8cvdbl(dbl_val, int8_val)
   double dbl_val;
   ifx_int8_t *int8_val;
```

dbl_val   The **double** value that **ifx_int8cvdbl()** converts to an **int8** type value.

int8_val   A pointer to the **int8** structure where **ifx_int8cvdbl()** places the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8cvdbl.ec** in the **demo** directory contains the following sample program.

```
/*
   * int8cvdbl.ec *

    The following program converts two double type numbers to
    INT8 types and displays the results.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char result[41];

main()
{
    mint x;
    ifx_int8_t num;
    double d = 2147483647;

printf("IFX_INT8CVDBL Sample ESQL Program running.\n\n");

    printf("Number 1 (double) = 1234.5678901234\n");
    if (x = ifx_int8cvdbl((double)1234.5678901234, &num))
        {
        printf("Error %d in converting double1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8toasc(&num, result, sizeof(result)))
        {
```

```
                    printf("Error %d in converting INT8 to string\n", x);
                    exit(1);
                    }
              result[40] = '\0';
              printf("  String Value = %s\n", result);

       /* notice that the ifx_int8cvdbl function truncates digits to the
       right of a decimal separator. */

              printf("Number 2 (double) = %.1f\n", d);
              if (x = ifx_int8cvdbl(d, &num))
                    {
                    printf("Error %d in converting double2 to INT8\n", x);
                    exit(1);
                    }
              if (x = ifx_int8toasc(&num, result, sizeof(result)))
                    {
                    printf("Error %d in converting second INT8 to string\n", x);
                    exit(1);
                    }
              result[40] = '\0';
              printf("  String Value = %s\n", result);

              printf("\nIFX_INT8CVDBL Sample Program over.\n\n");
              exit(0);
       }
```

Output:

```
IFX_INT8CVDBL Sample ESQL Program running.

Number 1 (double) = 1234.5678901234
  String Value = 1234
Number 2 (double) = 2147483647.0
  String Value = 2147483647

IFX_INT8CVDBL Sample Program over.
```

The ifx_int8cvdec() function

The **ifx_int8cvdec()** function converts a **decimal** type value into an **int8** type value.

Syntax:

```
mint ifx_int8cvdec(dec_val, int8_val)
   dec_t *dec_val;
   ifx_int8_t *int8_val;
```

dec_val   A pointer to the **decimal** structure that **ifx_int8cvdec()** converts to an **int8** type value.

int8_val   A pointer to the **int8** structure where **ifx_int8cvdec()** places the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8cdec.ec** in the **demo** directory contains the following sample program.

```
/*
 * ifx_int8cvdec.ec *

   The following program converts two INT8s types to DECIMALS and displays
   the results.
*/

#include <stdio.h>

EXEC SQL include decimal;
EXEC SQL include "int8.h";

char string1[] = "2949.3829398204382";
char string2[] = "3238299493";
char result[41];

main()
{
    mint x;
    ifx_int8_t n;
    dec_t num;

    printf("IFX_INT8CVDEC Sample ESQL Program running.\n\n");

    if (x = deccvasc(string1, strlen(string1), &num))
        {
        printf("Error %d in converting string1 to DECIMAL\n", x);
        exit(1);
        }
    if (x = ifx_int8cvdec(&num, &n))
        {
        printf("Error %d in converting DECIMAL1 to INT8\n", x);
        exit(1);
        }

/*  Convert the INT8 to ascii and display it. Note that the
    digits to the right of the decimal are truncated in the
    conversion.
*/

    if (x = ifx_int8toasc(&n, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("String 1 Value = %s\n", string1);
    printf("  INT8 type value = %s\n", result);


    if (x = deccvasc(string2, strlen(string2), &num))
        {
        printf("Error %d in converting string2 to DECIMAL\n", x);
        exit(1);
        }
    if (x = ifx_int8cvdec(&num, &n))
        {
        printf("Error %d in converting DECIMAL2 to INT8\n", x);
        exit(1);
        }
    printf("String 2 = %s\n", string2);

/* Convert the INT8 to ascii to display value. */
```

```
        if (x = ifx_int8toasc(&n, result, sizeof(result)))
            {
            printf("Error %d in converting INT8 to string\n", x);
            exit(1);
            }
        result[40] = '\0';
        printf("  INT8 type value = %s\n", result);

        printf("\nIFX_INT8CVDEC Sample Program over.\n\n");
        exit(0);
}
```

Output:

```
IFX_INT8CVDEC Sample ESQL Program running.

  String 1 Value = 2949.3829398204382
INT8 type value = 2949
String 2 = 3238299493
  INT8 type value = 3238299493

IFX_INT8CVDEC Sample Program over.
```

The ifx_int8cvflt() function

The **ifx_int8cvflt()** function converts a C **float** type number into an **int8** type number.

Syntax:

```
mint ifx_int8cvflt(flt_val, int8_val)
    double flt_val;
    ifx_int8_t *int8_val;
```

flt_val   The **float** value that **ifx_int8cvflt** converts to an **int8** type value.

int8_val   A pointer to the **int8** structure where **ifx_int8cvflt()** places the result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8cvflt.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8cvflt.ec *

   The following program converts two floats to INT8 types and displays
   the results.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char result[41];

main()
{
    mint x;
```

```
        ifx_int8_t num;

        printf("IFX_INT8CVFLT Sample ESQL Program running.\n\n");

        printf("Float 1 = 12944.321\n");

    /* Note that in the following conversion, the digits to the
       right of the decimal are ignored. */

        if (x = ifx_int8cvflt(12944.321, &num))
            {
            printf("Error %d in converting float1 to INT8\n", x);
            exit(1);
            }

    /* Convert int8 to ascii to display value. */

        if (x = ifx_int8toasc(&num, result, sizeof(result)))
            {
            printf("Error %d in converting INT8 to string\n", x);
            exit(1);
            }
        result[40] = '\0';
        printf("  The INT8 type value is = %s\n", result);
        printf("Float 2 = -33.43\n");

    /* Note that in the following conversion, the digits to the
       right of the decimal are ignored. */

        if (x = ifx_int8cvflt(-33.43, &num))
            {
            printf("Error %d in converting float2 to INT8\n", x);
            exit(1);
            }
        if (x = ifx_int8toasc(&num, result, sizeof(result)))
            {
            printf("Error %d in converting INT8 to string\n", x);
            exit(1);
            }
        result[40] = '\0';
        printf("  The second INT8 type value is = %s\n", result);

        printf("\nIFX_INT8CVFLT Sample Program over.\n\n");
        exit(0);
}
```

Output:

```
IFX_INT8CVFLT Sample ESQL Program running.

Float 1 = 12944.321
  The INT8 type value is = 12944
Float 2 = -33.43
  The second INT8 type value is = -33

IFX_INT8CVFLT Sample Program over.
```

The ifx_int8cvint() function

The **ifx_int8cvint()** function converts a C **int** type number into an **int8** type
number.

Syntax:

```
mint ifx_int8cvint(int_val, int8_val)
   mint int_val;
   ifx_int8_t *int8_val;
```

int_val   The **mint** value that **ifx_int8cvint()** converts to an **int8** type value.

int8_val   A pointer to the **int8** structure where **ifx_int8cvint()** places the
                   result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8cvint.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8cvint.ec *

    The following program converts two integers to INT8
    types and displays the results.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char result[41];

main()
{
    mint x;
    ifx_int8_t num;

    printf("IFX_INT8CVINT Sample ESQL Program running.\n\n");

    printf("Integer 1 = 129449233\n");
    if (x = ifx_int8cvint(129449233, &num))
        {
        printf("Error %d in converting int1 to INT8\n", x);
        exit(1);
        }

/* Convert int8 to ascii to display value. */

    if (x = ifx_int8toasc(&num, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("  The INT8 type value is = %s\n", result);

    printf("Integer 2 = -33\n");
    if (x = ifx_int8cvint(-33, &num))
        {
        printf("Error %d in converting int2 to INT8\n", x);
        exit(1);
        }

/* Convert int8 to ascii to display value. */
```

```
        if (x = ifx_int8toasc(&num, result, sizeof(result)))
            {
            printf("Error %d in converting INT8 to string\n", x);
            exit(1);
            }
        result[40] = '\0';
        printf("  The second INT8 type value is = %s\n", result);

        printf("\nIFX_INT8CVINT Sample Program over.\n\n");
        exit(0);
}
```

Output:
```
IFX_INT8CVINT Sample ESQL Program running.

Integer 1 = 129449233
  The INT8 type value is = 129449233
Integer 2 = -33
  The second INT8 type value is = -33

IFX_INT8CVINT Sample Program over.
```

The ifx_int8cvlong() function

The **ifx_int8cvlong()** function converts a C **long** type value into an **int8** type value.

Syntax:
```
mint ifx_int8cvlong(lng_val, int8_val)
    int4 lng_val;
    ifx_int8_t *int8_val;
```

lng_val	The **int4** integer that **ifx_int8cvlong()** converts to an **int8** type value.
int8_val	A pointer to the **int8** structure where **ifx_int8cvlong()** places the result of the conversion.

Return codes:

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

Examples:

The file **int8cvlong.ec** in the **demo** directory contains the following sample program.
```
/*
   * ifx_int8cvlong.ec *

   The following program converts two longs to INT8
   types and displays the results.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char result[41];

main()
{
```

```
     mint x;
     ifx_int8_t num;
     int4 n;

   printf("IFX_INT8CVLONG Sample ESQL Program running.\n\n");

    printf("Long Integer 1 = 129449233\n");
    if (x = ifx_int8cvlong(129449233L, &num))
        {
        printf("Error %d in converting long to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8toasc(&num, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("  String for INT8 type value = %s\n", result);

    n = 2147483646;                  /* set n */
    printf("Long Integer 2 = %d\n", n);
    if (x = ifx_int8cvlong(n, &num))
        {
        printf("Error %d in converting long to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8toasc(&num, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("   String for INT8 type value = %s\n", result);

    printf("\nIFX_INT8CVLONG Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
IFX_INT8CVLONG Sample ESQL Program running.

Long Integer 1 = 129449233
  String for INT8 type value = 129449233
Long Integer 2 = 2147483646
   String for INT8 type value = 2147483646

IFX_INT8CVLONG Sample Program over.
```

The ifx_int8div() function

The **ifx_int8div()** function divides two **int8** type values.

Syntax:

```
mint ifx_int8div(n1, n2, quotient)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *quotient;
```

n1   A pointer to the **int8** structure that contains the dividend.

n2   A pointer to the **int8** structure that contains the divisor.

quotient   A pointer to the **int8** structure that contains the quotient of *n1/n2*.

Usage:

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

Return codes:

0   The operation was successful.

-1202   The operation attempted to divide by zero.

Examples:

The file **int8div.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8div.ec *

   The following program divides two INT8 numbers and displays the result.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "480,999,777,666,345,567";
char string2[] = "80,765,456,765,456,654";
char result[41];

main()
{
    mint x;
    ifx_int8_t num1, num2, dvd;

    printf("IFX_INT8DIV Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8div(&num1, &num2, &dvd))
        {
        printf("Error %d in dividing num1 by num2\n", x);
        exit(1);
        }
    if (x = ifx_int8toasc(&dvd, result, sizeof(result)))
        {
        printf("Error %d in converting dividend to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("\t%s / %s = %s\n", string1, string2, result);

    printf("\nIFX_INT8DIV Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
IFX_INT8DIV Sample ESQL Program running.

   480,999,777,666,345,567 / 80,765,456,765,456,654 = 5

IFX_INT8DIV Sample Program over.
```

The ifx_int8mul() function

The **ifx_int8mul()** function multiplies two **int8** type values.

Syntax:

```
mint ifx_int8mul(n1, n2, product)
   ifx_int8_t *n1;
   ifx_int8_t *n2;
   ifx_int8_t *product;
```

n1	A pointer to the **int8** structure that contains the first operand.
n2	A pointer to the **int8** structure that contains the second operand.
product	A pointer to the **int8** structure that contains the product of *n1 * n2*.

Usage:

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

Return codes:

0	The operation was successful.
-1284	The operation resulted in overflow or underflow.

Examples:

The file **int8mul.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8mul.ec *

   The following program multiplies two INT8 numbers and
   displays the result.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "480,999,777,666,345";
char string2[] = "80";
char result[41];

main()
{
    mint x;
    ifx_int8_t num1, num2, prd;

    printf("IFX_INT8MUL Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
```

```
            }
        if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
            {
            printf("Error %d in converting string2 to INT8\n", x);
            exit(1);
            }
        if (x = ifx_int8mul(&num1, &num2, &prd))
            {
            printf("Error %d in multiplying num1 by num2\n", x);
            exit(1);
            }
        if (x = ifx_int8toasc(&prd, result, sizeof(result)))
            {
            printf("Error %d in converting product to string\n", x);
            exit(1);
            }
        result[40] = '\0';
        printf("\t%s * %s = %s\n", string1, string2, result);

        printf("\nIFX_INT8MUL Sample Program over.\n\n");
        exit(0);
}
```

Output:

```
IFX_INT8MUL Sample ESQL Program running.

    480,999,777,666,345 * 80 = 38479982213307600

IFX_INT8MUL Sample Program over.
```

The ifx_int8sub() function

The **ifx_int8sub()** function subtracts two **int8** type values.

Syntax:

```
mint ifx_int8sub(n1, n2, difference)
    ifx_int8_t *n1;
    ifx_int8_t *n2;
    ifx_int8_t *difference;
```

n1	A pointer to the **int8** structure that contains the first operand.
n2	A pointer to the **int8** structure that contains the second operand.
difference	A pointer to the **int8** structure that contains the difference of *n1* and *n2* (*n1* - *n2*).

Usage:

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

Return codes:

0	The subtraction was successful.
-1284	The subtraction resulted in overflow or underflow.

Examples:

The file **int8sub.ec** in the **demo** directory contains the following sample program.

```
/*
   *int8sub.ec *

    The following program obtains the difference of two INT8
    type values.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "6";
char string2[] = "9,223,372,036,854,775";
char string3[] = "999,999,999,999,999.5";
char result[41];

main()
{
    mint x;
    ifx_int8_t num1, num2, num3, sum;

    printf("IFX_INT8SUB Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }

    /* subtract num2 from num1 */

    if (x = ifx_int8sub(&num1, &num2, &sum))
        {
        printf("Error %d in subtracting INT8s\n", x);
        exit(1);
        }
    if (x = ifx_int8toasc(&sum, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 result to string\n", x);
        exit(1);
        }
    result[40] = '\0';
    printf("\t%s - %s = %s\n", string1, string2, result); /* display result */

    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }

    /* notice that digits right of the decimal are truncated. */

    if (x = ifx_int8sub(&num2, &num3, &sum))
        {
        printf("Error %d in subtracting INT8s\n", x);
        exit (1);
      }
    if (x = ifx_int8toasc(&sum, result, sizeof(result)))
        {
```

```
            printf("Error %d in converting INT8 result to string\n", x);
            exit(1);
            }
        result[40] = '\0';
        printf("\t%s - %s = %s\n", string2, string3, result); /* display result */

        printf("\nIFX_INT8SUB Sample Program over.\n\n");
        exit(0);
}
```

Output:

```
IFX_INT8SUB Sample ESQL Program running.

   6 - 9,223,372,036,854,775 = -9223372036854769
   9,223,372,036,854,775 - 999,999,999,999,999.5 = 8223372036854776

IFX_INT8SUB Sample Program over.
```

The ifx_int8toasc() function

The **ifx_int8toasc()** function converts an **int8** type number to a C **char** type value.

Syntax:

```
mint inf_int8toasc(int8_val, strng_val, len)
    ifx_int8_t *int8_val;
    char  *strng_val;
    mint  len;
```

int8_val	A pointer to the **int8** structure whose value **ifx_int8toasc()** converts to a text string.
strng_val	A pointer to the first byte of the character buffer where the **ifx_int8toasc()** function places the text string.
len	The size of **strng_val**, in bytes, minus 1 for the null terminator.

Usage:

If the **int8** number does not fit into a character string of length *len*, **ifx_int8toasc()** converts the number to an exponential notation. If the number still does not fit, **ifx_int8toasc()** fills the string with asterisks. If the number is shorter than the string, **ifx_int8toasc()** left-justifies the number and pads it on the right with blanks.

Because the character string that **ifx_int8toasc()** returns is not null terminated, your program must add a null character to the string before you print it.

When you use a nondefault locale (one other than U.S. English), **ifx_int8toasc()** supports non-ASCII characters in the *strng_val* character string. For more information, see the *IBM Informix GLS User's Guide*.

Return codes:

0	The conversion was successful.
-1207	The converted value does not fit into the allocated space.

Examples:

The file **int8toasc.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8toasc.ec *

    The following program converts three string
    constants to INT8 types and then uses ifx_int8toasc()
    to convert the INT8 values to C char type values.
*/

#include <stdio.h>
#define END sizeof(result)

EXEC SQL include "int8.h";

char string1[] = "-12,555,444,333,786,456";
char string2[] = "480";
char string3[] = "5.2";
char result[40];

main()
{
    mint x;
    ifx_int8_t num1, num2, num3;

    printf("IFX_INT8TOASC Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }
    printf("\nConverting INT8 back to ASCII\n");
    printf(" Executing: ifx_int8toasc(&num1, result, END - 1)");
    if (x = ifx_int8toasc(&num1, result, END - 1))
        printf("\tError %d in converting INT8 to string\n", x);
    else
        {
        result[END - 1] = '\0';            /* null terminate */
        printf("\n The value of the first INT8 is = %s\n", result);
        }
    printf("\nConverting second INT8 back to ASCII\n");
    printf(" Executing: ifx_int8toasc(&num2, result, END - 1)");
    if (x= ifx_int8toasc(&num2, result, END - 1))
        printf("\tError %d in converting INT8 to string\n", x);
    else
        {
        result[END - 1] = '\0';            /* null terminate */
        printf("\n The value of the 2nd INT8 is = %s\n", result);
        }

    printf("\nConverting third INT8 back to ASCII\n");
    printf(" Executing: ifx_int8toasc(&num3, result, END - 1)");
    /* note that the decimal is truncated */

    if (x= ifx_int8toasc(&num3, result, END - 1))
```

```
            printf("\tError %d in converting INT8 to string\n", x);
        else
            {
            result[END - 1] = '\0';                 /* null terminate */
            printf("\n The value of the 3rd INT8 is = %s\n", result);
            }
        printf("\nIFX_INT8TOASC Sample Program over.\n\n");
        exit(0);
}
```

Output:

```
IFX_INT8TOASC Sample ESQL Program running.


Converting INT8 back to ASCII
 Executing: ifx_int8toasc(&num1, result, sizeof(result)-1)
 The value of the first INT8 is = -12555444333786456

Converting second INT8 back to ASCII
 Executing: ifx_int8toasc(&num2, result, sizeof(result)-1)
 The value of the 2nd INT8 is = 480

Converting third INT8 back to ASCII
 Executing: ifx_int8toasc(&num3, result, END)
 The value of the 3rd INT8 is = 5

IFX_INT8TOASC Sample Program over.
```

The ifx_int8todbl() function

The **ifx_int8todbl()** function converts an **int8** type number into a C **double** type
number.

Syntax:
```
mint ifx_int8todbl(int8_val, dbl_val)
    ifx_int8_t *int8_val;
    double *dbl_val;
```

int8_val   A pointer to the **int8** structure whose value **ifx_int8todbl()**
                    converts to a **double** type value.

dbl_val   A pointer to a **double** value where **ifx_int8todbl()** places the result
                    of the conversion.

Usage:

The floating-point format of the host computer can result in loss of precision in the
conversion of an **int8** type number to a **double type number**.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8todbl.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8todbl.ec *

    The following program converts three strings to INT8
```

```
      types and then to C double types and displays the
      results.

*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-12,555,444,333,786,456";
char string2[] = "480";
char string3[] = "5.2";

main()
{
    mint x;
    double d =0;
    ifx_int8_t num1, num2, num3;

    printf("\nIFX_INT8TODBL Sample ESQL Program running.\n\n");

if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }
    printf("\nConverting INT8 to double");
    if (x= ifx_int8todbl(&num1, &d))
        {
        printf("\tError %d in converting INT8 to double\n", x);
        exit(1);
        }
else
        {
        printf("\nString 1= %s\n", string1);
        printf("INT8 value is = %.10f\n", d);
        }
printf("\nConverting second INT8 to double");
    if (x= ifx_int8todbl(&num2, &d))
        {
        printf("\tError %d in converting INT8 to double\n", x);
        exit(1);
        }
    else
        {
        printf("\nString2 = %s\n", string2);/*
      printf("INT8 value is =  %.10f\n",d);
        }
    printf("\nConverting third INT8 to double");
 /* Note that the decimal places will be truncated. */

    if (x= ifx_int8todbl(&num3, &d))
        {
        printf("\tError %d in converting INT8 to double\n", x);
```

```
            exit(1);
            }
        else
            {
            printf("\nString3 = %s\n", string3);
            printf("INT8 value is =  %.10f\n",d);
            }
        printf("\nIFX_INT8TODBL Sample Program over.\n\n");
        exit(0);
    }
```

Output:

```
IFX_INT8TODBL Sample ESQL Program running.


Converting INT8 to double

Executing: ifx_int8todbl(&num1,&d)
String 1= -12,555,444,333,786,456

 The value of the first double is = -12555444333786456.0000000000

Converting second INT8 to double

Executing: ifx_int8todbl(&num2, &d)
String2 = 480

 The value of the second double is =  480.0000000000

Converting third INT8 to double

Executing: ifx_int8todbl(&num3, &d)
String3 = 5.2

 The value of the third double is =  5.0000000000

IFX_INT8TODBL Sample Program over.
```

The ifx_int8todec() function

The **ifx_int8todec()** function converts an **int8** type number into a **decimal** type number.

Syntax:

```
mint ifx_int8todec(int8_val, dec_val)
    ifx_int8_t *int8_val;
    dec_t  *dec_val;
```

int8_val   A pointer to an **int8** structure whose value **ifx_int8todec()** converts
                  to a **decimal** type value.

dec_val   A pointer to a **decimal** structure in which **ifx_int8todec()** places the
                  result of the conversion.

Return codes:

0   The conversion was successful.

<0   The conversion was not successful.

Examples:

The file **int8todec.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8todec.ec *

   The following program converts three strings to INT8 types and
   converts the INT8 type values to decimal type values.
   Then the values are displayed.

*/

#include <stdio.h>

EXEC SQL include "int8.h";
#define END sizeof(result)

char string1[] = "-12,555,444,333,786,456";
char string2[] = "480";
char string3[] = "5.2";
char result [40];

main()
{
    mint x;
    dec_t d;
    ifx_int8_t num1, num2, num3;

    printf("IFX_INT8TODEC Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }

    printf("\n***Converting INT8 to decimal\n");
    printf("\nString 1= %s\n", string1);
    printf(" \nExecuting: ifx_int8todec(&num1,&d)");
    if (x= ifx_int8todec(&num1, &d))
        {
        printf("\tError %d in converting INT8 to decimal\n", x);
        exit(1);
        }
    else
        {
        printf("\nConverting Decimal to ASCII for display\n");
        printf("Executing: dectoasc(&d, result, END, -1)\n");
        if (x = dectoasc(&d, result, END, -1))
            printf("\tError %d in converting DECIMAL1 to string\n", x);
        else
            {
            result[END - 1] = '\0';              /* null terminate */
            printf("Result = %s\n", result);
            }
```

```
        }
    printf("\n***Converting second INT8 to decimal\n");
    printf("\nString2 = %s\n", string2);
    printf(" \nExecuting: ifx_int8todec(&num2, &d)");
    if (x= ifx_int8todec(&num2, &d))
        {
        printf("\tError %d in converting INT8 to decimal\n", x);
        exit(1);
        }
    else
        {
        printf("\nConverting Decimal to ASCII for display\n");
        printf("Executing: dectoasc(&d, result, END, -1)\n");
        if (x = dectoasc(&d, result, END, -1))
            printf("\tError %d in converting DECIMAL2 to string\n", x);
else
            {
            result[END - 1] = '\0';                    /* null terminate */
            printf("Result = %s\n", result);
            }
        }
    printf("\n***Converting third INT8 to decimal\n");
    printf("\nString3 = %s\n", string3);
    printf(" \nExecuting: ifx_int8todec(&num3, &d)");
    if (x= ifx_int8todec(&num3, &d))
        {
        printf("\tError %d in converting INT8 to decimal\n", x);
        exit(1);
        }
    else
        {
        printf("\nConverting Decimal to ASCII for display\n");
        printf("Executing: dectoasc(&d, result, END, -1)\n");

        /* note that the decimal is truncated */

        if (x = dectoasc(&d, result, END, -1))
            printf("\tError %d in converting DECIMAL3 to string\n", x);
        else
            {
            result[END - 1] = '\0';                    /* null terminate */
            printf("Result = %s\n", result);
            }
        }
    printf("\nIFX_INT8TODEC Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
IFX_INT8TODEC Sample ESQL Program running.


***Converting INT8 to decimal

String 1= -12,555,444,333,786,456

Executing: ifx_int8todec(&num1,&d)
Converting Decimal to ASCII for display
Executing: dectoasc(&d, result, END, -1)
Result = -12555444333786456.0

***Converting second INT8 to decimal

String2 = 480
```

```
Executing: ifx_int8todec(&num2, &d)
Converting Decimal to ASCII for display
Executing: dectoasc(&d, result, END, -1)
Result = 480.0

***Converting third INT8 to decimal

String3 = 5.2

Executing: ifx_int8todec(&num3, &d)
Converting Decimal to ASCII for display
Executing: dectoasc(&d, result, END, -1)
Result = 5.0

IFX_INT8TODEC Sample Program over.
```

The ifx_int8toflt() function

The **ifx_int8toflt()** function converts an **int8** type number into a C **float** type number.

Syntax:
```
mint ifx_int8toflt(int8_val, flt_val)
    ifx_int8_t *int8_val;
    float *flt_val;
```

int8_val   A pointer to an **int8** structure whose value **ifx_int8toflt()** converts to a **float** type value.

flt_val   A pointer to a **float** value where **ifx_int8toflt()** places the result of the conversion.

Usage:

The **ifx_int8toflt()** library function converts an **int8** value to a C float. The size of a C float depends upon the hardware and operating system of the computer you are using.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8toflt.ec** in the **demo** directory contains the following sample program.
```
/*
   * ifx_int8toflt.ec *

   The following program converts three strings to
   INT8 values and then to float values and
   displays the results.

*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-12,555.765";
```

```
                char string2[] = "480.76";
                char string3[] = "5.2";

                main()
                {
                    mint x;
                    float f =0.0;
                    ifx_int8_t num1, num2, num3;

                    printf("\nIFX_INT8TOFLT Sample ESQL Program running.\n\n");
                    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
                        {
                        printf("Error %d in converting string1 to INT8\n", x);
                        exit(1);
                        }
                    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
                        {
                        printf("Error %d in converting string2 to INT8\n", x);
                        exit(1);
                        }
                    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
                        {
                        printf("Error %d in converting string3 to INT8\n", x);
                        exit(1);
                        }

                    printf("\nConverting INT8 to float\n");
                    if (x= ifx_int8toflt(&num1, &f))
                        {
                        printf("\tError %d in converting INT8 to float\n", x);
                        exit(1);
                        }
                    else
                        {
                        printf("String 1= %s\n", string1);
                        printf("INT8 value is = %f\n", f);
                        }
                    printf("\nConverting second INT8 to float\n");
                    if (x= ifx_int8toflt(&num2, &f))
                        {
                        printf("\tError %d in converting INT8 to float\n", x);
                        exit(1);
                        }
                    else
                        {
                        printf("String2 = %s\n", string2);
                        printf("INT8 value is =  %f\n", f);
                      }
                    printf("\nConverting third INT8 to integer\n");

                /* Note that the decimal places will be truncated */

                    if (x= ifx_int8toflt(&num3, &f))
                        {
                        printf("\tError %d in converting INT8 to float\n", x);
                        exit(1);
                        }
                    else
                        {
                        printf("String3 = %s\n", string3);
                        printf("INT8 value is =  %f\n",f);
```

```
        }
    printf("\nIFX_INT8TOFLT Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
IFX_INT8TOFLT Sample ESQL Program running.


Converting INT8 to float

Executing: ifx_int8toflt(&num1,&f)
String 1= -12,555.765
The value of the first float is = -12555.000000


Converting second INT8 to float

Executing: ifx_int8toflt(&num2, &f)
String2 = 480.76
The value of the second float is =  480.000000


Converting third INT8 to integer

Executing: ifx_int8toflt(&num3, &f)
String3 = 5.2
The value of the third float is =  5.000000

IFX_INT8TOFLT Sample Program over.
```

The ifx_int8toint() function

The **ifx_int8toint()** function converts an **int8** type number into a C **int** type number.

Syntax:

```
mint ifx_int8toint(int8_val, int_val)
    ifx_int8_t *int8_val;
    mint *int_val;
```

int8_val   A pointer to an **int8** structure whose value **ifx_int8toint()** converts to an **mint** type value.

int_val   A pointer to an **mint** value where **ifx_int8toint()** places the result of the conversion.

Usage:

The **ifx_int8toint()** library function converts an **int8** value to a C integer. The size of a C integer depends upon the hardware and operating system of the computer you are using. Therefore, the **ifx_int8toint()** function equates an integer value with the SQL SMALLINT data type. The valid range of a SMALLINT is between 32767 and -32767. To convert larger **int8** values to larger integers, use the **ifx_int8tolong()** library function.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Examples:

The file **int8toint.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8toint.ec *

   The following program converts three strings to INT8 types and
   converts the INT8 type values to C integer type values.
   Then the values are displayed.

*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-12,555";
char string2[] = "480";
char string3[] = "5.2";

main()
{
    mint x;
    mint i =0;
    ifx_int8_t num1, num2, num3;

    printf("IFX_INT8TOINT Sample ESQL Program running.\n\n");
    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }
    printf("\nConverting INT8 to integer\n");
    if (x= ifx_int8toint(&num1, &i))
        {
        printf("\tError %d in converting INT8 to integer\n", x);
        exit(1);
        }
    else
        {
        printf("String 1= %s\n", string1);
        printf("INT8 value is = %d\n", i);
        }
printf("\nConverting second INT8 to integer\n");
    if (x= ifx_int8toint(&num2, &i))
        {
        printf("\tError %d in converting INT8 to integer\n", x);
        exit(1);
        }
    else
        {
```

```
                printf("String2 = %s\n", string2);
                printf("INT8 value is =  %d\n", i);
                }
        printf("\nConverting third INT8 to integer\n");

         /* note that the decimal will be truncated */

         if (x= ifx_int8toint(&num3, &i))
                {
                printf("\tError %d in converting INT8 to integer\n", x);
                exit(1);
                }
         else
                {
                printf("String3 = %s\n", string3);
                printf("INT8 value is =  %d\n",i);
                }
        printf("\nIFX_INT8TOINT Sample Program over.\n\n");
        exit(0);
}
```

Output:
```
IFX_INT8TOINT Sample ESQL Program running.


Converting INT8 to integer

Executing: ifx_int8toint(&num1,&i)
String 1= -12,555
The value of the first integer is = -12555


Converting second INT8 to integer

Executing: ifx_int8toint(&num2, &i)
String2 = 480
The value of the second integer is =  480


Converting third INT8 to integer

Executing: ifx_int8toint(&num3, &i)
String3 = 5.2
The value of the third integer is =  5

IFX_INT8TOINT Sample Program over.
```

The ifx_int8tolong() function

The **ifx_int8tolong()** function converts an **int8** type number into a C **long** type
number.

Syntax:
```
mint ifx_int8tolong(int8_val, lng_val)
   ifx_int8_t *int8_val;
   int4  *lng_val;
```

int8_val   A pointer to an **int8** structure whose value **ifx_int8tolong()**
                    converts to an **int4** integer type value.

lng_val   A pointer to an **int4** integer where **ifx_int8tolong()** places the result of the conversion.

Return codes:

0   The conversion was successful.

-1200   The magnitude of the **int8** type number is greater than 2,147,483,647.

Examples:

The file **int8tolong.ec** in the **demo** directory contains the following sample program.

```
/*
   * ifx_int8tolong.ec *

   The following program converts three strings to INT8 types and
   converts the INT8 type values to C long type values.
   Then the values are displayed.

*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-1,555,345,698";
char string2[] = "3,235,635";
char string3[] = "553.24";

main()
{
    int x;
    long l =0;
    ifx_int8_t num1, num2, num3;

    printf("IFX_INT8TOLONG Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }
    printf("\nConverting INT8 to long\n");
    if (x= ifx_int8tolong(&num1, &l))
        {
        printf("\tError %d in converting INT8 to long\n", x);
        exit(1);
        }
    else
        {
        printf("String 1= %s\n", string1);
```

```
            printf("INT8 value is = %d\n", l);
            }

    printf("\nConverting second INT8 to long\n");
    if (x= ifx_int8tolong(&num2, &l))
        {
        printf("\tError %d in converting INT8 to long\n", x);
        exit(1);
        }
    else
        {
        printf("String2 = %s\n", string2);
      printf("INT8 value is =  %d\n",l);
        }
    printf("\nConverting third INT8 to long\n");

/* Note that the decimal places will be truncated. */

    if (x= ifx_int8tolong(&num3, &l))
        {
        printf("\tError %d in converting INT8 to long\n", x);
        exit(1);
        }
    else
        {
        printf("String3 = %s\n", string3);
        printf("INT8 value is =  %d\n",l);
        }
    printf("\nIFX_INT8TOLONG Sample Program over.\n\n");
    exit(0);
}
```

Output:

```
IFX_INT8TOLONG Sample ESQL Program running.

Converting INT8 to long

Executing: ifx_int8tolong(&num1,&l)
String 1= -1,555,345,698
The value of the first long is = -1555345698


Converting second INT8 to long

Executing: ifx_int8tolong(&num2, &l)
String2 = 3,235,635
The value of the second long is =  3235635


Converting third INT8 to long

Executing: ifx_int8tolong(&num3, &l)
String3 = 553.24
The value of the third long is =  553

IFX_INT8TOLONG Sample Program over.
```

The ifx_lo_alter() function

The **ifx_lo_alter()** function alters the storage characteristics of an existing smart large object.

Syntax:

```
mint ifx_lo_alter(LO_ptr, LO_spec)
   ifx_lo_t *LO_ptr;
   ifx_lo_create_spec_t *LO_spec;
```

LO_ptr   A pointer to an LO-pointer structure that identifies the smart large object whose storage characteristics are altered. For more information about LO-pointer structures, see "The LO-Pointer Structure" on page 8-9.

LO_spec   A pointer to the LO-specification structure that contains the storage characteristics that **ifx_lo_alter()** saves for the smart large object that LO_*ptr* indicates. For more information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

Usage:

The **ifx_lo_alter()** function updates the storage characteristics of an existing smart large object with the characteristics in the LO-specification structure to which *LO_spec* points. With **ifx_lo_alter()**, you can change only the following storage characteristics:

- Logging characteristics

  You can set the LO_LOG or LO_NOLOG flag with the **ifx_lo_specget_flags()** function.

- Last-access time characteristics

  You can set the LO_KEEP_LASTACCESS_TIME or LO_NOKEEP_LASTACCESS_TIME flag with the **ifx_lo_specset_flags()** function.

- Extent size

  You can store a new integer value for the allocation extent size with the **ifx_lo_specset_extsz()** function. The new extent size applies only to extents written after the **ifx_lo_alter()** function completes.

The function obtains an exclusive lock for the entire smart large object before it proceeds with the update. It holds this lock until the update completes.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_col_info() function

The ifx_lo_create() function

The ifx_lo_def_create_spec() function

The ifx_lo_specset_extsz() function

The ifx_lo_close() function

The **ifx_lo_close()** function closes an open smart large object.

Syntax:
```
mint ifx_lo_close(LO_fd)
   mint LO_fd;
```

LO_fd   The LO file descriptor of the smart large object to close. For more
                 information about an LO file descriptor, see "The LO file
                 descriptor" on page 8-11.

Usage:

The **ifx_lo_close()** function closes the smart large object that is associated with the
LO file descriptor, *LO_fd*. The **ifx_lo_open()** and **ifx_lo_create()** functions return an
LO file descriptor when they successfully opens a smart large object.

When the **ifx_lo_close()** function closes a smart large object, the database server
attempts to unlock the smart large object. In some cases, the database server does
not permit the release of the lock until the end of the transaction. (If you do not
perform updates to smart large objects inside a BEGIN WORK transaction block,
every update is a separate transaction.) This behavior might occur if the isolation
mode is repeatable read or if the lock held is an exclusive lock.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the
                 cause of the failure.

Related functions:

The ifx_lo_create() function

The ifx_lo_open() function

Examples:

For an example of the **ifx_close()** function, see "The create_clob.ec program" on
page C-2.

The ifx_lo_col_info() function

The **ifx_lo_col_info()** function sets the fields of an LO-specification structure to the
column-level storage characteristics for a specified database column.

Syntax:
```
mint ifx_lo_col_info(column_name, LO_spec)
   char *column_name;
   ifx_lo_create_spec_t *LO_spec;
```

column_name   A pointer to a buffer that contains the name of the database
                 column whose column-level storage characteristics you wish to
                 use.

LO_spec	A pointer to the LO-specification structure in which to store the column-level storage characteristics for *column_name*. For more information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

Usage:

The **ifx_lo_col_info()** function sets the fields of the LO-specification structure to which *LO_spec* points, to the storage characteristics for the *column_name* database column. If this specified column does not have column-level storage characteristics defined for it, the database server uses the storage characteristics that are inherited. For more information about the inheritance hierarchy, see "Obtain storage characteristics" on page 8-6.

The *column_name* buffer must specify the column name in the following format:

```
database@server_name:table.column
```

If the column is in a database that is ANSI compliant, you can also include the *owner_name*, as follows:

```
database@server_name:owner.table.column
```

Important:: You must call the **ifx_lo_def_create_spec()** function before you call **ifx_lo_col_info()**.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_create() function

The ifx_lo_def_create_spec() function

Examples:

For an example of the **ifx_col_info()** function, see "The create_clob.ec program" on page C-2.

The ifx_lo_copy_to_file() function

The **ifx_lo_copy_to_file()** function copies the contents of a smart large object into an operating-system file.

Syntax:

```
mint ifx_lo_copy_to_file(LO_ptr, fname, flags, result)
   ifx_lo_t *LO_ptr;
   char *fname;
   mint flags;
   char *result;
```

LO_ptr	A pointer to the LO-pointer structure that you provide to identify the smart large object to copy. For more information about LO-pointer structures, see "The LO-Pointer Structure" on page 8-9.

fname	The full path name of the target file to hold the data.
flags	An integer that specifies the location of the fname file.
result	A pointer to a buffer that contains the file name that **ifx_lo_copy_to_file()** generates.

Usage:

The **ifx_lo_copy_to_file()** function can create the target files on either the server or the client computer. The flag values for the *flags* argument indicate the location of the file to copy. Valid values include the following constants, which the **locator.h** header file defines.

File-location constant	Purpose
LO_CLIENT_FILE	The *fname* file is on the client computer.
LO_SERVER_FILE	The *fname* file is on the server computer.

By default, the **ifx_lo_copy_to_file()** function generates a file name of the form:

`fname.hex_id`

In this format, *fname* is the file name you specify as an argument to **ifx_lo_copy_to_file()** and *hex_id* is the unique hexadecimal smart-large-object identifier. The maximum number of digits for a smart-large-object identifier is 17; however most smart large objects would have an identifier with significantly fewer digits.

For example, suppose you specify a *pathname* value as '/tmp/resume'.

If the CLOB column has an identifier of **203b2**, the **ifx_lo_copy_to_file()** function creates the file: **/tmp/resume.203b2**.

To change this default file name, you can specify the following wildcards in the file name portion of *fname*:

- One or more contiguous question mark (?) characters in the file name can generate a unique file name.

  The **ifx_lo_copy_to_file()** function replaces each question mark with a hexadecimal digit from the identifier of the BLOB or CLOB column. For example, suppose you specify a *pathname* value as '/tmp/resume??.txt'.

  The **ifx_lo_copy_to_file()** function puts 2 digits of the hexadecimal identifier into the name. If the CLOB column has an identifier of **203b2**, the **ifx_lo_copy_to_file()** function would create the file **/tmp/resumeb2.txt**.

  If you specify more than 17 question marks, the **ifx_lo_copy_to_file()** function ignores them.

- An exclamation point (!) at the end of the file name indicates that the file name does not need to be unique.

  For example, suppose you specify a path name value as '/tmp/resume.txt!'.

  The **ifx_lo_copy_to_file()** function does not use the smart-large-object identifier in the file name so it generates the following file: **/tmp/resume.txt**

  The exclamation point overrides the question marks in the file name specification.

Tip: These wildcards are also valid in the *fname* argument of the **ifx_lo_filename()** function. For more information about **ifx_lo_filename()**, see B-113.

Your application must ensure that there is sufficient space to hold the generated file.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_copy_to_lo() function

The ifx_lo_filename() function

The ifx_lo_copy_to_lo() function

The **ifx_lo_copy_to_lo()** function copies the contents of a file into an open smart large object.

Syntax:
```
mint ifx_lo_copy_to_lo(LO_fd, fname, flags)
   mint LO_fd;
   char *fname;
   mint flags;
```

LO_fd   The LO file descriptor for the open smart large object in which to write the file contents. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.

fname   The full path name of the source file that contains the data to copy.

flags   An integer that specifies the location of the fname file.

Usage:

The **ifx_lo_copy_to_lo()** function can copy the contents of a source file on either the server or the client computer. The flag values for the *flags* argument indicate the location of the file to copy. Valid values include the following constants, which the **locator.h** header file defines.

File-location constant	Purpose
LO_CLIENT_FILE	The *fname* file is on the client computer.
LO_SERVER_FILE	The *fname* file is on the server computer.
LO_APPEND	Append the data in *fname* to the end of the specified smart large object. This flag can be masked with one of the preceding flags.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_copy_to_file() function

The ifx_lo_create() function

The **ifx_lo_create()** function creates a new smart large object and opens it for access within an Informix ESQL/C program.

Syntax:

```
mint ifx_lo_create(LO_spec, flags, LO_ptr, error)
   ifx_lo_create_spec_t *LO_spec;
   mint flags;
   ifx_lo_t *LO_ptr;
   mint *error;
```

LO_spec	A pointer to the LO-specification structure that contains the storage characteristics for new smart large objects. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.
flags	An integer that specifies the mode in which to open the new smart large object. For more information, see "Access Modes" on page 8-13.
LO_ptr	A pointer to the LO-pointer structure for the new smart large object. For more information about LO-pointer structures, see "The LO-Pointer Structure" on page 8-9.
error	A pointer to an integer that contains the error code that **ifx_lo_create()** sets.

Usage:

The **ifx_lo_create()** function performs the following steps to create a new smart large object:

1. It creates a new LO-pointer structure and assigns a pointer to this structure to the *LO_ptr* argument.
2. It assigns the storage characteristics for the smart large object from the LO-specification structure, *LO_spec*.

   If the LO-specification structure does not contain storage characteristics (the associated fields are null), **ifx_lo_create()** uses the storage characteristics from the inheritance hierarchy for the new smart large object. The **ifx_lo_create()** function also uses the system-specified storage characteristics if the *LO_spec* pointer is null.

   For more information about the inheritance hierarchy, see "Obtain storage characteristics" on page 8-6.
3. It opens the new smart large object in the access mode that the *flags* argument specifies.

   The flag values for the *flags* argument indicate the mode of the smart large object after **ifx_lo_create()** successfully completes. Valid values include all access-mode constants, which Table 8-3 on page 8-14 shows. For more information about access modes, see "Opening a Smart Large Object" on page 8-13.
4. It returns an LO file descriptor that identifies the open smart large object.

Important: You must call the **ifx_lo_def_create_spec()** function to initialize an LO-specification structure before you call the **ifx_lo_create()** function.

IBM Informix uses the default parameters that the call to **ifx_lo_create()** establishes to determine whether subsequent operations result in locking and/or logging of the corresponding smart large object. For more information, see "Lightweight I/O" on page 8-15

Each **ifx_lo_create()** call is implicitly associated with the current connection. When this connection closes, the database server deallocates any smart large objects that are not referenced by any columns (those with a reference count of zero).

If the **ifx_lo_create()** function is successful, it returns a valid LO-file descriptor (*LO_fd*). You can then use the *LO_fd* to identify which smart large object to access in subsequent function calls such as **ifx_lo_read()** and **ifx_lo_write()**. However, a *LO_fd* is only valid within the current database connection.

Return codes:

A valid LO file descriptor	The function successfully created and opened the new smart large object.
-1	The function was not successful; examine the error for a detailed error code.

Related functions:

The ifx_lo_close() function

The ifx_lo_col_info() function

The ifx_lo_def_create_spec() function

Examples:

For an example of the **ifx_lo_create()** function, see "The create_clob.ec program" on page C-2.

The ifx_lo_def_create_spec() function

The **ifx_lo_def_create_spec()** function allocates and initializes an LO-specification structure.

Syntax:
```
mint ifx_lo_def_create_spec(LO_spec)
   ifx_lo_create_spec_t **LO_spec;
```

LO_spec   A pointer that points to a pointer to a new LO-specification structure that contains initialized fields. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

Usage:

The **ifx_lo_def_create_spec()** function creates and initializes a new LO-specification structure, **ifx_lo_create_spec_t**. The **ifx_lo_def_create_spec()** function initializes the new **ifx_lo_create_spec_t** structure with the appropriate null values and places its address in the *LO_spec* pointer. At the time the database server stores the large object, the database server interprets the null values to mean that system-specified

defaults should be used for the storage characteristics. For more information, see "The system-specified storage characteristics" on page 8-7.

Because the **ifx_lo_def_create_spec()** function allocates memory for the **ifx_lo_create_spec_t** structure, you must call the **ifx_lo_spec_free()** function to free that memory when you are finished using the structure.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_spec_free() function

The ifx_lo_col_info() function

The ifx_lo_create() function

Examples:

For an example of **ifx_lo_def_create_spec()**, see the "The create_clob.ec program" on page C-2.

The ifx_lo_filename() function

The **ifx_lo_filename()** function returns the path name that the database server would use if you copied a smart large object to a file.

Syntax:

```
mint ifx_lo_filename(LO_ptr, fname, result, result_buf_nbytes)
   ifx_lo_t *LO_ptr;
   char *fname;
   char *result;
   mint result_buf_nbytes;
```

LO_ptr	A pointer to the LO-pointer structure that identifies the smart large object to copy. For more information about LO-pointer structures, see "The LO-Pointer Structure" on page 8-9
fname	The full path name of the target file to hold the data.
result	A pointer to a buffer that contains the file name that **ifx_lo_copy_to_file()** would generate.
result_len	The size, in bytes, of the *result* character buffer.

Usage:

The **ifx_lo_filename()** function generates a file name from the *fname* argument that you provide. Use the **ifx_lo_filename()** function to determine the file name that the **ifx_lo_copy_to_file()** function would create for its *fname* argument.

By default, the **ifx_lo_copy_to_file()** function generates a file name of the form:
fname.hex_id

However, you can specify wildcards in the *fname* argument that can change this default file name. You can use these wildcards in the *fname* argument of **ifx_lo_filename()** to see what file name these wildcards generate. For more information about the wildcards that are valid in the *fname* argument, see the description of B-108.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_copy_to_file() function

The ifx_lo_from_buffer() function

The **ifx_lo_from_buffer()** function copies a specified number of bytes from a user-defined buffer into a smart large object.

Syntax:
```
mint ifx_lo_from_buffer(LO_ptr, size, buffer, error)
   ifx_lo_t *LO_ptr;
   mint size;
   char *buffer;
   mint *error;
```

LO_ptr   The LO-pointer structure for the smart large object into which you want to copy the data.

size   A **mint** that identifies the number of bytes to copy to the smart large object.

buffer   A pointer to a user-defined buffer from which you want to copy the data.

error   Contains the address of the **mint** that holds the error code that **ifx_lo_from_buffer()** sets

Usage:

The **ifx_lo_from_buffer()** function copies bytes, up to the size specified by *size*, from the user-defined buffer into the smart large object that the *LO_ptr* argument identifies. The write operation to the smart large object starts at a zero-byte offset. To use the **ifx_lo_from_buffer()** function, the smart large object must already exist in an sbspace before you copy the data.

Return codes:

0   The function was successful.

-1   The function was not successful.

The ifx_lo_lock() function

The **ifx_lo_lock()** function allows you to lock an explicit range of bytes in a smart large object.

Syntax:

```
mint ifx_lo_lock(LO_fd, offset, whence, range, lockmode)
   mint LO_fd;
   int8 *offset;
   mint whence;

   int8 *range;

   mint lockmode;
```

LO_fd	The LO-file descriptor for the smart large object in which to lock the range of bytes. For more information about an LO-file descriptor, see "The LO file descriptor" on page 8-11
offset	A pointer to the 8-byte integer (INT8) that specifies the offset within the smart large object at which the lock begins.
whence	A **mint** constant that specifies from what point the offset is calculated: the beginning of the smart large object, the current position within the smart large object, or the end of the smart large object.
range	A pointer to the 8-byte integer (INT8) that specifies the number of bytes to lock.
lockmode	The mode in which to lock the specified bytes. Set to LO_EXCLUSIVE_MODE for an exclusive lock or to LO_SHARED_MODE for a shared lock.

Usage:

The **ifx_lo_lock()** function locks the number of bytes specified by *range,* beginning at the location specified by *offset* and *whence*, for the smart large object that *LO_fd* specifies. The **ifx_lo_lock()** function places the type of lock that *lockmode* specifies. If you specify ISSLOCK, **ifx_lo_lock()** places a shared lock on the byte range. If you specify ISXLOCK, **ifx_lo_lock()** places an exclusive lock on the byte range.

Before you call **ifx_lo_lock()**, you must obtain a valid LO-file descriptor by calling either **ifx_lo_create()** to create a new smart large object, or by calling **ifx_lo_open()** to open an existing smart large object. For more information about *LO_fd*, see "The LO file descriptor" on page 8-11.

The **ifx_lo_lock()** function uses the *whence* and *offset* arguments to determine the seek position, as follows:

- The *whence* value identifies the position from which to start the seek.

   Valid values include the following constants, which the **locator.h** header file defines.

Whence constant	Starting seek position
LO_SEEK_SET	The start of the smart large object
LO_SEEK_CUR	The current seek position in the smart large object
LO_SEEK_END	The end of the smart large object

– The *offset* argument identifies the offset, in bytes, from the starting seek position (that the *whence* argument specifies) at which to begin locking bytes.

In addition to locking *nbytes*, you can also lock bytes from a specified offset to the end of the large object, which you can specify as either the current end or the maximum end of the large object. You can use two integer constants (LO_CURRENT_END and LO_MAX_END) to denote these values. To use one of these values, first convert it to an **int8** value and then use it for the *nbytes* argument.

Return codes:

0	The function was successful
< 0	The function was unsuccessful. The value returned is the **sqlcode**, which is the number of the IBM Informix error message. For more information about **sqlcode**, see Chapter 11, "Exception handling"

Related function:

The ifx_lo_unlock() function

The ifx_lo_open() function

The **ifx_lo_open()** function opens an existing smart large object for access.

Syntax:
```
mint ifx_lo_open(LO_ptr, flags, error)
    ifx_lo_t *LO_ptr;
    mint flags;
    mint *error;
```

LO_ptr	A pointer to the LO-pointer structure that identifies the smart large object to open. For more information about LO-pointer structures, see "The LO-Pointer Structure" on page 8-9.
flags	A **mint** that specifies the mode in which to open the smart large object that *LO_ptr* identifies. For more information, see "Access Modes" on page 8-13.
error	A pointer to a **mint** that contains the error code that **ifx_lo_open()** sets.

Usage:

Your IBM Informix ESQL/C program must call the **ifx_lo_open()** function for each instance of a smart large object that it needs to access.

The value of the *flags* argument indicates the mode of the smart large object after **ifx_lo_open()** successfully completes. For a description of valid values for the *flags* argument, see Table 8-3 on page 8-14.

IBM Informix uses the default parameters that **ifx_lo_open()** (or **ifx_lo_create()**) establishes to determine whether subsequent operations cause locking or logging to occur for the smart large object. For more information about the settings that affect the opening of a smart large object, see "Opening a Smart Large Object" on page 8-13.

Each **ifx_lo_open()** call is implicitly associated with the current connection. When this connection closes, the database server deallocates any smart large objects that are not referenced by any columns (those with a reference count of zero).

If the **ifx_lo_open()** function is successful, it returns a valid LO file descriptor (*LO_fd*). You can then use the file descriptor to identify which smart large object to access in subsequent function calls such as **ifx_lo_read()** and **ifx_lo_write()**. A *LO_fd* is valid only within the current database connection.

Once **ifx_lo_open()** has opened the smart large object, it sets the seek position in the returned LO file descriptor to byte zero (0). If the default range for locking is set for locking the entire smart large object, the **ifx_lo_open()** function can also obtain a lock on the smart large object, based on the following settings for the access mode:

- For dirty-read mode, the database server does not place a lock on the smart large object.
- For read-only mode, the database server obtains a shared lock on the smart large object.
- For write-only, write-append, or read-write mode, the database server obtains an update lock on the smart large object. When a call to the **ifx_lo_write()** or **ifx_lo_writewithseek()** function occurs, the database server promotes the lock to an exclusive lock.

The lock that **ifx_lo_open()** obtains is lost when the current transaction terminates. The database server obtains the lock again, however, when the next function that needs a lock executes. If this behavior is undesirable, use BEGIN WORK transaction blocks and place a COMMIT WORK or ROLLBACK WORK statement after the last statement that needs to use the lock.

Return codes:

-1	The function was not successful; examine the error for a detailed error code.
A valid LO file descriptor	The function has successfully opened the smart large object and returned a valid LO file descriptor.

Related functions:

The ifx_lo_close() function

The ifx_lo_create() function

The ifx_lo_read() function

The ifx_lo_write() function

Example:

For an example of the **ifx_lo_open()** function, see "The create_clob.ec program" on page C-2.

The ifx_lo_read() function

The **ifx_lo_read()** function reads a specified number of bytes of data from an open smart large object.

Syntax:

```
mint ifx_lo_read(LO_fd, buf, nbytes, error)
   mint LO_fd;
   char *buf;
   mint nbytes;
   mint *error;
```

LO_fd	The LO file descriptor for the smart large object from which to read. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.
buf	A pointer to a character buffer that contains the data that **ifx_lo_read()** reads from the smart large object.
nbytes	The size, in bytes, of the buf character buffer. This value cannot exceed 2 GB.
error	A pointer to a **mint** that contains the error code that **ifx_lo_read()** sets.

Usage:

The **ifx_lo_read()** function reads *nbytes* of data from the open smart large object that the *LO_fd* file descriptor identifies. The read begins at the current seek position for *LO_fd*. You can use the **ifx_lo_tell()** function to obtain the current seek position.

The function reads this data into the user-defined buffer to which *buf* points. The *buf* buffer must be less than 2 GB in size. To read smart large objects that are larger than 2 GB, read them in 2-GB chunks.

Return codes:

>=0	The number of bytes that the function has read from the smart large object into the *buf* character buffer.
-1	The function was not successful; examine the *error* for a detailed error code.

Related functions:

The ifx_lo_open() function

The ifx_lo_readwithseek() function

The ifx_lo_seek() function

The ifx_lo_tell() function

The ifx_lo_write() function

Example:

For an example of the **ifx_lo_read()** function, see "The upd_lo_descr.ec program" on page C-8.

The ifx_lo_readwithseek() function

The **ifx_lo_readwithseek()** function performs a seek operation and then reads a specified number of bytes of data from an open smart large object.

Syntax:

```
mint ifx_lo_readwithseek(LO_fd, buf, nbytes, offset, whence, error)
   char *buf;
   mint nbytes;
   ifx_int8_t *offset;
   mint whence;
   mint *error;
```

LO_fd	The LO file descriptor for the smart large object from which to read. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.
buf	A pointer to a character buffer that contains the data that **ifx_lo_readwithseek()** reads from the smart large object.
nbytes	The size, in bytes, of the buf character buffer. This value cannot exceed 2 gigabytes.
offset	A pointer to the 8-byte integer (INT8) offset from the starting seek position.
whence	A mint value that identifies the starting seek position.
error	A pointer to a mint that contains the error code that **ifx_lo_readwithseek()** sets.

Usage:

The **ifx_lo_readwithseek()** function reads *nbytes* of data from the open smart large object that the *LO_fd* file descriptor identifies.

The read begins at the seek position of *LO_fd* that the *offset* and *whence* arguments indicate, as follows:

- The *whence* argument identifies the position from which to start the seek.

  Valid values include the following constants, which the **locator.h** header file defines.

Whence constant	Starting seek position
LO_SEEK_SET	The start of the smart large object
LO_SEEK_CUR	The current seek position in the smart large object
LO_SEEK_END	The end of the smart large object

  - The *offset* argument identifies the offset, in bytes, from the starting seek position (that the *whence* argument specifies) to which the seek position should be set.

    For more information about how to access eight-bit (INT8) integers, see "The int8 Data Type" on page 5-2.

The function reads this data into the user-defined buffer to which *buf* points. The size of the *buf* buffer must be less than 2 GB. To read smart large objects that are larger than 2 GB, read them in 2-GB chunks.

Return codes:

>=0	The number of bytes that the function has read from the smart large object into the *buf* character buffer.
-1	The function was not successful; examine the *error* for a detailed error code.

Related functions:

The ifx_lo_open() function

The ifx_lo_read() function

The ifx_lo_seek() function

The ifx_lo_write() function

Example:

For an example of the **ifx_lo_readwithseek()** function, see "The create_clob.ec program" on page C-2.

The ifx_lo_release() function

The **ifx_lo_release()** function tells the database server to release the resources associated with a temporary smart large object.

Syntax:
```
mint ifx_lo_release(LO_ptr)
   ifx_lo_t *LO_ptr;
```

LO_ptr	The LO-pointer structure for the smart large object for which you want to release resources.

Usage:

The **ifx_lo_release()** function is useful for telling the database server when it is safe to release resources associated with temporary smart large objects. A *temporary* smart large object is one that has one or more LO handles, none of which have been inserted into a table. Temporary smart large objects can occur in the following ways:

- You create a smart large object with **ifx_lo_create()** but do not insert its LO handle into a column of the database.
- You invoke a user-defined routine that creates a smart large object in a query but never assigns its LO handle to a column of the database.

For example, the following query creates one smart large object for each row in the **table1** table and sends each one to the client application:
```
SELECT filetoblob(...) FROM table1;
```

The client application can use the **ifx_lo_release()** function to indicate to the database server when it finishes processing each of these smart large objects. Once

you call this function on a temporary smart large object, the database server can release the resources at any time. Further use of the LO handle and any associated LO file descriptors is not guaranteed to work.

Use of this function on smart large objects that are not temporary does not cause any incorrect behavior. However, the call is expensive and is not needed for permanent smart large objects.

Return codes:

0	The function was successful.
< 0	The function was not successful.

The ifx_lo_seek() function

The **ifx_lo_seek()** function sets the file position for the next read or write operation on the open smart large object.

Syntax:
```
mint ifx_lo_seek(LO_fd, offset, whence, seek_pos)
   mint LO_fd;
   ifx_int8_t *offset;
   mint whence;
   ifx_int8_t *seek_pos;
```

LO_fd	The LO file descriptor for the smart large object whose seek position you wish to change. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.
offset	A pointer to the 8-byte integer offset from the starting seek position.
whence	A **mint** value that identifies the starting seek position.
seek_pos	A pointer to the resultant 8-byte integer offset, relative to the start of the file, that corresponds to the position for the next read/write operation.

Usage:

The function uses the *whence* and *offset* arguments to determine the seek position, as follows:

- The *whence* value identifies the position from which to start the seek.

    Valid values include the following constants, which the **locator.h** header file defines.

Whence constant	Starting seek position
LO_SEEK_SET	The start of the smart large object
LO_SEEK_CUR	The current seek position in the smart large object
LO_SEEK_END	The end of the smart large object
	– The *offset* argument identifies the offset, in bytes, from the starting seek position (that the *whence* argument specifies) at which to begin the seek position.

The **ifx_lo_tell()** function returns the current seek position for an open smart large object.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_read() function

The ifx_lo_tell() function

The ifx_lo_write() function

Example:

For an example of the **ifx_lo_seek()** function, see "The ifx_lo_read() function" on page B-118.

The ifx_lo_spec_free() function

The **ifx_lo_spec_free()** function frees the resources of an LO-specification structure.

Syntax:
```
mint ifx_lo_spec_free(LO_spec)
   ifx_lo_create_spec_t *LO_spec;
```

LO_spec   A pointer to the LO-specification structure to free. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

Usage:

The **ifx_lo_spec_free()** function frees a LO-specification structure that was allocated by a call to **ifx_lo_def_create_spec()**. The *LO_spec* pointer points to the **ifx_lo_create_spec_t** structure which is to be freed.

IBM Informix ESQL/C does not perform memory management for a LO-specification structure. You must call **ifx_lo_spec_free()** for each LO-specification structure that you allocate with a call to the **ifx_lo_def_create_spec()** function.

Restriction: Do not use **ifx_lo_spec_free()** to free an **ifx_lo_create_spec_t** structure that you accessed through a call to the **ifx_lo_stat_cspec()** function. When you call **ifx_lo_stat_free()** to free the **ifx_lo_stat_t** structure, it also automatically frees the **ifx_lo_create_spec_t** structure. Use **ifx_lo_spec_free()** only to free an **ifx_lo_create_spec_t** structure that you created through a call to **ifx_lo_def_create_spec()**.

Return codes:

0   The function was successful.

| <0 | The function was not successful and the return value indicates the cause of the failure. |

Related function:

The ifx_lo_def_create_spec() function

Example:

For an example of the use of the **ifx_lo_spec_free()** function, see the "The create_clob.ec program" on page C-2.

The ifx_lo_specget_def_open_flags() function

The **ifx_lo_specget_def_open_flags()** function obtains the default open flags of a smart large object from an LO-specification structure.

Syntax:
```
mint ifx_lo_specget_def_open_flags(LO_spec)

    ifx_lo_create_spec_t *LO_spec;
```

| *LO_spec* | A pointer to the LO-specification structure from which to obtain the default open flags. |

Usage:

This function can be used to obtain the default open flags from a LO-specification structure. It can be used in conjunction with **ifx_lo_stat_cspec()** to obtain the default open flags that were specified when an existing smart large object was created.

Return codes:

| >=0 | The function was successful. The returned integer stores the values of the default open flags. |
| -1 | The function was unsuccessful |

Example:

For an example of the **ifx_lo_specget_def_open_flags()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_specget_estbytes() function

The **ifx_lo_specget_estbytes()** function obtains from an LO-specification structure the estimated size of a smart large object.

Syntax:
```
mint ifx_lo_specget_estbytes(LO_spec, estbytes)
    ifx_lo_create_spec_t *LO_spec;
    ifx_int8_t *estbytes;
```

| *LO_spec* | A pointer to the LO-specification structure from which to obtain the estimated size. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3. |

estbytes   A pointer to an **ifx_int8_t** structure into which
                  ifx_lo_specget_estbytes() puts the estimated number of bytes for
                  the smart large object.

Usage:

The *estbytes* value is the estimated final size, in bytes, of the smart large object.
This estimate is an optimization hint for the smart-large-object optimizer. For more
information about the estimated size, see Table 8-1 on page 8-4.

Important: You must call the **ifx_lo_def_create_spec()** function to initialize an
               LO-specification structure before you call **ifx_lo_specget_estbytes()**.
               You can use the **ifx_lo_col_info()** function to store storage
               characteristics that are associated with a particular column in an
               LO-specification structure.

Return codes:

0   The function was successful.

-1   The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_spec_free() function

The ifx_lo_specset_extsz() function

The ifx_lo_specset_flags() function

The ifx_lo_specget_maxbytes() function

The ifx_lo_specget_sbspace() function

The ifx_lo_specget_estbytes() function

Example:

For an example of the **ifx_lo_specget_estbytes()** function, see "The get_lo_info.ec
program" on page C-5.

The ifx_lo_specget_extsz() function

The **ifx_lo_specget_extsz()** function obtains from an LO-specification structure the
allocation extent size of a smart large object.

Syntax:
```
mint ifx_lo_specget_extsz(LO_spec)
    ifx_lo_create_spec_t *LO_spec;
```

LO_spec   A pointer to the LO-specification structure from which to obtain
                  the extent size. For information about the LO-specification
                  structure, see "The LO-specification structure" on page 8-3.

Usage:

The *extsz* value specifies the size, in bytes, of the allocation extents to be allocated for the smart large object when the database server writes beyond the end of the current extent. This value overrides the estimate that IBM Informix generates for how large an extent should be. For more information about the allocation extent, see Table 8-1 on page 8-4.

Important: You must call the **ifx_lo_def_create_spec()** function to initialize an LO-specification structure before you call **ifx_lo_specget_extsz()**. You can use the **ifx_lo_col_info()** function to store storage characteristics that are associated with a particular column in an LO-specification structure.

Return codes:

>=0   The function was successful and the return value indicates the extent size.

-1   The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_spec_free() function

The ifx_lo_specset_estbytes() function

The ifx_lo_specget_flags() function

The ifx_lo_specget_maxbytes() function

The ifx_lo_specget_sbspace() function

The ifx_lo_specset_extsz() function

Example:

For an example of the **ifx_lo_specget_extsz()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_specget_flags() function

The **ifx_lo_specget_flags()** function obtains from an LO-specification structure the create-time flags of a smart large object.

Syntax:
```
mint ifx_lo_specget_flags(LO_spec)
   ifx_lo_create_spec_t *LO_spec;
```

LO_spec   A pointer to the LO-specification structure from which to obtain the flag value. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

Usage:

The create-time flags provide the following information about a smart large object:

• Whether to use logging on the smart large object

- Whether to store the time of last access for the smart large object

These two indicators are masked together into the single flags value. For more information about the create-time flags, see Table 8-2 on page 8-5.

Important: You must call the **ifx_lo_def_create_spec()** function to initialize an LO-specification structure before you call **ifx_lo_specget_flags()**. You can use the **ifx_lo_col_info()** function to store storage characteristics that are associated with a particular column in an LO-specification structure.

Return codes:

>=0   The function was successful and the return value is the value of the create-time flags.

-1   The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_spec_free() function

The ifx_lo_specget_estbytes() function

The ifx_lo_specget_extsz() function

The ifx_lo_specget_maxbytes() function

The ifx_lo_specget_sbspace() function

The ifx_lo_specset_flags() function

Example:

For an example of the **ifx_lo_specget_flags()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_specget_maxbytes() function

The **ifx_lo_specget_maxbytes()** function obtains from an LO-specification structure the maximum size of a smart large object.

Syntax:

```
mint ifx_lo_specget_maxbytes(LO_spec, maxbytes)
   ifx_lo_create_spec_t *LO_spec;
   ifx_int8_t *maxbytes;
```

LO_spec   A pointer to the LO-specification structure from which to obtain the maximum size. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3

maxbytes   A pointer to an **int8** value into which **ifx_lo_specget_maxbytes()** puts the maximum size, in bytes, of the smart large object. If this value is **-1**, the smart large object has no size limit.

Usage:

IBM Informix does not allow the size of a smart large object to exceed the *maxbytes* value. For more information about the maximum size, see Table 8-1 on page 8-4.

Important: You must call the **ifx_lo_def_create_spec()** function to initialize an LO-specification structure before you call **ifx_lo_specget_maxbytes()**. You can use the **ifx_lo_col_info()** function to store storage characteristics that are associated with a particular column in an LO-specification structure.

Return codes:

0   The function was successful.

-1   The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_spec_free() function

The ifx_lo_specget_estbytes() function

The ifx_lo_specget_extsz() function

The ifx_lo_specget_flags() function

The ifx_lo_specget_sbspace() function

The ifx_lo_specset_maxbytes() function

Example:

For an example of the **ifx_lo_specget_maxbytes()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_specget_sbspace() function

The **ifx_lo_specget_sbspace()** function obtains from an LO-specification structure the name of an sbspace where a smart large object is stored.

Syntax:
```
mint ifx_lo_specget_sbspace(LO_spec, sbspace_name, length)
   ifx_lo_create_spec_t *LO_spec;
   char *sbspace_name;
   mint length;
```

LO_spec   A pointer to the LO-specification structure from which to obtain the sbspace name. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

sbspace_name   A character buffer into which **ifx_lo_specget_sbspace()** puts the name of the sbspace for the smart large object.

length   A mint value that specifies the size, in bytes, of the *sbspace_name* buffer.

Usage:

The **ifx_lo_specget_sbspace()** function returns the current setting for the name of the sbspace in which to store the smart large object. The function copies up to (*length*-1) bytes into the *sbspace_name* buffer and ensures that it is null terminated. For more information about an sbspace name, see Table 8-1 on page 8-4.

Important: You must call the **ifx_lo_def_create_spec()** function to initialize an LO-specification structure before you call **ifx_lo_specget_sbspaces()**. You can use the **ifx_lo_col_info()** function to store storage characteristics that are associated with a particular column in an LO-specification structure.

An sbspace name can be up to 18 characters long. However, you might want to allocate at least 129 bytes for the *sbspace_name* buffer to accommodate future increases in the length of an sbspace name.

Return codes:

0	The function was successful.
-1	The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_spec_free() function

The ifx_lo_specget_estbytes() function

The ifx_lo_specget_extsz() function

The ifx_lo_specget_flags() function

The ifx_lo_specget_maxbytes() function

The ifx_lo_specset_sbspace() function

Example:

For an example of the **ifx_lo_specget_sbspace()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_specset_def_open_flags() function

The **ifx_lo_specset_def_open_flags()** function sets the default open flags for a smart large object.

Syntax:
```
mint ifx_lo_specset_def_open_flags(LO_spec, flags)

   ifx_lo_create_spec_t *LO_spec;

   mint flags;
```

LO_spec	A pointer to the LO-specification structure in which the default open flags are to be set.
flags	A mint value for the default open flags of the smart large object.

Usage:

The most common use of this function is to specify that the smart large object always is to be opened using unbuffered I/O. This function can also be used to supply any required default open flags for a smart large object. The supplied flags are used whenever the smart large object is subsequently opened, unless explicitly overridden at open time.

Return codes:

0	The function was successful
-1	The function was unsuccessful

Example:
```
/* use unbuffered I/O on all opens for this LO */

ret = ifx_lo_specset_def_open_flags(lospec, LO_NOBUFFER);
```

The ifx_lo_specset_estbytes() function

The **ifx_lo_specset_estbytes()** function sets the estimated size of a smart large object.

Syntax:
```
mint ifx_lo_specset_estbytes(LO_spec, estbytes)
   ifx_lo_create_spec_t *LO_spec;
   ifx_int8_t *estbytes;
```

LO_spec	A pointer to the LO-specification structure in which to save the estimated size. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3
estbytes	A pointer to an **ifx_int8_t** structure that contains the estimated number of bytes for the smart large object.

Usage:

The *estbytes* value is the estimated final size, in bytes, of the smart large object. This estimate is an optimization hint for the smart-large-object optimizer. For more information about the estimated byte size, see Table 8-1 on page 8-4.

If you do not specify an *estbytes* value when you create a new smart large object, IBM Informix obtains the value from the inheritance hierarchy of storage characteristics. For more information about the inheritance hierarchy, see "Obtain storage characteristics" on page 8-6.

Do not change this system value unless you know the estimated size for the smart large object. If you do set the estimated size for a smart large object, do not specify a value much higher than the final size of the smart large object. Otherwise, the database server might allocate unused storage.

Return codes:

0	The function was successful.
-1	The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_specget_estbytes() function

The ifx_lo_specset_extsz() function

The ifx_lo_specset_flags() function

The ifx_lo_specset_maxbytes() function

The ifx_lo_specset_sbspace() function

The ifx_lo_spec_free() function

Example:

For an example of the **ifx_lo_specset_estbytes()** function, see "The create_clob.ec program" on page C-2.

The ifx_lo_specset_extsz() function

The **ifx_lo_specset_extsz()** function sets the allocation extent size for a smart large object.

Syntax:
```
mint ifx_lo_specset_extsz(LO_spec, extsz)
   ifx_lo_create_spec_t *LO_spec;
   mint extsz;
```

LO_spec	A pointer to the LO-specification structure in which to save the extent size. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.
extsz	An integer value for the size of the allocation extent of a smart large object.

Usage:

The *extsz* value specifies the size of the allocation extents to be allocated for the smart large object when the database server writes beyond the end of the current extent. This value overrides the estimate that IBM Informix generates for how large an extent should be. For more information about the allocation extent, see Table 8-1 on page 8-4.

If you do not specify an *extsz* value when you create a new smart large object, IBM Informix attempts to optimize the extent size based on past operations on the smart large object and other storage characteristics (such as maximum bytes) that it obtains from the inheritance hierarchy of storage characteristics. For more information about the inheritance hierarchy, see "Obtain storage characteristics" on page 8-6.

Do not change this system value unless you know the allocation extent size for the smart large object. Only applications that encounter severe storage fragmentation should ever set the allocation extent size. For such applications, make sure you know exactly the number of bytes by which to extend the smart large object.

Return codes:

| 0 | The function was successful. |
| -1 | The function was not successful. |

The ifx_lo_specset_flags() function

The **ifx_lo_specset_flags()** function sets the create-time flags of a smart large object.

Syntax:
```
mint ifx_lo_specset_flags(LO_spec, flags)
   ifx_lo_create_spec_t *LO_spec;
   mint flags;
```

LO_spec   A pointer to the LO-specification structure in which to save the flags value. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3

flags   An integer value for the create-time flags of the smart large object.

Usage:

The create-time flags provide the following information about a smart large object:
- Whether to use logging on the smart large object
- Whether to store the time of last access for the smart large object

These two indicators are masked together into the single flags value. For more information about the create-time flags, see Table 8-2 on page 8-5.

If you do not specify a *flags* value when you create a new smart large object, IBM Informix obtains the value from the inheritance hierarchy of storage characteristics. For more information about the inheritance hierarchy, see "Obtain storage characteristics" on page 8-6.

Return codes:

| 0 | The function was successful. |
| -1 | The function was not successful. |

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_spec_free() function

The ifx_lo_specget_flags() function

The ifx_lo_specset_extsz() function

The ifx_lo_specset_flags() function

The ifx_lo_specset_maxbytes() function

The ifx_lo_specset_sbspace() function

Example:

For an example of the **ifx_lo_specset_flags()** function, see "The create_clob.ec program" on page C-2.

The ifx_lo_specset_maxbytes() function

The **ifx_lo_specset_maxbytes()** function sets the maximum size for a smart large object.

Syntax:
```
mint ifx_lo_specset_maxbytes(LO_spec, maxbytes)
   ifx_lo_create_spec_t *LO_spec;
   ifx_int8_t *maxbytes;
```

LO_spec   A pointer to the LO-specification structure in which to save the maximum size. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

maxbytes   A pointer to an **ifx_int8_t** structure that contains the maximum number of bytes for the smart large object. If this value is -1, the smart large object has no size limit.

Usage:

IBM Informix does not allow the size of a smart large object to exceed the *maxbytes* value. The database server does not obtain the value from the inheritance hierarchy of storage characteristics. For more information about the maximum size, see Table 8-1 on page 8-4.

Return codes:

0   The function was successful.

-1   The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_specget_maxbytes() function

The ifx_lo_specset_estbytes() function

The ifx_lo_specset_extsz() function

The ifx_lo_specset_flags() function

The ifx_lo_specset_sbspace() function

The ifx_lo_spec_free() function

The ifx_lo_specset_sbspace() function

The **ifx_lo_specset_sbspace()** function sets the name of the sbspace for a smart large object.

Syntax:

```
mint ifx_lo_specset_sbspace(LO_spec, sbspace_name)
   ifx_lo_create_spec_t *LO_spec;
   char *sbspace_name;
```

sbspace_name A pointer to a buffer that contains the name of the sbspace in which to store the smart large object.

LO_spec A pointer to the LO-specification structure in which to save the sbspace name. For information about the LO-specification structure, see "The LO-specification structure" on page 8-3.

Usage:

The name of the sbspace can be at most 18 characters long. It must also be null terminated.

If you do not specify an *sbspace_name* when you create a new smart large object, IBM Informix obtains the sbspace name from either the column information or from the SBSPACENAME parameter of the ONCONFIG file. For more information, see "Obtain storage characteristics" on page 8-6.

Return codes:

0 The function was successful.

-1 The function was not successful.

Related functions:

The ifx_lo_def_create_spec() function

The ifx_lo_specget_sbspace() function

The ifx_lo_specset_estbytes() function

The ifx_lo_specset_extsz() function

The ifx_lo_specset_flags() function

The ifx_lo_specset_maxbytes() function

The ifx_lo_spec_free() function

The ifx_lo_stat() function

The **ifx_lo_stat()** function returns information about the current status of an open smart large object.

Syntax:
```
mint ifx_lo_stat(LO_fd, LO_stat)
   mint LO_fd;
   ifx_lo_stat_t **LO_stat;
```

LO_fd The LO-file descriptor for the open smart large object whose status information you wish to obtain. For more information about an LO-file descriptor, see "The LO file descriptor" on page 8-11.

LO_stat A pointer that points to a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more

information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The **ifx_lo_stat()** function allocates an LO-status structure, **ifx_lo_stat_t**, and initializes it with the status information for the smart large object that the *LO_fd* file descriptor identifies. To access the status information, use the IBM Informix ESQL/C accessor functions for the LO-status structure. For more information about the status information and the corresponding accessor functions, see Table 8-4 on page 8-19.

Use the **ifx_lo_stat_free()** function to deallocate an LO-status structure.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_create() function

The ifx_lo_open() function

The ifx_lo_stat_atime() function

The ifx_lo_stat_cspec() function

The ifx_lo_stat_ctime() function

The ifx_lo_stat_free() function

The ifx_lo_stat_mtime_sec() function

The ifx_lo_stat_size() function

Example:

For an example of the **ifx_lo_stat()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_stat_atime() function

The **ifx_lo_stat_atime()** function returns the time of last access for a smart large object.

Syntax:
```
mint ifx_lo_stat_atime(LO_stat)
   ifx_lo_stat_t *LO_stat;
```

LO_stat	A pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The time of last access is only guaranteed to be maintained if the smart large object has the LO_KEEP_LASTACCESS_TIME flag set. If you do not set this flag, the database server does not write this access-time value to disk. The resolution of the time that the **ifx_lo_stat_atime()** function returns is seconds.

The status information for the smart large object is in the LO-status structure to which *LO_stat* points. The **ifx_lo_stat()** function allocates this structure and fills it with the status information for a particular smart large object. Therefore, you must precede a call to **ifx_lo_stat_atime()** with a call to **ifx_lo_stat()**.

Return codes:

>=0   The last-access time for the smart large object that *LO_stat* identifies.

-1   The function was not successful.

Related functions:

The ifx_lo_stat() function

The ifx_lo_stat_cspec() function

The ifx_lo_stat_ctime() function

The ifx_lo_stat_free() function

The ifx_lo_stat_mtime_sec() function

Example:

For an example of the **ifx_lo_stat_atime()** function, see "The get_lo_info.ec program" on page C-5

The ifx_lo_stat_cspec() function

The **ifx_lo_stat_cspec()** function returns the LO-specification structure for a smart large object.

Syntax:
```
ifx_lo_create_spec_t *ifx_lo_stat_cspec(LO_stat)
   ifx_lo_stat_t *LO_stat;
```

LO_stat   A pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The **ifx_lo_stat_cspec()** function returns a pointer to an LO-specification structure, **ifx_lo_create_spec_t**, which contains the storage characteristics for the specified smart large object. You can use this LO-specification structure to create another smart large object with the same storage characteristics or to access the storage characteristics through the accessor (**ifx_specget_**) functions.

You must precede a call to **ifx_lo_stat_cspec()** with a call to **ifx_lo_stat()**. The **ifx_lo_stat()** function allocates the memory for the **ifx_lo_create_spec_t** structure, along with the **ifx_lo_stat_t** structure, and initializes it with the status information for the smart large object that you specified. When you call the **ifx_lo_stat_free()** function to free the **ifx_lo_stat_t** structure, it frees the **ifx_lo_create_spec_t** structure automatically.

Return codes:

A valid pointer to an LO-specification (**ifx_lo_create_spec_t**) structure
The function was successful.

NULL   The function was not successful.

Related functions:

The ifx_lo_stat() function

The ifx_lo_stat_atime() function

The ifx_lo_stat_ctime() function

The ifx_lo_stat_free() function

The ifx_lo_stat_mtime_sec() function

Example:

For an example of the **ifx_lo_stat_cspec()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_stat_ctime() function

The **ifx_lo_stat_ctime()** function returns the time of last change in status for a smart large object.

Syntax:
```
mint ifx_lo_stat_ctime(LO_stat)
   ifx_lo_stat_t *LO_stat;
```

LO_stat   A pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The last change in status includes modification of storage characteristics, including a change in the number of references and writes to the smart large object. The resolution of the time that the **ifx_lo_stat_ctime()** function returns is seconds.

The status information for the smart large object is in the LO-status structure to which *LO_stat* points. The **ifx_lo_stat()** function allocates this structure and fills it with the status information for a particular smart large object. Therefore, you must precede a call to **ifx_lo_stat_ctime()** with a call to **ifx_lo_stat()**.

Return codes:

>=0	The last-change time for the smart large object that *LO_stat* identifies.
-1	The function was not successful.

Related functions:

The ifx_lo_stat() function

The ifx_lo_stat_atime() function

The ifx_lo_stat_cspec() function

The ifx_lo_stat_free() function

The ifx_lo_stat_mtime_sec() function

Example:

For an example of the **ifx_lo_stat_ctime()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_stat_free() function

The **ifx_lo_stat_free()** function frees an LO-status structure.

Syntax:
```
mint ifx_lo_stat_free(LO_stat)
   ifx_lo_stat_t *LO_stat;
```

LO_stat	A pointer to an LO-status structure that the **ifx_lo_stat()** function has allocated. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The **ifx_lo_stat()** function returns status information about an open smart large object in an LO-status structure. When your application no longer needs this status information, use the **ifx_lo_stat_free()** function to deallocate the LO-status structure.

Return codes:

0	The function was successful.
-1	The function was not successful.

Related function:

The ifx_lo_stat() function

Example:

For an example of the **ifx_lo_stat_free()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_stat_mtime_sec() function

The **ifx_lo_stat_mtime_sec()** function returns the time of last modification for a smart large object.

Syntax:
```
mint ifx_lo_stat_mtime_sec(LO_stat)
   ifx_lo_stat_t *LO_stat;
```

LO_stat   A pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The resolution of the time that the **ifx_lo_stat_mtime_sec()** function returns is seconds.

The status information for the smart large object is in the LO-status structure to which *LO_stat* points. The **ifx_lo_stat()** function allocates this structure and fills it with the status information for a particular smart large object. Therefore, you must precede a call to **ifx_lo_stat_mtime_sec()** with a call to **ifx_lo_stat()**.

Return codes:

>=0   The last-modification time for the smart large object that *LO_stat* identifies.

-1   The function was not successful.

Related functions:

The ifx_lo_stat() function

The ifx_lo_stat_atime() function

The ifx_lo_stat_cspec() function

The ifx_lo_stat_ctime() function

The ifx_lo_stat_free() function

Example:

For an example of the **ifx_lo_stat_mtime_sec()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_stat_refcnt() function

The **ifx_lo_stat_refcnt()** function returns the number of references to a smart large object.

Syntax:
```
mint ifx_lo_stat_refcnt(LO_stat)
   ifx_lo_stat_t *LO_stat;
```

LO_stat   A pointer to an LO-status structure that **ifx_lo_stat()** allocates and

fills in with status information. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.

Usage:

The *refcnt* argument is the reference count for a smart large object. This count indicates the number of persistently stored LO-pointer (**ifx_lo_t**) structures that currently exist for the smart large object. The database server assumes that it can safely remove the smart large object and reuse any resources that are allocated to it when the reference count is zero (0) and any of the following conditions exist:

- The transaction in which the reference count is decremented commits.
- The connection terminates and the smart large object is created during this connection but its reference count is not incremented.

   The database server increments a reference counter when it stores the LO-pointer structure for a smart large object in a row.

The status information for the smart large object is in the LO-status structure to which *LO_stat* points. The **ifx_lo_stat()** function allocates this structure and fills it with the status information for a particular smart large object. Therefore, you must precede a call to **ifx_lo_stat_refcnt()** with a call to **ifx_lo_stat()**.

Return codes:

>=0	The reference count for the smart large object that *LO_stat* identifies.
-1	The function was not successful.

Related functions:

The ifx_lo_stat() function

The ifx_lo_stat_atime() function

The ifx_lo_stat_cspec() function

The ifx_lo_stat_ctime() function

The ifx_lo_stat_free() function

The ifx_lo_stat_mtime_sec() function

Example:

For an example of the **ifx_lo_stat_refcnt()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_stat_size() function

The **ifx_lo_stat_size()** function returns the size, in bytes, of a smart large object.

Syntax:
```
mint ifx_lo_stat_size(LO_stat, size)
   ifx_lo_stat_t *LO_stat;
   ifx_int8_t *size;
```

LO_stat	A pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information about the LO-status structure, see "Allocating and Accessing an LO-Status Structure" on page 8-18.
size	A pointer to an **ifx_int8_t** structure that **ifx_lo_stat_size()** fills with the size in bytes, of the smart large object.

Usage:

The status information for the smart large object is in the LO-status structure to which *LO_stat* points. The **ifx_lo_stat()** function allocates this structure and fills it with the status information for a particular smart large object. Therefore, you must precede a call to **ifx_lo_stat_size()** with a call to **ifx_lo_stat()**.

Return codes:

0	The function was successful.
-1	The function was not successful.

Related functions:

The ifx_lo_stat() function

The ifx_lo_stat_atime() function

The ifx_lo_stat_cspec() function

The ifx_lo_stat_ctime() function

The ifx_lo_stat_free() function

The ifx_lo_stat_mtime_sec() function

Example:

For an example of the **ifx_lo_stat_size()** function, see "The get_lo_info.ec program" on page C-5.

The ifx_lo_tell() function

The **ifx_lo_tell()** function returns the current file or seek position for an open smart large object.

Syntax:
```
mint ifx_lo_tell(LO_fd, seek_pos)
   mint LO_fd;
   ifx_int8_t *seek_pos;
```

LO_fd	The LO file descriptor for the open smart large object whose seek position you wish to determine. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.
seek_pos	A pointer to the 8-byte integer that identifies the current seek position.

Usage:

The seek position is the offset for the next read or write operation on the smart large object that is associated with the LO file descriptor, *LO_fd*. The **ifx_lo_tell()** function returns this seek position in the user-defined **int8** variable, *seek_pos*.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the failure.

Related functions:

The ifx_lo_readwithseek() function

The ifx_lo_seek() function

The ifx_lo_writewithseek() function

Example:

The ifx_lo_to_buffer() function

The **ifx_lo_to_buffer()** function copies a specified number of bytes from a smart large object into a user-defined buffer.

Syntax:
```
mint ifx_lo_to_buffer(LO_ptr, size, buf_ptr)
   ifx_lo_t *LO_ptr;

   mint size;
   char **buf_ptr;

   mint error;
```

LO_ptr   The LO-pointer structure for the smart large object from which you want to copy the data.

size   A **mint** that identifies the number of bytes to copy from the smart large object

buf_ptr   A doubly indirect pointer to a user-defined buffer to which you want to copy the data.

error   Contains the address of the **mint** that holds the error code that **ifx_lo_to_buffer()** sets

Usage:

The **ifx_lo_to_buffer()** function copies bytes, up to the size that the *size* argument specifies from the smart large object that the *LO_ptr* argument identifies. The read operation from the smart large object starts at a zero-byte offset. If the smart large object is smaller than the *size* value, **ifx_lo_to_buffer()** copies only the number of bytes in the smart large object. If the smart large object contains more than *size* bytes, the **ifx_lo_to_buffer()** function only copies up to *size* bytes into the user-defined buffer.

When **buf_ptr** is NULL, **ifx _lo_to_buffer()** allocates the memory for the user-defined buffer. Otherwise, the function assumes that you have allocated memory that *buf_ptr* identifies.

Return codes:

0	The number of bytes copied from the smart large object to the user-defined buffer that *buf_ptr* identifies.
-1	The function was not successful.

The ifx_lo_truncate() function

The **ifx_lo_truncate()** function truncates a smart large object at a specified byte position.

Syntax:
```
mint ifx_lo_truncate(LO_fd, offset)
   mint LO_fd;
   ifx_int8_t *offset;
```

LO_fd	The LO file descriptor for the open smart large object whose value you wish to truncate. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.
offset	A pointer to the 8-byte integer that identifies the offset at which the truncation of the smart large object begins.

Usage:

The **ifx_lo_truncate()** function sets the last valid byte of a smart large object to the specified *offset* value. If this *offset* value is beyond the current end of the smart large object, you actually extend the smart large object. If this *offset* value is less than the current end of the smart large object, the database server reclaims all storage, from the position that *offset* indicates to the end of the smart large object.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

The ifx_lo_unlock() function

The **ifx_lo_unlock()** function allows you to unlock a range of bytes in a smart large object that was locked by the **ifx_lo_lock()** function.

Syntax:
```
mint ifx_lo_unlock(lofd, offset, whence, range)
   mint lofd;
   int8 *offset;
   mint whence;

   int8 *range;
```

LO_fd	The LO-file descriptor for the smart large object in which to unlock the range of bytes. For more information about an LO-file descriptor, see "The LO file descriptor" on page 8-11.
offset	A pointer to the 8-byte integer (INT8) that specifies the offset within the smart large object at which unlocking is to begin.
whence	An integer constant that specifies from what point the offset is

calculated: the beginning of the smart large object, the current position within the smart large object, or the end of the smart large object.

range A pointer to the 8-byte integer (INT8) that specifies the number of bytes to unlock.

Usage:

The **ifx_lo_unlock()** function unlocks the number of bytes specified by *nbytes*, beginning at the offset specified by *offset* and *whence*, for the smart large object specified by *LO_fd*. Before calling **ifx_lo_unlock()**, you must obtain a valid LO-file descriptor by calling either **ifx_lo_create()** to create a new smart large object or by calling **ifx_lo_open()** to open an existing smart large object. For more information about the possible values for *offset*, *whence*, and *nbytes*, see "The ifx_lo_lock() function" on page B-115.

Return codes:

0   The function was successful

< 0   The function was unsuccessful. The value returned is the **sqlcode**, which is the number of the IBM Informix error message. For more information about **sqlcode**, see Chapter 11, "Exception handling"

Related function:

The ifx_lo_lock() function

The ifx_lo_write() function

The **ifx_lo_write()** function writes a specified number of bytes of data to an open smart large object.

Syntax:
```
mint ifx_lo_write(LO_fd, buf, nbytes, error)
   mint LO_fd;
   char *buf;
   mint nbytes;
   mint *error;
```

LO_fd The LO file descriptor for the smart large object to which to write. For more information about an LO file descriptor, see "The LO file descriptor" on page 8-11.

buf A pointer to a buffer that contains the data that the function writes to the smart large object.

nbytes The number of bytes to write to the smart large object. With a minimum length of 0, this value must be less than 2 GB.

error A pointer to a **mint** that contains the error code that **ifx_lo_write()** sets.

Usage:

The **ifx_lo_write()** function writes *nbytes* of data to the smart large object that the *LO_fd* file descriptor identifies. The write begins at the current seek position for *LO_fd*. You can use the **ifx_lo_tell()** function to obtain the current seek position.

The function obtains the data from the user-defined buffer to which *buf* points. The *buf* buffer must be less than 2 gigabytes in size.

If the database server writes less than *nbytes* of data to the smart large object, the **ifx_lo_write()** function returns the number of bytes that it wrote and sets the *error* value to point to a value that indicates the reason for the incomplete write operation. This condition can occur when the sbspace runs out of space.

Return codes:

>=0   The number of bytes that the function has written from the *buf* character buffer to the open smart large object.

-1   The function was not successful; examine the *error* for a detailed error code.

Related functions:

The ifx_lo_open() function

The ifx_lo_read() function

The ifx_lo_seek() function

The ifx_lo_tell() function

The ifx_lo_writewithseek() function

The ifx_lo_writewithseek() function

The **ifx_lo_writewithseek()** function performs a seek operation and then writes a specified number of bytes of data to an open smart large object.

Syntax:
```
mint ifx_lo_writewithseek(LO_fd, buf, nbytes, offset, whence, error)
   mint LO_fd;
   char *buf;
   mint nbytes;
   ifx_int8_t *offset;
   mint whence;
   mint *error;
```

LO_fd   The LO file descriptor for the smart large object to which to write. For more information about an LO-file descriptor, see "The LO file descriptor" on page 8-11.

buf   A pointer to a buffer that contains the data that the function writes to the smart large object.

nbytes   The number of bytes to write to the smart large object. This value cannot exceed 2 gigabytes.

offset   A pointer to the 8-byte integer (INT8) offset from the starting seek position.

whence   A **mint** value that identifies the starting seek position.

error   A pointer to a **mint** that contains the error code that **ifx_lo_writewithseek()** sets.

Usage:

The **ifx_lo_writewithseek()** function writes *nbytes* of data to the smart large object that the LO_*fd* file descriptor identifies. The function obtains the data to write from the user-defined buffer to which *buf* points. The buffer must be less than 2 gigabytes in size.

The write begins at the seek position of *LO_fd* that the *offset* and *whence* arguments indicate, as follows:

- The *whence* argument identifies the position from which to start the seek.

  Valid values include the following constants, which the **locator.h** header file defines.

Whence constant	Starting seek position
LO_SEEK_SET	The start of the smart large object
LO_SEEK_CUR	The current seek position in the smart large object
LO_SEEK_END	The end of the smart large object

  - The *offset* argument identifies the offset, in bytes, from the starting seek position (that the *whence* argument specifies) to which the seek position should be set.

    For more information about how to access eight-bit (INT8) integers, see "The int8 Data Type" on page 5-2.

If the database server writes less than *nbytes* of data to the smart large object, the **ifx_lo_writewithseek()** function returns the number of bytes that it wrote and sets the *error* value to point to a value that indicates the reason for the incomplete write operation. This condition can occur when the sbspace runs out of space.

Return codes:

| >=0 | The number of bytes that the function has written from the *buf* character buffer to the smart large object. |
| -1 | The function was not successful; examine the *error* for a detailed error code. |

Related functions:

The ifx_lo_open() function

The ifx_lo_writewithseek() function

The ifx_lo_seek() function

The ifx_lo_write() function

Example:

For an example of the **writewithseek()** function, see "The create_clob.ec program" on page C-2 and the "The ifx_lo_seek() function" on page B-121.

The ifx_lvar_alloc() function

The **ifx_lvar_alloc()** function specifies whether to allocate memory when fetching **lvarchar** data.

Syntax:
```
mint ifx_lvar_alloc(mintalloc)
    mint alloc;
```

alloc   The value of the allocation flag; either **1** or **0**

Usage:

When the flag is set to 1, ESQL/C automatically performs this memory allocation. You can use a flag value of 1 before a SELECT statement when you are unsure of the amount of data that the SELECT statement returns. When the flag is set to 0, ESQL/C does not automatically perform this memory allocation.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the error.

The ifx_putenv() function

The **ifx_putenv()** function changes the value of an existing environment variable, creates a new environment variable, or removes a variable from the runtime environment.

Syntax:
```
int ifx_putenv( envstring );
    const  char *envstring;
```

envstring   A pointer to a string of the form *varname=string*, in which *varname* is the name of the environment variable to add or modify and *string* is the variable value.

Usage:

The **ifx_putenv()** function adds new environment variables or modifies the values of existing environment variables in the **InetLogin** structure. These variables define the environment in which a process executes. If *varname* is already part of the environment, **ifx_putenv()** replaces the existing value with *string*; otherwise, **ifx_putenv()** adds *varname* to the environment, with the value *string*.

To remove a variable from the runtime environment, specify *varname* to its default value. If the default value is NULL, setting the variable to a null string with **ifx_putenv()** effectively removes it from the runtime environment. If the default value of the variable is not NULL, then setting the variable to a null string with **ifx_putenv()** resets the variable to its default value, but does not remove it from the runtime environment.

The **ifx_putenv()** function sets IBM Informix variables first and then non-Informix variables. For a list of Informix environment variables, see "Fields of the InetLogin Structure" on page 1-33.

The following call to the **ifx_putenv()** function changes the value of the **INFORMIXDIR** environment variable:

```
ifx_putenv( "informixdir=c:\informix" );
```

This function affects only the environment variable of the current process. The environment of the command processor does not change.

Return codes:

0	The call to **ifx_putenv()** was successful.
-1	The call to **ifx_putenv()** was not successful.

The ifx_strdate() function

The **ifx_strdate()** function converts a character string to an internal DATE.

Syntax:

```
mint ifx_strdate(str, jdate, dbcentury)
    char *str;
    int4 *jdate;
    char dbcentury;
```

str	A is a pointer to the string that contains the date to convert.
jdate	A pointer to a **int4** integer that receives the internal DATE value for the *str* string.
dbcentury	Can be one of the following characters, which determines which century to apply to the year portion of the date:

	R	Present. The function uses the two high-order digits of the current year to expand the year value.
	P	Past. The function uses the past and present centuries to expand the year value. It compares these two dates against the current date and uses the century that is prior to the current century. If both dates are prior to the current date, the function uses the century closest to the current date.
	F	Future. The function uses the present and the next centuries to expand the year value. It compares these against the current date and uses the century that is later than the current date. If both dates are later than the current date, the function uses the date closest to the current date.
	C	Closest. The function uses the past, present, and next centuries to expand the year value. It chooses the century that is closest to the current date.

Usage:

For the default locale, U.S. English, the **ifx_strdate()** function determines how to format the character string with the following precedence:

1. The format that the **DBDATE** environment variable specifies (if **DBDATE** is set). For more information about **DBDATE**, see the *IBM Informix Guide to SQL: Reference*.

2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set). For more information about **GL_DATE**, see the *IBM Informix GLS User's Guide*.

3. The default date form: mm/dd/yyyy. You can use any nonnumeric character as a separator between the month, day, and year. You can express the year as four digits (2007) or as two digits (07).

When you use a nondefault locale and do not set the **DBDATE** or **GL_DATE** environment variable, **ifx_strdate()** uses the date end-user format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide*.

When you use a two-digit year in the date string, the **ifx_strdate()** function uses the value of the *dbcentury* argument to determine which century to use. If you do not set the *dbcentury* argument, **ifx_strdate()** uses the DBCENTURY environment variable to determine which century to use. If you do not set **DBCENTURY**, **ifx_strdate()** assumes the current century for two-digit years. For information about the **DBCENTURY** environment variable, see the *IBM Informix Guide to SQL: Reference*.

Return codes:

0	The conversion was successful.
< 0	The conversion failed.
-1204	The *str* parameter specifies an invalid year.
-1205	The *str* parameter specifies an invalid month.
-1206	The *str* parameter specifies an invalid day.
-1212	Data conversion format must contain a month, day, or year component. **DBDATE** specifies the data conversion format.
-1218	The date specified by the *str* argument does not properly represent a date.

The ifx_var_alloc() function

The **ifx_var_alloc()** function allocates memory for the data buffer of an **lvarchar** or **var binary** host variable.

Syntax:

var binary
```
mint ifx_var_alloc(var_bin, var_size)
   var binary **var_bin
      int4 var_size;
```

lvarchar
```
mint ifx_var_alloc(lvar, var_size)
   lvarchar **lvar
   int4 var_size;
```

var_bin	The address of the **var binary** pointer host variable whose data buffer is allocated.
lvar	The address of the **lvarchar pointer** host variable whose data buffer is allocated.
var_size	The size, in bytes, of the data buffer to allocate.

Usage:

The allocation flag of the **ifx_var_flag()** function notifies IBM Informix ESQL/C of the allocation method to use for the data buffer. If you set the allocation flag in **ifx_var_flag()** to 0, you must explicitly allocate memory for the data buffer of a **var binary** host variable with the **ifx_var_alloc()** function.

Important: Whether you allocate memory or allow Informix ESQL/C to allocate the memory for you, you must free the allocated memory using the **ifx_var_dealloc()** function.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the error.

Related functions:

The ifx_var_dealloc() function

The ifx_var_flag() function

The ifx_var_dealloc() function

The **ifx_var_dealloc()** function deallocates the memory that was allocated for the data buffer of a **var binary** host variable.

Syntax:

var binary
```
mint ifx_var_dealloc(var_bin)
    var binary **var_bin;
```

lvarchar
```
mint ifx_var_dealloc(lvar)
    lvarchar **lvar;
```

var_bin   The address of the **var binary** pointer host variable whose data buffer is deallocated.

lvar   The address of the **lvarchar pointer** host variable whose data buffer is allocated.

Usage:

The allocation flag of the **ifx_var_flag()** function tells Informix ESQL/C which allocation method to use for the data buffer. Regardless of whether Informix ESQL/C (allocation flag set to 1) or your application (allocation flag set to 0) allocates the memory, you must explicitly deallocate memory that was allocated to an **lvarchar** or the data buffer of **var binary** host variable.

Return codes:

0   The function was successful.

<0   The function was not successful and the return value indicates the cause of the error.

Related functions:

The ifx_var_alloc() function

The ifx_var_flag() function

The ifx_var_flag() function

The **ifx_var_flag()** function determines how memory is allocated for the data buffer of an **lvarchar** or **var binary** host variable.

Syntax:

var binary
```
mint ifx_var_flag(var_bin, flag)
   var binary **var_bin;
   int2 flag;
```

lvarchar
```
mint ifx_var_flag(lvar, flag)
   lvarchar **lvar;
   int2 flag;
```

flag	The **int2** value of the allocation flag, either 0 or 1.
var_bin	The address of the **var binary** host variable.
lvar	The address of the **lvarchar pointer** host variable.

Usage:

The value of the *flag* argument is the allocation flag. It determines who handles memory allocation for the data of the *var_bin* host variable, as follows:

- When *flag* is one, IBM Informix ESQL/C automatically performs this memory allocation.

  You can use a *flag* value of 1 before a SELECT statement when you are unsure of the amount of data that the SELECT returns.

- When *flag* is zero, Informix ESQL/C does not automatically perform this memory allocation.

  When you set flag to 0, you must allocate memory for the data buffer of the *lvar* or *var_bin* variable with the **ifx_var_alloc()** functions.

If you do not call the **ifx_var_flag()** function for an **lvarchar** or **var binary** host variable, Informix ESQL/C allocates memory for its data buffer. Whether you allocate memory for the **lvarchar** or **var binary** variable, or allow Informix ESQL/C to do it for you, you must free the memory with the **ifx_var_dealloc()** function.

Return codes:

0	The function was successful.
<0	The function was *not* successful and the return value indicates the cause of the error.

Related functions:

The ifx_var_alloc() function

The ifx_var_freevar() function

The ifx_var_freevar() function frees memory that has been allocated for the **var binary** and **lvarchar pointer** host variables.

Syntax:

```
int fix_var_freevar(var_bin)
   var binary *var_bin;
```

var_bin   The address of the **var binary** or **lvarchar pointer** host variable.

Usage:

Whenever you have a **var binary** or **lvarchar pointer** host variable, as shown in the following example, you must explicitly free memory that is allocated for it by using the **ifx_var_freevar()** function.

```
EXEC SQL var binary 'polygon' poly;
EXEC SQL lvarchar *c;
```

The following example illustrates the use of **ifx_var_freevar()**. You must explicitly free memory that has been allocated for **var binary** and **lvarchar pointer** host variables by using the **ifx_var_freevar()** function.

```
ifx_var_freevar(&poly);
ifx_var_freevar(&c);
```

If you do not use **ifx_var_dealloc()** to deallocate memory that has been allocated for the data buffer of the **var binary** host variable, **ifx_var_freevar()** will do so. It then frees the memory of the **var binary** and **lvarchar pointer** host variables. In the example above, after **ifx_var_freevar()** was called, *poly* and *c* would be set to null.

Return codes:

0   The function was successful

<0   The function was not successful and the return value indicates the cause of the error

Related functions:

The ifx_var_flag() function
The ifx_var_alloc() function
The ifx_var_dealloc() function

The ifx_var_getdata() function

The **ifx_var_getdata()** function returns the data from an **lvarchar** or **var binary** host variable.

Syntax:

var binary

```
void *ifx_var_getdata(var_bin)
   var binary **var_bin;
```

lvarchar

```
void *ifx_var_getdata(lvar)
   lvarchar **lvar;
```

var_bin	The address of the **var binary** host variable whose data is retrieved.
lvar	The address of the **lvarchar pointer** host variable whose data is retrieved.

Usage:

The **ifx_var_getdata()** function returns the data as a **void *** pointer. Your Informix ESQL/C application must cast this pointer to the correct data type. When you use **ifx_var_getdata()** on an **lvarchar pointer**, you must cast the returned (void) pointer to a C-language **character pointer** (char *).

Return codes:

Null pointer   The function was not successful.

Valid pointer to the data buffer
          The function was successful.

Related functions:

The ifx_var_getlen() function

The ifx_var_setdata() function

The ifx_var_getlen() function

The ifx_var_getlen() function returns the length of the data in an **lvarchar pointer** or **var binary** host variable.

Syntax:

var binary
```
mint ifx_var_getlen(var_bin)
   var binary **var_bin;
```

lvarchar
```
mint ifx_var_getlen(lvar)
   lvarchar **lvar;
```

var_bin	The address of the var binary host variable whose length is returned.
lvar	The address of the lvarchar pointer host variable whose length is returned.

Usage:

The length that the **ifx_var_getlen()** function returns is the number of bytes that have been allocated for the data buffer of the *lvar* or *var_bin* host variable.

If you get an **lvarchar pointer** or **var binary** from a descriptor area by using the DATA clause of a GET DESCRIPTOR statement, the value will be null terminated.

If you use **ifx_var_getlen()** on such a variable the length returned will include the null terminator. To get the correct length use the LENGTH clause of the GET DESCRIPTOR statement.

Return codes:

>=0   The length of the data buffer for the var_bin host variable.

<0   The function was not successful.

Related functions:

The ifx_var_getdata() function

The ifx_var_setlen() function

The ifx_var_isnull() function

The **ifx_var_isnull()** function checks whether an **lvarchar** or **var binary** host variable contains a null value.

Syntax:

var binary
```
mint ifx_var_isnull(var_bin)
   var binary **var_bin;
```

lvarchar
```
mint ifx_var_isnull(lvar)
   lvarchar **lvar;
```

var_bin   The address of the **var binary** host variable.

lvar   The address of the **lvarchar pointer** host variable.

Usage:

The **ifx_var_isnull()** function checks for a null value in an **lvarchar** or **var binary** host variable. To determine whether an IBM Informix ESQL/C host variable of any other data type contains null, use the **risnull()** library function. For more information, see B-191

Return codes:

0   The opaque-type data is not a null value.

1   The opaque-type data is a null value.

Related function:

The ifx_var_setnull() function

The ifx_var_setdata() function

The **ifx_var_setdata()** function stores data in an **lvarchar** or **var binary** host variable.

Syntax:

var binary

```
mint ifx_var_setdata(var_bin, buffer, buf_len)
   var binary **var_bin;
   char *buffer;
   int4 buf_len;
```

lvarchar

```
mint ifx_var_setdata(lvar, buffer, buf_len)
   lvarchar **lvar;
   char *buffer;
   int4 buf_len;
```

buffer	A character buffer that contains the data to store in the *lvar* or *var_bin* host variable.
buf_len	The length, in bytes, of the buffer.
var_bin	The address of the **var binary** host variable.
lvar	The address of the **lvarchar pointer** host variable.

Usage:

The **ifx_var_setdata()** function stores the data in *buffer* in the data buffer of the *lvar* or *var_bin* host variable. For an **lvarchar pointer** host variable, IBM Informix ESQL/C expects the data inside *buffer* to be null-terminated ASCII data.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the error.

Related functions:

The ifx_var_getdata() function

The ifx_var_setlen() function

The ifx_var_setlen() function

The **ifx_var_setlen()** function stores the length of the data buffer for an **lvarchar** or **var binary** host variable.

Syntax:

var binary

```
mint ifx_var_setlen(var_bin, length)
   var binary **var_bin;
   int4 length
```

lvarchar

```
mint ifx_var_setlen(lvar, length)
   lvarchar **lvar;
   int4 length
```

length	The length, in bytes, of the data buffer to allocate for the **var binary** data.
var_bin	The address of the **var binary** host variable.
lvar	The address of the **lvarchar pointer** host variable.

Usage:

The *length* that the **ifx_var_setlen()** function sets is the number of bytes to allocate for the data buffer of the *lvar* or *var_bin* host variable. Call this function to change the size of the data buffer that the **ifx_var_alloc()** function allocated for the *lvar* or *var_bin* host variable.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the error.

Related functions:

The ifx_var_getlen() function

The ifx_var_setdata() function

The ifx_var_setnull() function

The **ifx_var_setnull()** function sets an **lvarchar** or **var binary** host variable to a null value.

Syntax:

var binary
```
mint ifx_var_setnull(var_bin, flag)
   var binary **var_bin;
   mint flag
```

lvarchar
```
mint ifx_var_setnull(var_bin, flag)
   var binary **var_bin;
   mint flag;
```

var_bin	The address of the **var binary** host variable.
lvar	The address of the **lvarchar pointer** host variable.
flag	The value 0 to indicate a non-null value or 1 to indicate a null value.

Usage:

The **ifx_var_setnull()** function sets a host variable of type **lvarchar** or **var binary** to a null value. To set an IBM Informix ESQL/C host variable of any other data type to null, use the **rsetnull()** library function. For more information, see B-197.

Return codes:

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the error.

Related functionf:

The ifx_var_isnull() function

The incvasc() function

The **incvasc()** function converts a string that conforms to the ANSI SQL standard for an INTERVAL value to an **interval** value. For information about the ANSI SQL **interval** standard, see "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

Syntax:

```
mint incvasc(inbuf, invvalue)
   char *inbuf;
   intrvl_t *invvalue;
```

inbuf	A pointer to a buffer that contains an ANSI-standard INTERVAL string.
invvalue	A pointer to an initialized **interval** variable.

Usage:

You must initialize the **interval** variable in *invvalue* with the qualifier that you want this variable to have.

The character string in *inbuf* can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can only contain characters that are digits and delimiters that are appropriate to the qualifier fields of the **interval** variable.

If the character string is an empty string, the **incvasc()** function sets the value in *invvalue* to null. If the character string is acceptable, the function sets the value in the **interval** variable and returns zero. Otherwise, the function sets the value in the **interval** value to null.

Return codes:

0	The conversion was successful.
-1260	It is not possible to convert between the specified types.
-1261	Too many digits in the first field of **datetime** or **interval**.
-1262	Non-numeric character in **datetime** or **interval**.
-1263	A field in a **datetime** or **interval** value is out of range or incorrect.
-1264	Extra characters at the end of a **datetime** or **interval** value.
-1265	Overflow occurred on a **datetime** or **interval** operation.
-1266	A **datetime** or **interval** value is incompatible with the operation.
-1267	The result of a **datetime** computation is out of range.
-1268	A parameter contains an invalid **datetime** or **interval** qualifier.

Example:

The **demo** directory contains this sample program in the file **incvasc.ec**.

```
/*
   * 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>

EXEC SQL include datetime;

main()
{
    mint x;

    EXEC SQL BEGIN DECLARE SECTION;
      interval day to second in1;
    EXEC SQL END DECLARE SECTION;

    printf("INCVASC Sample ESQL Program running.\n\n");

    printf("Interval string #1 = 20 3:10:35\n");
    if(x = incvasc("20 3:10:35", &in1))
      printf("Result = failed with conversion error:%d\n",x);
    else
      printf("Result = successful conversion\n");

/*
    * Note that the following literal string has a 26 in the hours field
    */
    printf("\nInterval string #2 = 20 26:10:35\n");
    if(x = incvasc("20 26:10:35", &in1))
      printf("Result = failed with conversion error:%d\n",x);
    else
      printf("Result = successful conversion\n");

  /*
    * Try to convert using an invalid qualifier (YEAR to SECOND)
    */
    printf("\nInterval string #3 = 2007-02-11 3:10:35\n");
    in1.in_qual = TU_IENCODE(4, TU_YEAR, TU_SECOND);
    if(x = incvasc("2007-02-11 3:10:35", &in1))
      printf("Result = failed with conversion error:%d\n",x);
    else
      printf("Result = successful conversion\n");

    printf("\nINCVASC Sample Program over.\n\n");
}
```

Output:

```
INCVASC Sample ESQL Program running.

Interval string #1 = 20 3:10:35
Result = successful conversion

Interval string #2 = 20 26:10:35
Result = failed with conversion error:-1263
```

```
              Interval string #3 = 2007-02-11 3:10:35
              Result = failed with conversion error:-1268

              INCVASC Sample Program over.
```

The incvfmtasc() function

The **incvfmtasc()** function uses a formatting mask to convert a character string to an **interval** value.

Syntax:
```
mint incvfmtasc(inbuf, fmtstring, invvalue)
   char *inbuf;
   char *fmtstring;
   intrvl_t *invvalue;
```

inbuf   A pointer to a buffer that contains the string to convert.

fmtstring   A pointer to the buffer that contains the formatting mask to use for the inbuf string. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*).

invvalue   A pointer to the initialized **interval** variable.

Usage:

You must initialize the **interval** variable in *invvalue* with the qualifier you want this variable to have. The **interval** variable does not need to specify the same qualifier as the formatting mask. When the **interval** qualifier is different from the implied formatting-mask qualifier, **incvfmtasc()** converts the result to appropriate units as necessary. However, both qualifiers must belong to the same interval class: either the **year to month** class or the **day to fraction** class.

All fields in the character string in *inbuf* must be contiguous. In other words, if the qualifier is **hour to second**, you must specify all values for **hour**, **minute**, and **second** somewhere in the string, or **incvfmtasc()** returns an error.

The *inbuf* character string can have leading and trailing spaces. However, from the first significant digit to the last, *inbuf* can contain only digits and delimiters that are appropriate for the qualifier fields that the formatting mask implies. For more information about acceptable digits and delimiters for an INTERVAL value, see "ANSI SQL standards for DATETIME and INTERVAL values" on page 6-7.

If the character string is acceptable, the **incvfmtasc()** function sets the **interval** value in *invvalue* and returns zero. Otherwise, the function returns an error code and the **interval** variable contains an unpredictable value.

The formatting directives **%B**, **%b**, and **%p**, which the **DBTIME** environment variable accepts, are not applicable in *fmtstring* because *month name* and *A.M./P.M.* information is not relevant for intervals of time. Use the **%Y** directive if the **interval** is more than 99 years (**%y** can handle only two digits). For hours, use **%H** (not **%I**, because **%I** can represent only 12 hours). If *fmtstring* is an empty string, the function returns an error.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Example:

The **demo** directory contains this sample program in the file **incvfmtasc.ec**.

```
/* *incvfmtasc.ec*
 The following program illustrates the conversion of two strings
 to three interval values.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    char out_str[30];
    char out_str2[30];
    char out_str3[30];
    mint x;


EXEC SQL BEGIN DECLARE SECTION;
     interval day to minute short_time;
     interval minute(5) to second moment;
     interval hour to second long_moment;
   EXEC SQL END DECLARE SECTION;

  printf("INCVFMTASC Sample ESQL Program running.\n\n");

  /* Initialize short_time */
  printf("Interval value #1 = 20 days, 3 hours, 40 minutes\n");
     x = incvfmtasc("20 days, 3 hours, 40 minutes",
       "%d days, %H hours, %M minutes", &short_time);

  /*Convert the internal format to ascii in ANSI format, for displaying. */
   x = intoasc(&short_time, out_str);
   printf("Interval value (day to minute) = %s\n", out_str);

  /* Initialize moment */
  printf("\nInterval value #2 = 428 minutes, 30 seconds\n");
  x = incvfmtasc("428 minutes, 30 seconds",
      "%M minutes, %S seconds", &moment);

  /* Convert the internal format to ascii in ANSI format, for displaying. */
   x = intoasc(&moment, out_str2);
   printf("Interval value (minute to second) = %s\n", out_str2);

  /* Initialize long_moment */
  printf("\nInterval value #3 = 428 minutes, 30 seconds\n");
   x = incvfmtasc("428 minutes, 30 seconds",
      "%M minutes, %S seconds", &long_moment);

  /*Convert the internal format to ascii in ANSI format, for displaying. */
   x = intoasc(&long_moment, out_str3);
   printf("Interval value (hour to second) = %s\n", out_str3);

  printf("\nINCVFMTASC Sample Program over.\n\n");
}
```

Output:

```
INVCFMTASC Sample ESQL Program running.

Interval value #1 = 20 days, 3 hours, 40 minutes
Interval value (day to minute) = 20 03:40

Interval value #2 = 428 minutes, 30 seconds
Interval value (minute to second) = 428:30

Interval value #3 = 428 minute, 30 seconds
Interval value (hour to second) = 7:08:30

INVCFMTASC Sample Program over.
```

The intoasc() function

The **intoasc()** function converts the field values of an **interval** variable to an ASCII string that conforms to the ANSI SQL standard. For information about the ANSI SQL INTERVAL standard, see 6-7.

Syntax:

```
mint intoasc(invvalue, outbuf)
   intrvl_t *invvalue;
   char *outbuf;
```

invvalue   A pointer to an initialized **interval** variable to convert.

outbuf   A pointer to the buffer that receives the ANSI-standard INTERVAL string for the value in *invvalue*.

Usage:

The **intoasc()** function converts the digits of the fields in the **interval** variable to their character equivalents and copies them to the *outbuf* character string with delimiters (hyphen, space, colon, or period) between them. You must initialize the **interval** variable in *invvalue* with the qualifier that you want the character string to have.

The character string does not include the qualifier or the parentheses that SQL statements use to delimit an INTERVAL literal. The *outbuf* string conforms to ANSI SQL standards. It includes one character for each delimiter (hyphen, space, colon, or period) plus fields with the following sizes.

Field	Field size
Leading field	As specified by precision
Fraction	As specified by precision
All other fields	Two digits

An **interval** value with the **day(5) to fraction(5)** qualifier produces the maximum length of output. The string equivalent contains 16 digits, 4 delimiters, and the null terminator, for a total of 21 bytes:

```
DDDDD HH:MM:SS.FFFFF
```

If you do not initialize the qualifier of the **interval** variable, the **intoasc()** function returns an unpredictable value, but this value does not exceed 21 bytes.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

Example:

The **demo** directory contains this sample program in the file **intoasc.ec**.

```
/*
   * intoasc.ec *

   The following program illustrates the conversion of an interval (intvl_t)
   into an ASCII string.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    mint x;
    char out_str[10];

    EXEC SQL BEGIN DECLARE SECTION;
      interval day(3) to day in1;
    EXEC SQL END DECLARE SECTION;

    printf("INTOASC Sample ESQL Program running.\n\n");

    printf("Interval (day(3) to day) string is '3'\n");
    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("\tInterval value after conversion  is '%s'\n", out_str);
      }

    printf("\nINTOASC Sample Program over.\n\n");
}
```

Output:

```
INTOASC Sample ESQL Program running.

Interval (day(3) to day) string is '3'
Interval value afer conversion  is '   3'

INTOASC Sample Program over.
```

The intofmtasc() function

The **intofmtasc()** function uses a formatting mask to convert an **interval** variable to a character string.

Syntax:

```
mint intofmtasc(invvalue, outbuf, buflen, fmtstring)
   intrvl_t *invvalue;
   char *outbuf;
   mint buflen;
   char *fmtstring;
```

invvalue   A pointer to an initialized **interval** variable to convert.

outbuf	A pointer to the buffer that receives the string for the value in invvalue.
buflen	The length of the outbuf buffer.
fmtstring	A pointer to the buffer that contains the formatting mask to use for the outbuf string. This time-formatting mask contains the same formatting directives that the **DBTIME** environment variable supports. (For a list of these directives, see the description of **DBTIME** in the *IBM Informix Guide to SQL: Reference*).

Usage:

You must initialize the **interval** variable in *invvalue* with the qualifier that you want the character string to have. If you do not initialize the **interval** variable, the function returns an unpredictable value. The character string in *outbuf* does *not* include the qualifier or the parentheses that SQL statements use to delimit an INTERVAL literal.

The formatting mask, *fmtstring*, does not need to imply the same qualifiers as the **interval** variable. When the implied formatting-mask qualifier is different from the **interval** qualifier, **intofmtasc()** converts the result to appropriate units, as necessary (as if it called the **invextend()** function). However, both qualifiers must belong to the same class: either the **year to month** class or the **day to fraction** class.

If *fmtstring* is an empty string, the **intofmtasc()** function sets *outbuf* to an empty string.

The formatting directives **%B**, **%b**, and **%p**, which the **DBTIME** environment variable accepts, are not applicable in *fmtstring* because *month name* and *A.M./P.M.* information is not relevant for intervals of time. Use the **%Y** directive if the **interval** is more than 99 years (**%y** can handle only two digits). For hours, use **%H** (not **%I**, because **%I** can represent only 12 hours). If *fmtstring* is an empty string, the function returns an error.

If the character string and the formatting mask are acceptable, the **incvfmtasc()** function sets the **interval** value in *invvalue* and returns zero. Otherwise, the function returns an error code and the **interval** variable contains an unpredictable value.

Return codes:

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

Example:

The **demo** directory contains this sample program in the file **intofmtasc.ec**.

```
/*
 *intofmtasc.ec*
 The following program illustrates the conversion of interval values
 to ASCII strings with the specified formats.
*/

#include <stdio.h>

EXEC SQL include datetime;
```

```
main()
{
    char out_str[60];
    char out_str2[60];
    char out_str3[60];
    mint x;

    EXEC SQL BEGIN DECLARE SECTION;
      interval day to minute short_time;
      interval minute(5) to second moment;
    EXEC SQL END DECLARE SECTION;

  printf("INTOFMTASC  Sample ESQL Program running.\n\n");

 /* Initialize short_time (day to minute) interval value */
    printf("Interval string #1 = '20 days, 3 hours, 40 minutes'\n");
    x = incvfmtasc("20 days, 3 hours, 40 minutes",
       "%d days, %H hours, %M minutes", &short_time);
  /* Turn the interval into ascii string of a certain format. */
   x = intofmtasc(&short_time, out_str, sizeof(out_str),
      "%d days, %H hours, %M minutes to go!");
   printf("\tFormatted value: %s\n", out_str);

    /* Initialize moment (minute(5) to second interval value */
    printf("\nInterval string #2: '428 minutes, 30 seconds'\n");
    x = incvfmtasc("428 minutes, 30 seconds",
       "%M minutes, %S seconds", &moment);

    /* Turn each interval into ascii string of a certain format. Note
     * that the second and third calls to intofmtasc both use moment
     * as the input variable, but the output strings have different
     * formats.
     */
    x = intofmtasc(&moment, out_str2, sizeof(out_str2),
       "%M minutes and %S seconds left.");
    x = intofmtasc(&moment, out_str3, sizeof(out_str3),
       "%H hours, %M minutes, and %S seconds still left.");

    /* Print each resulting string */
    printf("\tFormatted value: %s\n", out_str2);
    printf("\tFormatted value: %s\n", out_str3);

    printf("\nINTOFMTASC Sample Program over.\n\n");
}
```

Output:

```
INTOFMTASC Sample ESQL Program running.

Interval string #1: '20 days, 3 hours, 40 minutes'
   Formatted value: 20 days, 03 hours, 40 minutes to go!

Interval string #2: '428 minutes, 30 seconds'
   Formatted value: 428 minutes and 30 seconds left.
   Formatted value: 07 hours, 08 minutes, and 30 seconds still left.

INTOFMTASC Sample Program over.
```

The invdivdbl() function

The **invdivdbl()** function divides an **interval** value by a numeric value.

Syntax:

```
mint invdivdbl(iv, num, ov)
   intrvl_t *iv;
   double num;
   intrvl_t *ov;
```

iv   A pointer to an **interval** variable to be divided.

num   A numeric divisor value.

ov   A pointer to an **interval** variable with a valid qualifier.

Usage:

The input and output qualifiers must both belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class. If the qualifier for *ov* is different from the qualifier for *iv* (within the same class), the **invdivdbl()** function extends the result (as the **invextend()** function defines).

The **invdivdbl()** function divides the **interval** value in *iv* by *num* and stores the result in *ov*.

The value in *num* can be either a positive or a negative value.

Return codes:

0   The division was successful.

<0   The division failed.

-1200   A numeric value is too large (in magnitude).

-1201   A numeric value is too small (in magnitude).

-1202   The *num* parameter is zero (0).

-1265   Overflow occurred on an **interval** operation.

-1266   An **interval** value is incompatible with the operation.

-1268   A parameter contains an invalid **interval** qualifier.

Example:

The **demo** directory contains this sample program in the file **invdivdbl.ec**.

```
/*
   * indivdbl.ec *

   The following program divides an INTERVAL type variable by a numeric
   value and stores the result in an INTERVAL variable. The operation is
   done twice, using INTERVALs with different qualifiers to store the     result.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    char out_str[16];

    EXEC SQL BEGIN DECLARE SECTION;
        interval day to second daytosec1;
        interval hour to minute hrtomin;
        interval day to second daytosec2;
```

```
        EXEC SQL END DECLARE SECTION;

        printf("INVDIVDBL Sample ESQL Program running.\n\n");

        /* Input is 3 days, 5 hours, 27 minutes, and 30 seconds */
        printf("Interval (day to second) string = '3 5:27:30'\n");
        incvasc("3 5:27:30", &daytosec1);

        /* Divide input value by 3.0, store in hour to min interval  */
        invdivdbl(&daytosec1, (double) 3.0, &hrtomin);

        /* Convert the internal format to ascii for displaying */
        intoasc(&hrtomin, out_str);
        printf("Divisor (double)                =              3.0  \n");
        printf("----------------------------------------------------\n");
        printf("Quotient #1 (hour to minute)  = '%s'\n", out_str);

        /* Divide input value by 3.0, store in day to sec interval variable */
        invdivdbl(&hrtomin, (double) 3.0, &daytosec2);

        /* Convert the internal format to ascii for displaying */
        intoasc(&daytosec2, out_str);
        printf("Quotient #2 (day to second)    = '%s'\n", out_str);

        printf("\nINVDIVDBL Sample Program over.\n\n");
}
```

Output:

```
INVDIVDBL Sample ESQL Program running.

Interval (day to second) string = '3 5:27:30'
Divisor (double)                =         3.0
--------------------------------------------
Quotient #1 (hour to minute)  = '  25:49'
Quotient #2 (day to second) = '  1 01:49:10'

INVDIVDBL Sample Program over.
```

The invdivinv() function

The **invdivinv()** function divides an **interval** value by another **interval** value.

Syntax:

```
mint invdivinv(i1, i2, num)
   intrvl_t *i1, *i2;
   double *num;
```

i1	A pointer to an **interval** variable that is the dividend.
i2	A pointer to an **interval** variable that is the divisor.
num	A pointer to the **double** value that is the quotient.

Usage:

The **invdivinv()** function divides the **interval** value in *i1* by *i2*, and stores the result in *num*. The result can be either positive or negative.

Both the input and output qualifiers must belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class. If necessary, the **invdivinv()** function extends the **interval** value in *i2* to match the qualifier for *i1* before the division.

Return codes:

0	The division was successful.
<0	The division failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1266	An **interval** value is incompatible with the operation.
-1268	A parameter contains an invalid **interval** qualifier.

Example:

The **demo** directory contains this sample program in the file **invdivinv.ec**.

```
/*
    * invdivinv.ec *

    The following program divides one interval value by another and
    displays the resulting numeric value.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    mint x;
    char out_str[16];

    EXEC SQL BEGIN DECLARE SECTION;
      interval hour to minute hrtomin1, hrtomin2;
      double res;
    EXEC SQL END DECLARE SECTION;

    printf("INVDIVINV Sample ESQL Program running.\n\n");

    printf("Interval #1 (hour to minute) = 75:27\n");
    incvasc("75:27", &hrtomin1);
    printf("Interval #2 (hour to minute) = 19:10\n");
    incvasc("19:10", &hrtomin2);

printf("--------------------------------------------\n");
    invdivinv(&hrtomin1, &hrtomin2, &res);
    printf("Quotient (double)              =  %.1f\n", res);

    printf("\nINVDIVINV Sample Program over.\n\n");
}
```

Output:

```
INVDIVINV Sample ESQL Program running.

Interval #1 (hour to minute) = 75.27
Interval #2 (hour to minute) = 19:10
```

```
------------------------------------
Quotient (double)          =  3.9

INVDIVINV Sample Program over.
```

The invextend() function

The **invextend()** function copies an **interval** value under a different qualifier. Extending is the operation of adding or dropping fields of an INTERVAL value to make it match a given qualifier. For INTERVAL values, both qualifiers must belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class.

Syntax:
```
mint invextend(in_inv, out_inv)
   intrvl_t *in_inv, *out_inv;
```

in_inv   A pointer to the **interval** variable to extend.

out_inv   A pointer to the **interval** variable with a valid qualifier to use for the extension.

Usage:

The **invextend()** function copies the qualifier-field digits of *in_inv* **interval** variable to the *out_inv* **interval** variable. The qualifier of the *out_inv* variable controls the copy.

The function discards any fields in *in_inv* that are to the right of the least-significant field in *out_inv*. The function fills in any fields in *out_inv* that are not present in *in_inv* as follows:
- It fills the fields to the right of the least-significant field in *in_inv* with zeros.
- It sets the fields to the left of the most-significant field in *in_inv* to valid **interval** values.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

-1266   An **interval** value is incompatible with the operation.

-1268   A parameter contains an invalid **interval** qualifier.

Example:

The **demo** directory contains this sample program in the file **invextend.ec**. The example illustrates **interval** extension. In the second result, the output contains zeros in the **seconds** field, and the **days** field has been set to 3.

```
/*
   * invextend.ec *

   The following program illustrates INTERVAL extension. It extends an INTERVAL
   value to another INTERVAL value with a different qualifier. Note that in the
   second example, the output contains zeros in the seconds field and the
   days field has been set to 3.
*/

#include <stdio.h>
```

```
                EXEC SQL include datetime;

                main()
                {
                    mint x;
                    char out_str[16];
                ;
                    EXEC SQL BEGIN DECLARE SECTION;
                      interval hour to minute hrtomin;
                      interval hour to hour hrtohr;
                      interval day to second daytosec;
                    EXEC SQL END DECLARE SECTION;

                    printf("INVEXTEND Sample ESQL Program running.\n\n");

                    printf("Interval (hour to minute) value =      75.27\n");
                    incvasc("75:27", &hrtomin);

                    /* Extend to hour-to-hour and convert the internal format to
                     * ascii for displaying
                     */
                    invextend(&hrtomin, &hrtohr);
                    intoasc(&hrtohr, out_str);
                    printf("Extended (hour to hour) value =    %s\n", out_str);

                    /* Extend to day-to-second and convert the internal format to
                     * ascii for displaying

                     */
                    invextend(&hrtomin, &daytosec);
                    intoasc(&daytosec, out_str);
                    printf("Extended (day to second) value =: %s\n", out_str);

                    printf("\nINVEXTEND Sample Program over.\n\n");
                }
```

Output:

```
INVEXTEND Sample ESQL Program running.

Interval (hour to minute) value =    75:27
Extended (hour to hour) value   =    75
Extended (day to second) value  =  3 03:27:00

INVEXTEND Sample Program over.
```

The invmuldbl() function

The **invmuldbl()** function multiplies an **interval** value by a numeric value.

Syntax:

```
mint invmuldbl(iv, num, ov)
    intrvl_t *iv;
    double num;
    intrvl_t *ov;
```

iv   A pointer to the **interval** variable to multiply.

num   The numeric **double** value.

ov   A pointer to the **interval** variable with a valid qualifier.

Usage:

The **invmuldbl()** function multiplies the **interval** value in *iv* by *num* and stores the result in *ov*. The value in *num* can be either positive or negative.

Both the input and output qualifiers must belong to the same **interval** class: either the **year to month** class or the **day to fraction(5)** class. If the qualifier for *ov* is different from the qualifier for *iv* (but of the same class), the **invmuldbl()** function extends the result (as the **invextend()** function defines).

Return codes:

0	The multiplication was successful.
<0	The multiplication failed.
-1200	A numeric value is too large (in magnitude).
-1201	A numeric value is too small (in magnitude).
-1266	An **interval** value is incompatible with the operation.
-1268	A parameter contains an invalid **interval** qualifier.

Example:

The **demo** directory contains this sample program in the file **invmuldbl.ec**. The example illustrates how to multiply an **interval** value by a numeric value. The second multiplication illustrates the result of **interval** multiplication when the input and output qualifiers are different.

```
/*
   * invmuldbl.ec *

   The following program multiplies an INTERVAL type variable by a numeric value
   and stores the result in an INTERVAL variable. The operation is done twice,
   using INTERVALs with different qualifiers to store the result.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    char out_str[16];

    EXEC SQL BEGIN DECLARE SECTION;
       interval hour to minute hrtomin1;
       interval hour to minute hrtomin2;
       interval day to second daytosec;
    EXEC SQL END DECLARE SECTION;

    printf("INVMULDBL Sample ESQL Program running.\n\n");


    /* input is 25 hours, and 49 minutes */
    printf("Interval (hour to minute)      =      25:49\n");
    incvasc("25:49", &hrtomin1);
    printf("Multiplier (double)                  =          3.0\n");
    printf("-----------------------------------------------------\n");

    /* Convert the internal format to ascii for displaying */
    invmuldbl(&hrtomin1, (double) 3.0, &hrtomin2);
    intoasc(&hrtomin2, out_str);
    printf("Product #1 (hour to minute)  =      '%s'\n", out_str);
```

```
          /* Convert the internal format to ascii for displaying */
          invmuldbl(&hrtomin1, (double) 3.0, &daytosec);
          intoasc(&daytosec, out_str);
          printf("Product #2 (day to second)    = '%s'\n", out_str);

          printf("\nINVMULDBL Sample Program over.\n\n");
}
```

Output:

```
INVMULDBL Sample ESQL Program running.

Interval (hour to minute) =      25:49
Multiplier (double) =         3.0
---------------------------------------------
Product #1 (hour to minute)  =    ' 77:27'
Product #2 (day to second)   = '  3 05:27:00'

INVMULDBL Sample Program over.
```

The ldchar() function

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

Syntax:

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

from	A pointer to the fixed-length source string.
count	The number of bytes in the fixed-length source string.
to	A pointer to the first byte of a null-terminated destination string. The to argument can point to the same location as the *from* argument, or to a location that overlaps the *from* argument. If this is the case, **ldchar()** does not preserve the value to which *from* points.

Example:

This sample program is in the **ldchar.ec** file in the **demo** directory.

```
/*
   * 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];

    printf("LDCHAR Sample ESQL Program running.\n\n");
```

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

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

        printf("\nLDCHAR Sample Program over.\n\n");
}
```

Output:

```
LDCHAR Sample ESQL Program running.

    Source: [abcd   ]
    Destination: [abcd]

    Source: [abcd  g ]
    Destination: [abcd  g]

LDCHAR Sample Program over.
```

The rdatestr() function

The **rdatestr()** function converts an internal DATE to a character string.

Syntax:

```
mint rdatestr(jdate, outbuf)
    int4 jdate;
    char *outbuf;
```

jdate   The internal representation of the date to format.

outbuf   A pointer to the buffer that receives the string for the jdate value.

Usage:

For the default locale, U.S. English, the **rdatestr()** function determines how to interpret the format of the character string with the following precedence:

1. The format that the **DBDATE** environment variable specifies (if **DBDATE** is set). For more information about **DBDATE**, see the *IBM Informix Guide to SQL: Reference*.
2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set). For more information about **GL_DATE**, see the *IBM Informix GLS User's Guide*.
3. The default date form: mm/dd/yyyy.

When you use a nondefault locale and do not set the **DBDATE** or **GL_DATE** environment variable, **rdatestr()** uses the date end-user format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide*.

Return codes:

0   The conversion was successful.

<0   The conversion failed.

-1210   The internal date could not be converted to the character string format.

-1212	Data conversion format must contain a month, day, or year component. **DBDATE** specifies the data conversion format.

Example:

The **demo** directory contains this sample program in the **rtoday.ec** file.

```
/*
   * 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()
{
   mint errnum;
   char today_date[20];
   int4 i_date;

   printf("RTODAY Sample ESQL Program running.\n\n");

    /* 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);

   printf("\nRTODAY Sample Program over.\n\n");
}
```

Output:
```
RTODAY Sample ESQL Program running.

   Today's date is 10/26/2007.

RTODAY Sample Program over.
```

The rdayofweek() function

The **rdayofweek()** function returns the day of the week as an integer value for an internal DATE.

Syntax:
```
mint rdayofweek(jdate)
   int4 jdate;
```

jdate	The internal representation of the date.

Return codes:

0	Sunday
1	Monday
2	Tuesday
3	Wednesday

4	Thursday
5	Friday
6	Saturday

Example:

The **demo** directory contains this sample program in the **rdayofweek.ec** file.

```
/*
   * rdayofweek.ec *

   The following program accepts a date entered from the console.
*/

#include <stdio.h>

main()
{
   mint errnum;
   int4 i_date;
   char *day_name;
   char date[20];
   int x;

   static char fmtstr[9] = "mmddyyyy";

   printf("RDAYOFWEEK Sample ESQL Program running.\n\n");

   /* Allow user to enter a date */
   printf("Enter a date as a single string, month.day.year\n");
   gets(date);

   printf("\nThe date string is %s.\n", date);

   /* Put entered date in internal format */
   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("This date is a %s.\n", day_name);
      }

   printf("\nRDAYOFWEEK Sample Program over.\n\n");
}
```

Output:

```
RDAYOFWEEK Sample ESQL Program running.

Enter a date as a single string, month.day.year
10.13.07

The date string is 10.13.07.
This date is a Saturday.

RDAYOFWEEK Sample Program over.
```

The rdefmtdate() function

The **rdefmtdate()** function uses a formatting mask to convert a character string to an internal DATE format.

Syntax:

```
mint rdefmtdate(jdate, fmtstring, inbuf)
    int4 *jdate;
    char *fmtstring;
    char *inbuf;
```

jdate	A pointer to a **int4** integer value that receives the internal DATE value for the *inbuf* string.
fmtstring	A pointer to the buffer that contains the formatting mask to use the inbuf string.
inbuf	A pointer to the buffer that contains the date string to convert.

Usage:

The *fmtstring* argument of the **rdefmtdate()** function points to the date-formatting mask, which contains formats that describe how to interpret the date string. For more information about these date formats, see "Formatting date strings" on page 6-1

The *input* string and the *fmtstring* must be in the same sequential order in terms of month, day, and year. They need not, however, contain the same literals or the same representation for month, day, and year.

You can include the weekday format (ww), in *fmtstring*, but the database server ignores that format. Nothing from the *inbuf* corresponds to the weekday format.

The following combinations of *fmtstring* and *input* are valid.

Formatting mask	Input
mmddyy	Dec. 25th, 2007
mmddyyyy	Dec. 25th, 2007
mmm. dd. yyyy	dec 25 2007
mmm. dd. yyyy	DEC-25-2007
mmm. dd. yyyy	122507
mmm. dd. yyyy	12/25/07
yy/mm/dd	07/12/25
yy/mm/dd	2007, December 25th

| yy/mm/dd | In the year 2007, the month of December, it is the 25th day |
| dd-mm-yy | This 25th day of December, 2007 |

If the value stored in *inbuf* is a four-digit year, the **rdefmtdate()** function uses that value. If the value stored in *inbuf* is a two-digit year, the **rdefmtdate()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, Informix ESQL/C uses the 20th century. For information about how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *fmtstring* argument of **rdefmtdate()**. For more information, see the *IBM Informix GLS User's Guide*.

Return codes:

If you use an invalid date-string format, **rdefmtdate()** returns an error code and sets the internal DATE to the current date. The following are possible return codes.

0	The operation was successful.
-1204	The *input* parameter specifies an invalid year.
-1205	The *input* parameter specifies an invalid month.
-1206	The *input* parameter specifies an invalid day.
-1209	Because *input* does not contain delimiters between the year, month, and day, the length of *input* must be exactly six or eight bytes.
-1212	*fmtstring* does not specify a year, a month, and a day.

Example:

The **demo** directory contains this sample program in the **rdefmtdate.ec** file.

```
/*
 * 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()
{
    mint x;
    char date[20];
    int4 i_date;
    char *day_name;

    static  char fmtstr[9] = "mmddyyyy";

    printf("RDEFMTDATE Sample ESQL Program running.\n\n");

    printf("Enter a date as a single string, month.day.year\n");
    gets(date);
```

```
        printf("\nThe date string 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);
          }

      printf("\nRDEFMTDATE Sample Program over.\n\n");
}
```

Output:

```
RDEFMTDATE Sample ESQL Program running.

Enter a date as a single string, month.day.year
080894

The date string is 080894
The day of the week is Monday.

RDEFMTDATE Sample Program over.
```

The rdownshift() function

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

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

s   A pointer to a null-terminated string.

Usage:

The **rdownshift()** function refers to the current locale to determine uppercase and
lowercase letters. For the default locale, U.S. English, **rdownshift()** uses the ASCII
lowercase (a-z) and uppercase (A-Z) letters.

If you use a nondefault locale, **rdownshift()** uses the lowercase and uppercase
letters that the locale defines. For more information, see the *IBM Informix GLS
User's Guide*.

Return codes:

This sample program is in the **rdownshift.ec** file in the **demo** directory.

```
/*
   * rdownshift.ec *

   The following program uses rdownshift() on a string containing
   alphanumeric and punctuation characters.
*/

#include <stdio.h>

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

    printf("RDOWNSHIFT Sample ESQL Program running.\n\n");

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

    printf("\nRDOWNSHIFT Sample Program over.\n\n");
}
```

Output:

```
RDOWNSHIFT Sample ESQL Program running.

    Input string...: [123ABCDEFGHIJK'.;]
    After downshift: [123abcdefghijk'.;]

RDOWNSHIFT Sample Program over.
```

The ReleaseConnect() function (Windows)

The **ReleaseConnect()** function is available only in Windows environments. It releases, or terminates, the explicit connection and clears all allocated memory.

Important: Informix ESQL/C supports the **ReleaseConnect()** connection library function for compatibility with Version 5.01 Informix ESQL/C for Windows applications. When you write new Informix ESQL/C applications for Windows environments, use the SQL DISCONNECT statement to terminate an established explicit connection.

Syntax:

```
void *ReleaseConnect ( void *CnctHndl )
```

CnctHndl   A connection handle returned by a previous **GetConnect()** call.

Usage:

The **ReleaseConnect()** function maps to a simple SQL DISCONNECT statement (one *without* an ALL, CURRENT, or DEFAULT option). The **ReleaseConnect()** call *by itself* is equivalent to the following SQL statement:

```
EXEC SQL disconnect db_connection;
```

In this example, *db_connection* is the name of an existing connection that the
GetConnect() function has established. You pass this *db_connection* name to
ReleaseConnect() as an argument; it is a connection handle for the desired
connection.

For example, the following code fragment uses **ReleaseConnect()** to close an
explicit connection to the **stores7** database on the default database server:

```
void *cnctHndl;
.
.
.

cnctHndl = GetConnect();
EXEC SQL database stores7;
.
.
.

EXEC SQL close database;
cnctHndl = ReleaseConnect( cnctHndl );
```

Call **ReleaseConnect()** once for each connection that **GetConnect()** has established.
The **ReleaseConnect()** function closes any open database before it terminates the
current connection. It fails if any transactions are open in the current connection.

It is good programming practice to close the database explicitly with the SQL
CLOSE DATABASE statement before the call to **ReleaseConnect()**.

Important: Because the **ReleaseConnect()** function maps to a DISCONNECT
statement, it sets the SQLCODE and SQLSTATE status codes to
indicate the success or failure of the connection termination request.
This behavior differs from that of **ReleaseConnect()** in Version 5.01
Informix ESQL/C for Windows, in which this function did not set the
SQLCODE and SQLSTATE values.

The **ReleaseConnect()** function differs from the DISCONNECT statement in the
way that it obtains the connection name. **ReleaseConnect()** uses an internally
generated name that is stored in the connection handle; you must specify this
handle as an argument to the **ReleaseConnect()** call. The DISCONNECT statement
uses an internally generated connection name only for a connection that a
CONNECT statement without an AS clause has established; if the connection has a
user-defined connection name (which the AS clause of the CONNECT statement
specifies), DISCONNECT uses this name.

Return codes:

CnctHndl   The call to **ReleaseConnect()** was successful if the function has
returned a connection handle that matches the one passed to it.

The rfmtdate() function

The **rfmtdate()** function uses a formatting mask to convert an internal DATE
format to a character string.

Syntax:

```
mint rfmtdate(jdate, fmtstring, outbuf)
   int4 jdate;
   char *fmtstring;
   char *outbuf;
```

jdate   The internal representation of a date to convert.

fmtstring	A pointer to the buffer that contains the formatting mask to use the jdate value.
outbuf	A pointer to the buffer that receives the formatted string for the jdate value.

Usage:

The *fmtstring* argument of the **rfmtdate()** function points to the date-formatting mask, which contains formats that describe how to format the date string. For more information about these date formats, see "Formatting Numeric Strings" on page 5-9.

The examples in the following list use the formatting mask in *fmtstring* to convert the integer *jdate*, whose value corresponds to December 25, 2007, to a formatted string *outbuf*. You must specify one or more fields.

Formatting mask	Formatted tesult
"mmdd"	1225
"mmddyy"	122507
"ddmmyy"	251207
"yydd"	0725
"yymmdd"	071225
"dd"	25
"yy/mm/dd"	07/12/25
"yy mm dd"	07 12 25
"yy-mm-dd"	07-12-25
"mmm. dd, yyyy"	Dec. 25, 2007
"mmm dd yyyy"	Dec 25 2007
"yyyy dd mm"	2007 25 12
"mmm dd yyyy"	Dec 25 2007
"ddd, mmm. dd, yyyy"	Tue, Dec. 25, 2007
"ww mmm. dd, yyyy"	Tue Dec. 25, 2007
"(ddd) mmm. dd, yyyy"	(Tue) Dec. 25, 2007
"mmyyddmm"	25071225
""	unpredictable result

When you use a nondefault locale whose dates contain eras, you can use extended-format strings in the *fmtstring* argument of **rfmtdate()**. For more information, see the *IBM Informix GLS User's Guide*.

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).
-1212	Format string is NULL or invalid.

Example:

The **demo** directory contains this sample program in the **rfmtdate.ec** file.

```
/*
   * rfmtdate.ec *

   The following program converts a date from internal format to
   a specified format using rfmtdate().
*/

#include <stdio.h>

main()
{
   char the_date[15];
   int4 i_date;
   mint x;
   int errnum;
   static short mdy_array[3] = { 12, 10, 2007 };

    printf("RFMTDATE Sample ESQL Program running.\n\n");

   if ((errnum = rmdyjul(mdy_array, &i_date)) == 0)
      {

      /*
       * Convert date to "mm-dd-yyyy" format
       */
      if (x = rfmtdate(i_date, "mm-dd-yyyy", the_date))
         printf("First rfmtdate() call failed with error %d\n", x);
      else
         printf("\tConverted date (mm-dd-yyy): %s\n", the_date);

      /*
       * Convert date to "mm.dd.yy" format
       */
      if (x = rfmtdate(i_date, "mm.dd.yy", the_date))
         printf("Second rfmtdate() call failed with error %d\n",x);
      else
         printf("\tConverted date (mm.dd.yy): %s\n", the_date);

      /*
       * Convert date to "mmm ddth, yyyy" format
       */
      if (x = rfmtdate(i_date, "mmm ddth, yyyy", the_date))
         printf("Third rfmtdate() call failed with error %d\n", x);
       else
         printf("\tConverted date (mmm ddth, yyyy): %s\n", the_date);
      }

   printf("\nRFMTDATE Sample Program over.\n\n");
}
```

Output:

```
RFMTDATE Sample ESQL Program running.

   Converted date (mm-dd-yyy): 12-10-2007.
   Converted date (mm.dd.yy): 12.10.07.
   Converted date (mmm ddth, yyyy): Dec 10th, 2007

RFMTDATE Sample Program over.
```

The rfmtdec() function

The **rfmtdec()** function uses a formatting mask to convert a **decimal** value to a character string.

Syntax:

```
mint rfmtdec(dec_val, fmtstring, outbuf)
   dec_t *dec_val;
   char *fmtstring;
   char *outbuf;
```

dec_val	A pointer to the **decimal** value to format.
fmtstring	A pointer to a character buffer that contains the formatting mask to use for the dec_val value.
outbuf	A pointer to a character buffer that receives the formatted string for the dec_val value.

Usage:

The *fmtstring* argument of the **rfmtdec()** function points to the numeric-formatting mask, which contains characters that describe how to format the **decimal** value. For more information about these formatting characters, see "Formatting Numeric Strings" on page 5-9.

When you use **rfmtdec()** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, **rfmtdec()** uses the currency symbols that the client locale defines. The default locale, U.S. English, defines currency symbols as if you set **DBMONEY** to "**$,.**". (For a discussion of **DBMONEY**, see the *IBM Informix Guide to SQL: Reference*.)

When you use a nondefault locale that has a multibyte code set, **rfmtdec()** supports multibyte characters in the format string. For more information, see the *IBM Informix GLS User's Guide*.

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:

The **demo** directory contains this sample program in the file **rfmtdec.ec**.

```
/*
 * 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>

EXEC SQL include decimal;

char *strings[] =
```

```
        {
        "210203.204",
        "4894",
        "443.334899312",
        "-12344455",
        0
        };

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

char result[41];

main()
{
    mint x;
    mint s = 0, f;
    dec_t num;

    printf("RFMTDEC Sample ESQL Program running.\n\n");

    while(strings[s])
      {
      /*
       *  Convert each string to DECIMAL
       */
      printf("String = %s\n", strings[s]);
      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 DECIMAL num for each of formats[f]
          */
         rfmtdec(&num, formats[f], result);
         /*
          * Display result and bump to next format (f++)
          */
         result[40] = '\0';
         printf("  Format String = '%s'\t", formats[f++]);
         printf("\tResult = '%s'\n", result);
         }
      ++s;                                      /* bump to next string */
      printf("\n");                     /* separate result groups */
      }

    printf("\nRFMTDEC Sample Program over.\n\n");
}
```

Output:

```
RFMTDEC Sample ESQL Program running.

String = 210203.204
  Format String = '**##########'            Result = '**     210203'
  Format String =  '$$$$$$$$$$.##'          Result = '   $210203.20'
  Format String =  '(&&,&&&,&&&.)'          Result = ' 000210,203. '
  Format String =  '<,<<<,<<<,<<<'          Result = '210,203'
  Format String = ' $*********.**'          Result = '$***210203.20'

String = 4894
  Format String = '**##########'            Result = ' **       4894'
  Format String =  '$$$$$$$$$$.##'          Result = '    $4894.00'
  Format String =  '(&&,&&&,&&&.)'          Result = ' 000004,894. '
  Format String =  '<,<<<,<<<,<<<'          Result = '4,894'
  Format String = ' $*********.**'          Result = '$*****4894.00'

String = 443.334899312
  Format String = '**##########'            Result = ' **        443'
  Format String =  '$$$$$$$$$$.##'          Result = '     $443.33'
  Format String =  '(&&,&&&,&&&.)'          Result = '  0000000443. '
  Format String =  '<,<<<,<<<,<<<'          Result = ' 443'
  Format String = ' $*********.**'          Result = '$******443.33'

String = -12344455
  Format String = '**##########'            Result = ' **   12344455'
  Format String =  '$$$$$$$$$$.##'          Result = '  $12344455.00'
  Format String =  '(&&,&&&,&&&.)'          Result = '(12,344,455.)'
  Format String =  '<,<<<,<<<,<<<'          Result = '12,344,455'
  Format String = ' $*********.**'          Result = ' $*12344455.00'

RFMTDEC Sample Program over.
```

The rfmtdouble() function

The **rfmtdouble()** function uses a formatting mask to convert a **double** value to a character string.

Syntax:

```
mint rfmtdouble(dbl_val, fmtstring, outbuf)
   double dbl_val;
   char *fmtstring;
   char *outbuf;
```

dbl_val   The **double** number to format.

fmtstring   A pointer to a character buffer that contains the formatting mask
               for the value in dbl_val.

outbuf   A pointer to a character buffer that receives the formatted string
               for the value in dbl_val.

Usage:

The *fmtstring* argument of the **rfmtdouble()** function points to the numeric-formatting mask, which contains characters that describe how to format the **double** value. For more information about these formatting characters, see "Formatting Numeric Strings" on page 5-9.

When you use **rfmtdouble()** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, **rfmtdouble()** uses the currency symbols that the

client locale defines. The default locale, U.S. English, defines currency symbols as if you set **DBMONEY** to "**$,.**". (For a discussion of **DBMONEY**, see the *IBM Informix Guide to SQL: Reference*.)

When you use a nondefault locale that has a multibyte code set, **rfmtdouble()** supports multibyte characters in the format string. For more information, see the *IBM Informix GLS User's Guide*.

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:

The **demo** directory contains this sample program in the file **rfmtdouble.ec**.

```
/*
   * rfmtdouble.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[41];

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

    printf("RFMTDOUBLE Sample ESQL Program running.\n\n");

    while(dbls[i])                          /* for each number in dbls */
      {
      printf("Double Number = %g\n", dbls[i]);
      f = 0;
      while(formats[f])                 /* format with each of formats[] */
        {
```

```
            if (x = rfmtdouble(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++)
             */
            result[40] = '\0';
            printf("  Format String = '%s'\t", formats[f++]);
            printf("\tResult = '%s'\n", result);
            }
        ++i;                            /* bump to next double */
        printf("\n");                   /* separate result groups */
        }
    printf("\nRFMTDOUBLE Sample Program over.\n\n");
}
```

Output:

```
RFMTDOUBLE Sample ESQL Program running.

Double Number = 210203
  Format String = '############'        Result = '      210203'
  Format String = '<,<<<,<<<,<<<'       Result = '210,203'
  Format String = '$$$$$$$$$$.##'       Result = '   $210203.20'
  Format String = '(&&,&&&,&&&.)'       Result = '  000210,203. '
  Format String = '$*********.**'        Result = '$***210203.20'

Double Number = 4894
  Format String = '############'        Result = '        4894'
  Format String = '<,<<<,<<<,<<<'       Result = '4,894'
  Format String = '$$$$$$$$$$.##'       Result = '     $4894.00'
  Format String = '(&&,&&&,&&&.)'       Result = ' 000004,894. '
  Format String = '$*********.**'        Result = '$*****4894.00'

Double Number = 443.335
  Format String = '############'        Result = '         443'
  Format String = '<,<<<,<<<,<<<'       Result = '443'
  Format String = '$$$$$$$$$$.##'       Result = '      $443.33'
  Format String = '(&&,&&&,&&&.)'       Result = '  0000000443. '
  Format String = '$*********.**'        Result = '$******443.33'

Double Number = -1.23445e+07
  Format String = '############'        Result = '    12344455'
  Format String = '<,<<<,<<<,<<<'       Result = '12,344,455'
  Format String = '$$$$$$$$$$.##'       Result = '  $12344455.00'
  Format String = '(&&,&&&,&&&.)'       Result = '(12,344,455.)'
  Format String = '$*********.**'        Result = '$*12344455.00'

RFMTDOUBLE Sample Program over.
```

The rfmtlong() function

The **rfmtlong()** function uses a formatting mask to convert a C **long** value to a character string.

Syntax:

```
mint rfmtlong(lng_val, fmtstring, outbuf)
    int4 lng_val;
    char *fmtstring;
    char *outbuf;
```

lng_val   The **int4** integer that **rmtlong()** converts to character value.

fmtstring   A pointer to a character buffer that contains the formatting mask for the value in lng_val.

outbuf   A pointer to a character buffer that receives the formatted string for the value in lng_val.

Usage:

The *fmtstring* argument of the **rfmtlong()** function points to the numeric-formatting mask, which contains characters that describe how to format the **long integer** value. For more information about these formatting characters, see "Formatting Numeric Strings" on page 5-9.

When you use **rfmtlong()** to format MONEY values, the function uses the currency symbols that the **DBMONEY** environment variable specifies. If you do not set this environment variable, **rfmtlong()** uses the currency symbols that the client locale defines. The default locale, U.S. English, defines currency symbols as if you set **DBMONEY** to "**$,.**". (For a discussion of **DBMONEY**, see the *IBM Informix Guide to SQL: Reference*.)

When you use a nondefault locale that has a multibyte code set, **rfmtlong()** supports multibyte characters in the format string. For more information, see the *IBM Informix GLS User's Guide*.

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:

The **demo** directory contains this sample program in the file **rfmtlong.ec**.

```
/*
   * 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[] =
   {
   21020304,
   334899312,
   -334899312,
   -12344455,
   0
   };

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

```
   "$************.**",
   0
   };

char result[41];

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

    printf("RFMTLONG Sample ESQL Program running.\n\n");

    while(lngs[s])                          /* for each long in lngs[] */
      {
      printf("Long Number = %d\n", lngs[s]);
      f = 0;
      while(formats[f])                     /* format with each of formats[] */
          {
          if (x = rfmtlong(lngs[s], formats[f], result))
              {
              printf("Error %d in formatting %d using %s.\n",
                  x, lngs[s], formats[f]);
              break;
              }
          /*
           * Display result and bump to next format (f++)
           */
          result[40] = '\0';
          printf("  Format String = '%s'\t", formats[f++]);
          printf("\tResult = '%s'\n",  result);
          }
      ++s;                                  /* bump to next long */
      printf("\n");                         /* separate display groups */
      }

    printf("\nRFMTLONG Sample Program over.\n\n");
}
```

Output:

```
RFMTLONG ESQL Sample Program running.

Long Number = 21020304
  Format String = '###############'           Result = '       21020304'
  Format String = '$$$$$$$$$$$$$.##'           Result = '    $21020304.00'
  Format String = '(&,&&&,&&&,&&&.)'           Result = ' 00021,020,304. '
  Format String = '<<<<,<<<,<<<,<<<'           Result = '21,020,304'
  Format String = '$************.**'           Result = '$****21020304.00'

Long Number = 334899312
  Format String = '###############'           Result = '      334899312'
  Format String = '$$$$$$$$$$$$$.##'           Result = '   $334899312.00'
  Format String = '(&,&&&,&&&,&&&.)'           Result = ' 00334,899,312. '
  Format String = '<<<<,<<<,<<<,<<<'           Result = '334,899,312'
  Format String = '$************.**'           Result = '$***334899312.00'

Long Number = -334899312
  Format String = '###############'           Result = '      334899312'
  Format String = '$$$$$$$$$$$$$.##'           Result = '   $334899312.00'
  Format String = '(&,&&&,&&&,&&&.)'           Result = '(00334,899,312.)'
  Format String = '<<<<,<<<,<<<,<<<'           Result = '334,899,312'
  Format String = '$************.**'           Result = '$***334899312.00'
```

```
Long Number = -12344455
  Format String = '###############'            Result = '        12344455'
  Format String = '$$$$$$$$$$$$$.##'           Result = '    $12344455.00'
  Format String = '(&,&&&,&&&,&&&.)'           Result = '(00012,344,455.)'
  Format String = '<<<<,<<<,<<<,<<<'           Result = '12,344,455'
  Format String = '$************.**'           Result = ' $****12344455.00'

RFMTLONG Sample Program over.
```

The rgetlmsg() function

The **rgetlmsg()** function retrieves the corresponding error message for a given Informix-specific error number. The **rgetlmsg()** function allows for error numbers in the range of a **long** integer.

Syntax:

```
mint rgetlmsg(msgnum, msgstr, lenmsgstr, msglen)
   int4  msgnum;
   char *msgstr;
   mint lenmsgstr;
   mint *msglen;
```

msgnum	The error number. The four-byte parameter provides for the full range of Informix-specific error numbers.
msgstr	A pointer to the buffer that receives the message string (the output buffer).
lenmsgstr	The size of the msgstr output buffer. Make this value the size of the largest message that you expect to retrieve.
msglen	A pointer to the **mint** that contains the actual length of the message that **rgetlmsg()** returns.

Usage:

The *msgnum* error number is typically the value of **SQLCODE** (or **sqlca.sqlcode**). You can also retrieve message text for ISAM errors (in **sqlca.sqlerrd[1]**). The **rgetlmsg()** function uses the IBM Informix error message files (in the **$INFORMIXDIR/msg** directory) for error message text.

The **rgetlmsg()** function returns the actual size of the message that you request in the fourth parameter, *msglen*. You can use this value to adjust the size of the message area if it is too small. If the returned message is longer than the buffer that you provide, the function truncates the message. You can also use the *msglen* value to display only that portion of the *msgstr* message buffer that contains error text.

Return codes:

0	The conversion was successful.
-1227	Message file not found.
-1228	Message number not found in message file.
-1231	Cannot seek within message file.
-1232	Message buffer too small.

For error message documentation, use the **finderr** or **Error Message** utility or view *IBM Informix Error Messages* at the IBM Informix information center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

Example:

This sample program is in the **rgetlmsg.ec** file in the Informix ESQL/C **demo** directory.

```
/*
 * rgetlmsg.ec *
 *
 * The following program demonstrates the usage of rgetlmsg() function.
 * It displays an error message after trying to create a table that
 * already exists.
 */
EXEC SQL include sqlca; /* this include is optional */

main()
{
    mint msg_len;
    char errmsg[400];

    printf("\nRGETLMSG Sample ESQL Program running.\n\n");
    EXEC SQL connect to 'stores7';

    EXEC SQL create table customer (name char(20));

    if(SQLCODE != 0)
       {
       rgetlmsg(SQLCODE, errmsg, sizeof(errmsg), &msg_len);
       printf("\nError %d: ", SQLCODE);
       printf(errmsg, sqlca.sqlerrm);
       }
    printf("\nRGETLMSG Sample Program over.\n\n");
}
```

This example uses the error message parameter in **sqlca.sqlerrm** to display the name of the table. This use of **sqlca.sqlerrm** is valid because the error message contains a format parameter that **printf()** recognizes. If the error message did not contain the format parameter, no error would result.

Output:

```
RGETLMSG Sample ESQL Program running.


Error -310: Table (informix.customer) already exists in database.

RGETLMSG Sample Program over.
```

The rgetmsg() function

The **rgetmsg()** function retrieves the error message text for a given Informix-specific error number. The **rgetmsg()** function can handle a **short** error number and, therefore, can only handle error numbers in the range of -32768 to +32767. For this reason, use the **rgetlmsg()** function in all new IBM Informix ESQL/C code.

Syntax:

```
mint rgetmsg(msgnum, msgstr, lenmsgstr)
   mint  msgnum;
   char *msgstr;
   mint lenmsgstr;
```

msgnum   The error number. The two-byte parameter restricts error numbers to between -32768 and +32767.

msgstr   A pointer to the buffer that receives the message string (the output buffer).

lenmsgstr   The size of the *msgstr* output buffer. Make this value the size of the largest message that you expect to retrieve.

Usage:

Typically **SQLCODE** (**sqlca.sqlcode**) contains the error number. You can also retrieve message text for ISAM errors (in **sqlca.sqlerrd[1]**). The **rgetmsg()** function uses the Informix error message files (in the **$INFORMIXDIR/msg** directory) for error message text. If the message is longer than the size of the buffer that you provide, the function truncates the message to fit.

Important: Informix ESQL/C supports the **rgetmsg()** function for compatibility with earlier versions. Some Informix error numbers currently exceed the range that the **short** integer, **msgnum**, supports. The **rgetlmsg()** function, which supports **long** integers as error numbers, is recommended over **rgetmsg()**.

If your program passes the value in the **SQLCODE** variable (or **sqlca.sqlcode**) directly as *msgnum*, cast the **SQLCODE** value as a **short** data type. The *msgnum* argument of **rgetmsg()** has a **short** data type while the **SQLCODE** value is has a **long** data type.

Return codes:

0   The conversion was successful.

-1227   Message file not found.

-1228   Message number not found in message file.

-1231   Cannot seek within message file.

-1232   Message buffer too small.

For error message documentation, use the **finderr** or **Error Message** utility or view *IBM Informix Error Messages* at the IBM Informix information center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

Example:

This sample program is in the **rgetmsg.ec** file in the Informix ESQL/C **demo** directory.

```
/*
 * rgetmsg.ec *
 *
 * The following program demonstrates the usage of the rgetmsg() function.
 * It displays an error message after trying to create a table that already
 * exists.
 */
EXEC SQL include sqlca; /* this include is optional */
```

```
main()
{
    char errmsg[400];

    printf("\nRGETMSG Sample ESQL Program running.\n\n");
    EXEC SQL connect to 'stores7';

    EXEC SQL create table customer (name char(20));
    if(SQLCODE != 0)
      {
      rgetmsg((short)SQLCODE, errmsg, sizeof(errmsg));
      printf("\nError %d: ", SQLCODE);
      printf(errmsg, sqlca.sqlerrm);
      }
    printf("\nRGETMSG Sample Program over.\n\n");
}
```

Output:

```
RGETMSG Sample ESQL Program running.


Error -310: Table (informix.customer) already exists in database.

RGETMSG Sample Program over.
```

The risnull() function

The **risnull()** function checks whether a C or an IBM Informix ESQL/C variable contains a null value.

Syntax:

```
mint risnull(type; ptrvar)
   mint type;
   char *ptrvar;
```

type	An integer that corresponds to the data type of a C or Informix ESQL/C variable. This *type* can be any data type *except* **var binary** or an **lvarchar** pointer variable. For more information, see "Data Type Constants" on page 3-3
ptrvar	A pointer to the C or Informix ESQL/C variable.

Usage:

The **risnull()** function determines whether Informix ESQL/C variables of all data types *except* **var binary** and **lvarchar** pointer variables contain a null value. To determine whether a **var binary** or **lvarchar** pointer host variable contains null, use the **ifx_var_isnull()** macro. For more information, see "The ifx_var_isnull() function" on page B-153.

Return codes:

1	The variable does contain a null value.
0	The variable does not contain a null value.

Example:

This sample program is in the **risnull.ec** file in the **demo** directory.

```
/*
 * risnull.ec *

   This program checks the paid_date column of the orders table for NULL
   to determine whether an order has been paid.
*/

#include <stdio.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY        1
#define NOWARNNOTIFY    0

main()
{
    char ans;
    int4 ret, exp_chk();

    EXEC SQL BEGIN DECLARE SECTION;
        int4 order_num;

      mint order_date, ship_date, paid_date;
    EXEC SQL END DECLARE SECTION;

    printf("RISNULL Sample ESQL Program running.\n\n");
    EXEC SQL connect to 'stores7';         /* open stores7 database*/
    exp_chk("CONNECT TO stores7", NOWARNNOTIFY)

    EXEC SQL declare c cursor for
        select order_num, order_date, ship_date, paid_date from orders;
    EXEC SQL open c;
    if(exp_chk("OPEN c", WARNNOTIFY) == 1)  /* Found warnings */
      exit(1);
    printf("\n Order#\tPaid?\n");         /* print column hdgs */
    while(1)
      {
      EXEC SQL fetch c into :order_num, :order_date, :ship_date, :paid_date;
      if ((ret = exp_chk("FETCH c")) == 100)   /* if end of rows */
         break;          /* terminate loop */
      if(ret < 0)
         exit(1);
      printf("%5d\t", order_num);
      if (risnull(CDATETYPE, (char *)&paid_date)) /* is price NULL ? */
         printf("NO\n");
      else
         printf("Yes\n");
      }
    printf("\nRISNULL Sample Program over.\n\n");
}

/*
 * The exp_chk() file contains the exception handling functions to
 * check the SQLSTATE status variable to see if an error has occurred
 * following an SQL statement. If a warning or an error has
 * occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 * prints the detail for each exception that is returned.
 */

EXEC SQL include exp_chk.ec
```

For a complete listing of the **exp_chk()** function, see "Guide to the exp_chk.ec File" on page 11-32 or see the **exp_chk.ec** file for a listing of this exception-handling function.

Output:

```
RISNULL Sample ESQL Program running.


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

RISNULL Sample Program over.
```

The rjulmdy() function

The **rjulmdy()** function creates an array of three **short** integer values that represent the month, day, and year from an internal DATE value.

Syntax:

```
mint rjulmdy(jdate, mdy)
    int4 jdate;
    int2 mdy[3];
```

jdate   The internal representation of the date.

mdy   An array of **short** integers, where *mdy[0]* is the month (1 to 12), *mdy[1]* is the day (1 - 31), and *mdy[2]* is the year (1 - 9999).

Return codes:

0   The operation was successful.

< 0   The operation failed.

-1210   The internal date could not be converted to the character string format.

Example:

The **demo** directory contains this sample program in the **rjulmdy.ec** file.

```
/*
 * rjulmdy.ec *

   The following program accepts a date entered from the console and converts
   it to an array of three short integers that contain the month, day, and year.
*/

#include <stdio.h>

main()
{
   int4 i_date;
   short mdy_array[3];
   mint errnum;
   char date[20];
   mint x;

   static char fmtstr[9] = "mmddyyyy";

   printf("RJULMDY Sample ESQL Program running.\n\n");

   /* Allow user to enter a date */
   printf("Enter a date as a single string, month.day.year\n");
   gets(date);

   printf("\nThe date string is %s.\n", date);

   /* Put entered date in internal format */
   if (x = rdefmtdate(&i_date, fmtstr, date))
      printf("Error %d on rdefmtdate conversion\n", x);
   else
      {

   /* Convert from internal format to MDY array */
      if ((errnum = rjulmdy(i_date, mdy_array)) == 0)
         {
         printf("\tThe month component is: %d\n", mdy_array[0]);
         printf("\tThe  day  component is: %d\n", mdy_array[1]);
         printf("\tThe year  component is: %d\n", mdy_array[2]);
         }
      else
         printf("rjulmdy() call failed with error %d", errnum);
      }

    printf("\nRJULMDY Sample Program over.\n\n");
}
```

Output:

```
RJULMDY Sample ESQL Program running.

Enter a date as a single string, month.day.year
10.12.07

The date string is 10.12.07.
The month component is: 10
The day component is: 12
The year component is: 2007

RJULMDY Sample Program over.
```

The rleapyear() function

The **rleapyear()** function returns 1 (TRUE) when the argument that is passed to it is a leap year and 0 (FALSE) when it is not.

Syntax:

```
mint rleapyear(year)
   mint year;
```

year   An integer.

Usage:

The argument *year* must be the year component of a date and not the date itself. You must express the *year* in full form (2007) and not abbreviated form (07).

Return codes:

1   The year is a leap year.

0   The year is not a leap year.

Example:

The **demo** directory contains this sample program in the **rleapyear.ec** file.

```
/*
   * rleapyear.ec *

   The following program accepts a date entered from the console
   and stores this date into an int4, which stores the date in
   an 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()
{
   int4 i_date;
   mint errnum;
   short mdy_array[3];
   char date[20];
   mint x;

   static char fmtstr[9] = "mmddyyyy";

   printf("RLEAPYEAR Sample Program running.\n\n");

    /* Allow user to enter a date */
   printf("Enter a date as a single string, month.day.year\n");
   gets(date);

   printf("\nThe date string is %s.\n", date);

   /* Put entered date in internal format */
   if (x = rdefmtdate(&i_date, fmtstr, date))
     printf("Error %d on rdefmtdate conversion\n", x);
   else
      {

      /* 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);
        }

     printf("\nRLEAPYEAR Sample Program over.\n\n");
}
```

Output:

```
RLEAPYEAR Sample ESQL Program running.

Enter a date as a single string, month.day.year
10.12.07

The date string is 10.12.07.
2007 is not a leap year

RLEAPYEAR Sample Program over.
```

The rmdyjul() function

The **rmdyjul()** function creates an internal DATE from an array of three **short** integer values that represent month, day, and year.

Syntax:

```
mint rmdyjul(mdy, jdate)
   int2 mdy[3];
   int4 *jdate;
```

mdy	An array of **short** integer values, where *mdy[0]* is the month (1 to 12), *mdy[1]* is the day (1 - 31), and *mdy[2]* is the year (1 - 9999).
jdate	A pointer to a **long** integer that receives the internal DATE value for the *mdy* array.

Usage:

You can express the year in full form (2007) or abbreviated form (07).

Return codes:

0	The operation was successful.
-1204	The *mdy[2]* variable contains an invalid year.
-1205	The *mdy[0]* variable contains an invalid month.
-1206	The *mdy[1]* variable contains an invalid day.

Example:

The **demo** directory contains this sample program in the **rmdyjul.ec** file.

```
/*
   * rmdyjul.ec *
```

```
        This program converts an array of short integers containing values
        for month, day and year into an integer that stores the date in
        internal format.
*/

#include <stdio.h>

main()
{
    int4 i_date;
    mint errnum;
    static  short mdy_array[3] = { 12, 21, 2007 };
    char str_date[15];

    printf("RMDYJUL Sample ESQL Program running.\n\n");

    /* Convert MDY array into internal format */
    if ((errnum = rmdyjul(mdy_array, &i_date)) == 0)
      {
      rfmtdate(i_date, "mmm dd yyyy", str_date);
      printf("Date '%s' converted to internal format\n", str_date);
      }
    else
      printf("rmdyjul() call failed with errnum = %d\n", errnum);

    printf("\nRMDYJUL Sample Program over.\n\n");
}
```

Output:

```
RMDYJUL Sample ESQL Program running.

Date 'Dec 21 2007' converted to internal format

RMDYJUL Sample Program over.
```

The rsetnull() function

The **rsetnull()** function sets a C variable to a value that corresponds to a database null value.

Syntax:

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

type	A **mint** that corresponds to the data type of a C or IBM Informix ESQL/C variable. This *type* can be any data type except **var binary** or an **lvarchar** pointer variable. For more information, see "Data Type Constants" on page 3-3.
ptrvar	A pointer to the C or Informix ESQL/C variable.

Usage:

The **rsetnull()** function sets to null Informix ESQL/C variables of all data types except **var binary** and **lvarchar** pointer host variables. To set a **var binary** or **lvarchar** pointer host variable to null, use the **ifx_var_setnull()** macro. For more information, see "The ifx_var_setnull() function" on page B-155

Example:

This sample program is in the **rsetnull.ec** file in the **demo** directory.

```
/*
 * 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>
EXEC SQL include decimal;
EXEC SQL include sqltypes;

#define WARNNOTIFY      1
#define NOWARNNOTIFY    0

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

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

main()
{
   char decdsply[20];
   char ans;
   int4 ret, exp_chk();

   EXEC SQL BEGIN DECLARE SECTION;
      short stock_num;
      char description[16];
      dec_t unit_price;
      char manu_code[4];
   EXEC SQL END DECLARE SECTION;

    printf("RSETNULL Sample ESQL Program running.\n\n");
    EXEC SQL connect to 'stores7';          /* connect to stores7 */
    exp_chk("Connect to stores7", NOWARNNOTIFY);

    printf("This program selects all rows for a given manufacturer\n");
    printf("from the stock table and allows you to set the unit_price\n");
    printf("to NULL.\n");
    printf("\nTo begin, enter a manufacturer code - for example: 'HSK'\n");
    printf("\nEnter Manufacturer code: ");        /* prompt for mfr. code */
    gets(manu_code);                /* get mfr. code */
    EXEC SQL declare upcurs cursor for       /* declare cursor */
      select stock_num, description, unit_price from stock
      where manu_code = :manu_code
      for update of unit_price;
    rupshift(manu_code);          /* Make mfr code upper case */
    EXEC SQL open upcurs;  /* open select cursor */
    if(exp_chk("Open cursor", WARNNOTIFY) == 1)
      exit(1);

    /*
     * Display Column Headings
     */
    printf("\nStock # \tDescription \t\tUnit Price");
    while(1)
      {
      /* get a row */
      EXEC SQL fetch upcurs into :stock_num, :description, :unit_price;
      if ((ret = exp_chk("fetch", WARNNOTIFY)) == 100)  /* if end of rows */
         break;
      if(ret == 1)
```

```
            exit(1);
        if(risnull(CDECIMALTYPE, (char *) &unit_price))  /* unit_price NULL? */
            continue;                /* skip to next row */
        rfmtdec(&unit_price, format, decdsply); /* format unit_price */
        /* display item */
        printf("\n\t%d\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. . . Set unit_price to NULL ? (y/n) ");
            scanf("%1s", &ans);
            }
        if (ans == 'y')                        /* if yes, NULL to unit_price */
            {
            rsetnull(CDECIMALTYPE, (char *) &unit_price);
            EXEC SQL update stock set unit_price = :unit_price
                where current of upcurs;        /* and update current row */
            if(exp_chk("UPDATE", WARNNOTIFY) == 1)
                exit(1);
            }
        }
    printf("\nRSETNULL Sample Program over.\n\n");
}

/*
 *  The exp_chk() file contains the exception handling functions to
 *  check the SQLSTATE status variable to see if an error has occurred
 *  following an SQL statement. If a warning or an error has
 *  occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 *  prints the detail for each exception that is returned.
 */

EXEC SQL include exp_chk.ec
```

For a complete listing of the **exp_chk()** function, see "Guide to the exp_chk.ec File"
on page 11-32 or see the **exp_chk.ec** file for a listing of this exception-handling
function.

Output:
```
RSETNULL Sample ESQL Program running.

This program selects all rows for a given manufacturer
from the stock table and allows you to set the unit_price
to NULL.

To begin, enter a manufacturer code - for example: 'HSK'

Enter Manufacturer code: HSK

Stock #        Description           Unit Price
    1           baseball gloves        $800.00
 . . . Set unit_price to NULL ? (y/n) n


    3           baseball bat           $240.00
. . . Set unit_price to NULL ? (y/n) y


    4           football               $960.00
. . . Set unit_price to NULL ? (y/n) n
```

```
          110         helmet                    $600.00
. . . Set unit_price to NULL ? (y/n) y

RSETNULL Sample Program over.
```

The rstod() function

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

Syntax:

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

string   A pointer to a null-terminated string.

double_val   A pointer to a **double** value that holds the converted value.

Usage:

=0   The conversion was successful.

!=0   The conversion failed.

Example:

This sample program is in the **rstod.ec** file in the **demo** directory.

```
/*
   * rstod.ec *

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

#include <stdio.h>

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

   printf("RSTOD Sample ESQL Program running.\n\n");

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

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

   printf("Converting String 3: %s\n", string3);
   if ((errnum = rstod(string3, &d)) == 0)
     printf("\tResult = %.8f\n\n", d);
   else
```

```
        printf("\tError %d in conversion of string 3\n\n", errnum);

     printf("\nRSTOD Sample Program over.\n\n");
}
```

Output:
```
RSTOD Sample ESQL Program running.

Converting String 1: 123456788764321
   Result  =  1234567887654321.000000

Converting String 2: 12345678.87654321
   Result = 12345678.87654321

Converting String 3: zzzzzzzzzzzzzzzz
   Error -1213 in conversion of string 3

RSTOD Sample Program over.
```

The rstoi() function

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

Syntax:
```
mint rstoi(string, ival)
   char *string;
   mint *ival;
```

string   A pointer to a null-terminated string.

ival   A pointer to a **mint** value that holds the converted value.

Usage:

The legal range of values is -32767 - 32767. The value -32768 is not valid because this value is a reserved value that indicates null.

If *string* corresponds to a null integer, *ival* points to the representation for a SMALLINT null. 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:

This sample program is in the **rstoi.ec** file in the **demo** directory.
```
/*
   * rstoi.ec *

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

#include <stdio.h>

EXEC SQL include sqltypes;
```

```
main()
{
    mint err;
    mint i;
    short s;

    printf("RSTOI Sample ESQL Program running.\n\n");

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

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

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

    printf("\nRSTOI Sample Program over.\n\n");
}
```

Output:

```
RSTOI Sample ESQL Program running.

Converting String 'abc':
   Error -1213 in conversion of string #1

Converting String '32766':
   Result = 32766

Converting String '':
   Result = NULL

RSTOI Sample Program over.
```

The rstol() function

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

Syntax:

```
mint rstol(string, long_int)
   char *string;
   mlong *long_int;
```

string   A pointer to a null-terminated string.

long_int   A pointer to an **mlong** value that holds the converted value.

Usage:

The legal range of values is -2,147,483,647 - 2,147,483,647. The value
-2,147,483,648 is not valid because this value is a reserved value that indicates
null.

Return codes:

=0   The conversion was successful.

!=0   The conversion failed.

Example:

This sample program is in the **rstol.ec** file in the **demo** directory.

```
/*
   * rstol.ec *

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

#include <stdio.h>

EXEC SQL include sqltypes;

main()
{
    mint err;
    mlong l;

    printf("RSTOL Sample ESQL Program running.\n\n");

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

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

    l = 0;
    printf("Converting String '':\n");
    if((err = rstol("", &l)) == 0)
      {
      if(risnull(CLONGTYPE, (char *) &l))
         printf("\tResult = NULL\n\n", l);
      else
         printf("\tResult = %ld\n\n", l);
      }
    else
```

```
        printf("\tError %d in conversion of string #3\n\n", err);

    printf("\nRSTOL Sample Program over.\n\n");
}
```

Output:
```
RSTOL Sample ESQL Program running.

Converting String 'abc':
   Error -1213 in conversion of string #1

Converting String '2147483646':
   Result = 2147483646

Converting String '':
   Result = NULL


RSTOL Sample Program over.
```

The rstrdate() function

The **rstrdate()** function converts a character string to an internal DATE.

Syntax:
```
mint rstrdate(inbuf, jdate)
   char *inbuf;
   int4 *jdate;
```

inbuf A pointer to the string that contains the date to convert.

jdate A pointer to an **int4** integer that receives the internal DATE value for the *inbuf* string.

Usage:

For the default locale, U.S. English, the **rstrdate()** function determines how to format the character string with the following precedence:

1. The format that the **DBDATE** environment variable specifies (if **DBDATE** is set). For more information about **DBDATE**, see the *IBM Informix Guide to SQL: Reference*.
2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set). For more information about **GL_DATE**, see the *IBM Informix GLS User's Guide*.
3. The default date form: mm/dd/yyyy. You can use any nonnumeric character as a separator between the month, day, and year. You can express the year as four digits (2007) or as two digits (07).

When you use a nondefault locale and do not set the **DBDATE** or **GL_DATE** environment variable, **rstrdate()** uses the date end-user format that the client locale defines. For more information, see the *IBM Informix GLS User's Guide*.

When you use a two-digit year in the date string, the **rstrdate()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **rstrdate()** assumes the 20th century for two-digit years. For information about how to set **DBCENTURY**, see the *IBM Informix Guide to SQL: Reference*.

Return codes:

0	The conversion was successful.
< 0	The conversion failed.
-1204	The *inbuf* parameter specifies an invalid year.
-1205	The *inbuf* parameter specifies an invalid month.
-1206	The *inbuf* parameter specifies an invalid day.
-1212	Data conversion format must contain a month, day, or year component. **DBDATE** specifies the data conversion format.
-1218	The date specified by the *inbuf* argument does not properly represent a date.

Example:

The **demo** directory contains this sample program in the **rstrdate.ec** file.

```
/*
   * rstrdate.ec *
   The following program converts a character string
   in "mmddyyyy" format to an internal date format.
*/

#include <stdio.h>

main()
{
    int4 i_date;
    mint errnum;
    char str_date[15];

    printf("RSTRDATE Sample ESQL Program running.\n\n");

    /* Convert Sept. 6th, 2007 into i_date */
    if ((errnum = rstrdate("9.6.2007", &i_date)) == 0)
      {

      rfmtdate(i_date, "mmm dd yyyy", str_date);
      printf("Date '%s' converted to internal format\n" str_date);
      }
    else
      printf("rstrdate() call failed with error %d\n", errnum);

    printf("\nRSTRDATE Sample Program over.\n\n");
}
```

Output:

```
RSTRDATE Sample ESQL Program running.

Date 'Sep 06 2007' converted to internal format

RSTRDATE Sample Program over.
```

The rtoday() function

The **rtoday()** function returns the system date as a long integer value.

Syntax:

```
void rtoday(today)
   int4 *today;
```

today   A pointer to an **int4** value that receives the internal DATE.

Usage:

The **rtoday()** function obtains the system date on the client computer, not the server computer.

Example:

The **demo** directory contains this sample program in the **rtoday.ec** file.

```
/*
   * rtoday.ec *

   The following program obtains today's date from the system,
   converts it to ASCII using rdatestr(), and displays the result.
*/

#include <stdio.h>

main()
{
   mint errnum;
   char today_date[20];
   int4 i_date;

   printf("RTODAY Sample ESQL Program running.\n\n");

    /* 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);

   printf("\nRTODAY Sample Program over.\n\n");
}
```

Output:

```
RTODAY Sample ESQL Program running.

   Today's date is 09/16/2007.

RTODAY Sample Program over.
```

The rtypalign() function

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

Syntax:

32 bit

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

64 bit

```
mlong rtypalign(pos, type)
   mlong pos;
   mint type;
```

pos   The current position in a buffer.

type   An integer that corresponds to the data type of a C or Informix ESQL/C variable. This *type* can be any data type except the following:

- **var binary**
- CFIXBINTYPE
- CVARBINTYPE
- SQLUDTVAR
- SQLUDTFIXED

For more information, see "Data Type Constants" on page 3-3.

Usage:

The **rtypalign()** and **rtypmsize()** functions are useful when you use an **sqlda** structure to dynamically fetch data into a buffer. On many hardware platforms, integer and other numeric data types must begin on a work boundary. The C language memory allocation routines allocate memory that is suitably aligned for any data type, including structures. However, these routines do not perform alignment for the constituent components of the structure. The programmer is responsible for performing that alignment with functions such as **rtypalign()** and **rtypmsize()**. These functions provide machine independence for storing column data.

After a DESCRIBE statement determines column information, Informix ESQL/C stores the value of *type* in **sqlda.sqlvar->sqltype**.

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

Return codes:

>0   The return value is the offset of the next proper boundary for a variable of *type* data type.

Example:

This sample program is in the **rtypalign.ec** file in the **demo** directory.

```
/*
   * rtypalign.ec *

   The following program prepares a select on all columns of the orders
   table and then calculates the proper alignment for each column in a buffer.
*/

#include <decimal.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY      1
#define NOWARNNOTIFY    0

main()
```

```
{
    mint i, pos;
    int4 ret, exp_chk();
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    printf("RTYPALIGN Sample ESQL Program running.\n\n");

    EXEC SQL connect to 'stores7';                 /* open stores7 database */
    exp_chk("Connect to", NOWARNNOTIFY);

    EXEC SQL prepare query_1 from "select * from orders";   /* prepare select */
    if(exp_chk("Prepare", WARNNOTIFY) == 1)
      exit(1);

    EXEC SQL describe query_1 into sql_desc;   /* initialize sqlda */
    if(exp_chk("Describe", WARNNOTIFY) == 1)
      exit(1);

    col = sql_desc->sqlvar;
    printf("\n\ttype\t\tlen\tnext\taligned\n");          /* display column hdgs. */
    printf("\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->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->sqllen;                          /* set next position */
      ++col;                                       /* bump to next column */
    }
    printf("\nRTYPALIGN Sample Program over.\n\n");
}

/*
 *  The exp_chk() file contains the exception handling functions to
 *  check the SQLSTATE status variable to see if an error has occurred
 *  following an SQL statement. If a warning or an error has
 *  occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 *  prints the detail for each exception that is returned.
 */

EXEC SQL include exp_chk.ec
```

For a complete listing of the **exp_chk()** function, see "Guide to the exp_chk.ec File"
on page 11-32 or see the **exp_chk.ec** file for a listing of this exception-handling
function.

Output:

```
RTYPALIGN Sample ESQL Program running.
```

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
decimal	22	68	68
money	22	90	90
date	4	112	112

RTYPALIGN Sample Program over.

The rtypmsize() function

The **rtypmsize()** function returns the number of bytes you must allocate in memory for the specified Informix ESQL/C or SQL data type.

Syntax:

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

sqltype   The integer code of the Informix ESQL/C or SQL data type. For more information, see "Data Type Constants" on page 3-3.

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

Usage:

The **rtypalign()** and **rtypmsize()** functions are useful when you use an **sqlda** structure to dynamically fetch data into a buffer. These functions provide machine independence for the column-data storage.

The **rtypmsize()** function is provided to use with the **sqlda** structure that a DESCRIBE statement initializes. After a DESCRIBE statement determines column information, the value of *sqltype* and *sqllen* components reside in the components of the same name in each **sqlda.sqlvar** structure.

When **rtypmsize()** determines sizes for character data, keep the following size information in mind:
- For CCHARTYPE (**char**) and CSTRINGTYPE (**string**), Informix ESQL/C adds one byte to the number of characters for the null terminator.
- For CFIXCHARTYPE (**fixchar**), Informix ESQL/C does not add a null terminator.

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

Return codes:

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

>0   The return value is the number of bytes that the *sqltype* data type requires.

Example:

This sample program is in the **rtypmsize.ec** file in the **demo** directory.

```
/*
 * rtypmsize.ec *

   This program prepares a select statement on all columns of the
   catalog table. Then it displays the data type of each column and
   the number of bytes needed to store it in memory.
*/

#include <stdio.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY      1
#define NOWARNNOTIFY    0

EXEC SQL BEGIN DECLARE SECTION;
    char db_name[20];
EXEC SQL END DECLARE SECTION;

main(argc, argv)
int argc;
char *argv[];
{
    mint i;
    char db_stmnt[50];
   int4 exp_chk();
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    printf("RTYPMSIZE Sample ESQL Program running.\n\n");

    if (argc > 2)            /* correct no. of args? */
      {
      printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
        argv[0]);
      exit(1);
      }
   strcpy(db_name, "stores7");
    if (argc == 2)
      strcpy(db_name, argv[1]);

    EXEC SQL connect to :db_name;
    sprintf(db_stmnt, "CONNECT TO %s", argv[1]);
    exp_chk(db_stmnt, NOWARNNOTIFY);

    printf("Connected to '%s' database.", db_name);

     EXEC SQL prepare query_1 from 'select * from catalog'; /* prepare select */
     if(exp_chk("Prepare", WARNNOTIFY) == 1)
       exit(1);
     EXEC SQL describe query_1 into sql_desc;          /* setup sqlda */
     if(exp_chk("Describe", WARNNOTIFY) == 1)
       exit(1);
     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->sqld; i++, col++)
    printf("\t%-20s%-8s%3d\n", col->sqlname, rtypname(col->sqltype),
```

```
                  rtypmsize(col->sqltype, col->sqllen));

        printf("\nRTYPMSIZE Sample Program over.\n\n");
}

/*
 *  The exp_chk() file contains the exception handling functions to
 *  check the SQLSTATE status variable to see if an error has occurred
 *  following an SQL statement. If a warning or an error has
 *  occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 *  prints the detail for each exception that is returned.
 */
EXEC SQL include exp_chk.ec
```

For a complete listing of the **exp_chk()** function, see "Guide to the exp_chk.ec File" on page 11-32 or see the **exp_chk.ec** file for a listing of this exception-handling function.

Output:

```
RTYPMSIZE Sample ESQL Program running.

Connected to stores7 database.
```

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

```
RTYPMSIZE Sample Program over.
```

The rtypname() function

The **rtypname()** function returns a null-terminated string that contains the name of the specified SQL data type.

Syntax:

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

sqltype   An integer code for one of the SQL data types. For more information, see "Data Type Constants" on page 3-3

The **rtypname()** function converts a constant for an Informix SQL data type (which **sqltypes.h** defines) to a character string.

Return codes:

The **rtypname** function returns a pointer to a string that contains the name of the data type specified *sqltype*. If *sqltype* is an invalid value, **rtypname()** returns a null string (" ").

Example:

This sample program is in the **rtypname.ec** file in the **demo** directory.

```
/*
 * rtypname.ec *

   This program displays the name and the data type of each column
   in the 'orders' table.
*/

#include <stdio.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY      1
#define NOWARNNOTIFY    0

main(argc, argv)
int argc;
char *argv[];
{
    mint i;
    int4 err_chk();
    char db_stmnt[50];
    char *rtypname();
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    EXEC SQL BEGIN DECLARE SECTION;
    char db_name[20];
    EXEC SQL END DECLARE SECTION;

    printf("RTYPNAME Sample ESQL Program running.\n\n");

    if (argc > 2)              /* correct no. of args? */
      {
      printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
         argv[0]);
      exit(1);
      }
    strcpy(db_name, "stores7");

    if (argc == 2)
       strcpy(db_name, argv[1]);

    EXEC SQL connect to :db_name;
    sprintf(db_stmnt, "CONNECT TO %s", argv[1]);
    exp_chk(db_stmnt, NOWARNNOTIFY);

    printf("Connected to '%s' database.", db_name);
     EXEC SQL prepare query_1 from 'select * from orders';  /* prepare select */
     if(exp_chk("Prepare", WARNNOTIFY) == 1)
       exit(1);
     EXEC SQL describe query_1 into sql_desc;       /* initialize sqlda */
     if(exp_chk("Describe", WARNNOTIFY) == 1)
       exit(1);
     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->sqld; i++, col++)
    printf("\t%-15s\t\t%s\n", col->sqlname, rtypname(col->sqltype));

     printf("\nRTYPNAME Sample Program over.\n\n");
}
```

```
/*
 *  The exp_chk() file contains the exception handling functions to
 *  check the SQLSTATE status variable to see if an error has occurred
 *  following an SQL statement. If a warning or an error has
 *  occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 *  prints the detail for each exception that is returned.
 */

EXEC SQL include exp_chk.ec
```

For a complete listing of the **exp_chk()** function, see "Guide to the exp_chk.ec File" on page 11-32 or see the **exp_chk.ec** file for a listing of this exception-handling function.

Output:
```
RTYPNAME Sample ESQL Program running.

Connected to stores7 database
   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

RTYPNAME Sample Program over.
```

The rtypwidth() function

The **rtypwidth()** function returns the minimum number of characters that a character data type needs to avoid truncation when you convert a value with an SQL data type to a character data type.

Syntax:
```
mint rtypwidth(sqltype, sqllen)
   mint sqltype;
   mint sqllen;
```

sqltype   The integer code of the SQL data type. For more information, see "Data Type Constants" on page 3-3.

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

Usage:

The **rtypwidth()** function is provided for use with the **sqlda** structure that a DESCRIBE statement initializes. 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 data type.

>0   The return value is the minimum number of characters that the
sqltype data type requires.

Example:

This sample program is in the **rtypwidth.ec** file in the **demo** directory.

```
/*
   * rtypwidth.ec *

   This program displays the name of each column in the 'orders' table and
   the number of characters required to store the column when the
   data type is converted to characters.
*/

#include <stdio.h>

#define WARNNOTIFY      1
#define NOWARNNOTIFY    0

main(argc, argv)
int argc;
char *argv[];
{
    mint i, numchars;

    int4 exp_chk();
    char db_stmnt[50];
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    EXEC SQL BEGIN DECLARE SECTION;
    char db_name[20];
    EXEC SQL END DECLARE SECTION;

    printf("RTYPWIDTH Sample ESQL Program running.\n\n");

    if (argc > 2)               /* correct no. of args? */
      {
      printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
      argv[0]);
      exit(1);
      }
    strcpy(db_name, "stores7");
    if (argc == 2)
      strcpy(db_name, argv[1]);

    EXEC SQL connect to :db_name;
    sprintf(db_stmnt, "CONNECT TO %s", argv[1]);
    exp_chk(db_stmnt, NOWARNNOTIFY);

    printf("Connected to %s\n", db_name);

    EXEC SQL prepare query_1 from 'select * from orders';  /* prepare select */
    if(exp_chk("Prepare", WARNNOTIFY) == 1)
      exit(1);
    EXEC SQL describe query_1 into sql_desc;               /* setup sqlda */
    if(exp_chk("Describe", WARNNOTIFY) == 1)
      exit(1);
    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->sqld; i++, col++)
      {
      numchars = rtypwidth(col->sqltype, col->sqllen);
      printf("\t%-15s\t%d\n", col->sqlname, numchars);
      }

    printf("\nRTYPWIDTH Sample Program over.\n\n");
}

/*
 *  The exp_chk() file contains the exception handling functions to
 *  check the SQLSTATE status variable to see if an error has occurred
 *  following an SQL statement. If a warning or an error has
 *  occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 *  prints the detail for each exception that is returned.
 */
EXEC SQL include exp_chk.ec
```

Output:

```
RTYPWIDTH Sample ESQL Program running.

Connected to stores7

    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

RTYPWIDTH Sample Program over.
```

The rupshift() function

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

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

s   A pointer to a null-terminated string.

Usage:

The **rupshift()** function refers to the current locale to determine uppercase and
lowercase letters. For the default locale, U.S. English, **rupshift()** uses the ASCII
lowercase (a-z) and uppercase (A-Z) letters.

If you use a nondefault locale, **rupshift()** uses the lowercase and uppercase letters
that the locale defines. For more information, see the *IBM Informix GLS User's
Guide*.

Example:

This sample program is in the **rupshift.ec** file in the **demo** directory.

```
/*
   * 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[] = "123abcdefghijkl;.";

    printf("RUPSHIFT Sample ESQL Program running.\n\n");

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

    printf("\nRUPSHIFT Sample Program over.\n\n");
}
```

Output:

```
RUPSHIFT Sample ESQL Program running.

  Input  string: 123abcdefghijkl;.
  After upshift: 123ABCDEFGHIJKL;.

RUPSHIFT Sample Program over.
```

The SetConnect() function (Windows)

The **SetConnect()** function is available only in Windows environments. It switches the connection to a specified explicit connection.

Important: Informix ESQL/C supports the **SetConnect()** connection library function for compatibility with Version 5.01 Informix ESQL/C for Windows applications. When you write new Informix ESQL/C applications for Windows environments, use the SQL SET CONNECTION statement to switch to another active connection.

Syntax:

```
void *SetConnect ( void *CnctHndl )
```

CnctHndl   A connection handle that a previous **GetConnect()** call has returned.

Usage:

The **SetConnect()** function maps to a simple SQL SET CONNECTION statement (one without a DEFAULT option). The **SetConnect()** call is equivalent to the following SQL statement:

```
EXEC SQL set connection db_connection;
```

In this example, *db_connection* is the name of an existing connection that the **GetConnect()** function has established. You pass this *db_connection* name to the **SetConnect()** function as an argument. It is a connection handle for the connection that you want to make active.

If you pass a null handle, the **SetConnect()** function returns the current connection handle and *does not* change the current connection. If no current connection exits when you pass a null handle, **SetConnect()** returns null.

For example, the following code fragment uses **SetConnect()** to switch from a connection to the **accounts** database on the **acctsrvr** database server (**cnctHndl2**) to a **customers** database on the **mainsrvr** database server (**cnctHndl1**):

```
void *cnctHndl1, *cnctHndl2, *prevHndl;
.
.
.
lish connection 'cnctHndl1' to customers@mainsrvr */
strcpy(InetLogin.InfxServer, "mainsrvr");
cnctHndl1 = GetConnect();
EXEC SQL database customers;
.
.
.
/* Establish connection 'cnctHndl2' to accounts@acctsrvr */
strcpy(InetLogin.InfxServer, "acctsrvr");
cnctHndl2 = GetConnect();
EXEC SQL database accounts;
.
.
.
prevHndl = SetConnect( cnctHndl1 );
```

Important: Because the **SetConnect()** function maps to a SET CONNECTION statement, it sets the SQLCODE and SQLSTATE status codes to indicate the success or failure of the connection switch request. This behavior differs from that of **SetConnect()** in Version 5.01 Informix ESQL/C for Windows, in which this function did not set the SQLCODE and SQLSTATE values.

The **SetConnect()** function differs from the SET CONNECTION statement in the way that it obtains the connection name. **SetConnect()** uses an internally generated name that is stored in the connection handle.You must specify this handle as an argument to the **SetConnect()** call. The SET CONNECTION statement uses the user-defined connection name that the AS clause of the CONNECT statement specifies.

Important: Because the **GetConnect()** function maps to a CONNECT statement with the WITH CONCURRENT TRANSACTION clause, it allows an explicit connection with open transactions to become dormant. Your Informix ESQL/C application does not need to ensure that the current transaction was committed or rolled back before it calls the **SetConnect()** function to switch to another explicit connection.

Return codes:

CnctHndl   The call to **SetConnect()** was successful if the function has returned a connection handle of the connection that is now dormant.

null pointer   The call to **SetConnect()** was *not* successful, indicating that no explicit connection was established.

The sqgetdbs() function

The **sqgetdbs()** function returns the names of databases that a database server can access.

Syntax:

```
mint sqgetdbs(ret_fcnt, dbnarray, dbnsize, dbnbuffer, dbnbufsz)
   mint *ret_fcnt;
   char **dbnarray;
   mint dbnsize;
   char *dbnbuffer;
   mint dbnbufsz;
```

ret_fcnt	A pointer to the number of database names that the function returns.
dbnarray	A user-defined array of character pointers.
dbnsize	The size of the *dbnarray* user-defined array.
dbnbuffer	A pointer to a user-defined buffer that contains the names of the databases that the function returns.
dbnbufsz	The size of the *dbnbuffer* user-defined buffer.

Usage:

You must provide the following user-defined data structures to the **sqgetdbs()** function:

- The *dbnbuffer* buffer holds the names of the null-terminated database names that **sqgetdbs()** returns.
- The *dbnarray* array holds pointers to the database names that the function stores in the *dbnbuffer* buffer. For example, *dbnarray*[0] points to the first character of the first database name (in *dbnbuffer*), *dbnarray*[1] points to the first character of the second database name, and so on.

If the application is connected to a database server, a call to the **sqgetdbs()** function returns the names of the databases that are available in the database server of the current connection. This includes the user-defined databases as well as the **sysmaster** database. Otherwise, it returns the database names that are available in the default database server (that the **INFORMIXSERVER** environment variable indicates). If you use the **DBPATH** environment variable to identify additional database servers that contain databases, **sqgetdbs()** also lists the databases that are available on these database servers. It first lists the databases that are available through **DBPATH** and then the databases that are available through the **INFORMIXSERVER** environment variable.

Return codes:

0	Successfully obtained database names
<0	Unable to obtain database names

Example:

The **sqgetdbs.ec** file in the **demo** directory contains this sample program.

```
/*
   * sqgetdbs.ec *

   This program lists the available databases in the database server
   of the current connection.
*/

#include <stdio.h>

/* Defines used with exception-handling function: exp_chk() */
#define WARNNOTIFY      1
```

```
#define NOWARNNOTIFY    0

/* Defines used for user-defined data structures for sqgetdbs() */
#define BUFFSZ          256
#define NUM_DBNAMES      10

main()
{
    char db_buffer[ BUFFSZ ];      /* buffer for database names */
    char *dbnames[ NUM_DBNAMES ]; /* array of pointers to database
                                       names in 'db_buffer' */
    mint num_returned;               /* number of database names returned */
    mint ret, i;

    printf("SQGETDBS Sample ESQL Program running.\n\n");

    EXEC SQL connect to default;
    exp_chk("CONNECT TO default server", NOWARNNOTIFY);
    printf("Connected to default server.\n");

    ret = sqgetdbs(&num_returned, dbnames, NUM_DBNAMES,
        db_buffer, BUFFSZ);
    if(ret < 0)
        {
        printf("Unable to obtain names of databases.\n");
        exit(1);
        }

    printf("\nNumber of database names returned = %d\n", num_returned);

    printf("Databases currently available:\n");
    for (i = 0; i < num_returned; i++)
        printf("\t%s\n", dbnames[i]);
    printf("\nSQGETDBS Sample Program over.\n\n");
}

/*
 * The exp_chk() file contains the exception handling functions to
 * check the SQLSTATE status variable to see if an error has occurred
 * following an SQL statement. If a warning or an error has
 * occurred, exp_chk() executes the GET DIAGNOSTICS statement and
 * displays the detail for each exception that is returned.
 */
EXEC SQL include exp_chk.ec;
```

For a source listing of the **exp_chk()** exception-handling function, see Chapter 11, "Exception handling."

Output:

The output you see from the **sqgetdbs** sample program depends on how you set your **INFORMIXSERVER** and **DBPATH** environment variables. The following sample output assumes that the **INFORMIXSERVER** environment variable is set to **mainserver** and that this database server contains three databases that are called **stores7**, **sysmaster**, and **tpc**. This output also assumes that the **DBPATH** environment is not set.

```
SQGETDBS Sample ESQL Program running.

Connected to default server.

Number of database names returned = 3
Databases currently available:
```

```
                   stores7@mainserver
                   sysmaster@mainserver
                   tpc@mainserver

               SQGETDBS Sample Program over.
```

The sqlbreak() function

The **sqlbreak()** function sends the database server a request to interrupt processing of the current SQL request. You generally call this function to interrupt long queries.

Syntax:

```
mint sqlbreak();
```

Usage:

The **sqlbreak()** function sends the interrupt request to the database server of the current connection. When the database server receives this request, it must determine if the SQL request is interruptible. Some types of database operations are not interruptible and others cannot be interrupted at certain points. You can interrupt the following SQL statements.

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- CREATE TABLE
- DELETE
- EXECUTE PROCEDURE
- INSERT
- OPEN
- SELECT
- UPDATE

If the SQL request can be interrupted, the database server takes the following actions:

1. Discontinues execution of the current SQL request
2. Sets SQLCODE (**sqlca.sqlcode**) to a negative value (-213)
3. Returns control to the application

When the application regains control after an interrupted SQL request, any resources that are allocated to the SQL statement remain allocated. Any open databases, cursors, and transactions remain open. Any system-descriptor areas or **sqlda** structures remain allocated. The application program is responsible for the graceful termination of the program; it must release resources and roll back the current transaction.

While the database server executes an SQL request, the application is blocked, waiting for results from the database server. To call **sqlbreak()**, you must first set up some mechanism to unblock the application process. Two possible methods follow:

- Provide the application end user with the ability to interrupt an SQL request once it has begun execution.

When the user presses the Interrupt key, the application becomes unblocked and calls the SIGINT signal-handler function. This signal-handler function includes a call to **sqlbreak()** to interrupt the database server. For more information, see 12-23.

- Specify a timeout interval with the **sqlbreakcallback()** function.

  After the timeout interval elapses, the application becomes unblocked and calls the callback function. This callback function includes a call to **sqlbreak()** to interrupt the database server. For more information, see 12-24.

Before your program calls **sqlbreak()**, verify with the **sqldone()** function that the database server is currently processing an SQL request.

Return codes:

0	The call to **sqlbreak()** was successful. The database server connection exists and either a request to interrupt was sent successfully or the database server was idle.
!=0	No database server is running (no database connection exists) when you called **sqlbreak()**.

The sqlbreakcallback() function

The **sqlbreakcallback()** function allows you to specify a timeout interval and to register a callback function. The callback function provides a method for the application to regain control when the database server is processing an SQL request.

Restriction: Do not use the **sqlbreakcallback()** function if your Informix ESQL/C application uses shared memory (**olipcshm**) as the **nettype** to connect to an IBM Informix database server. Shared memory is not a true network protocol and does not handle the nonblocking I/O that is needed to support a callback function. When you use **sqlbreakcallback()** with shared memory, the call appears to register the callback function successfully (it returns zero); however, during SQL requests, the application never calls the callback function.

Syntax:
```
mint sqlbreakcallback(timeout, callbackfunc_ptr);
   int4 timeout;
   void (* callbackfunc_ptr)(int status);
```

timeout	The interval of time to wait for an SQL request to execute before the application process regains control.

This value can be as follows:

-1	clears the timeout value.
0	immediately calls the function that *callbackfunc_ptr* indicates.
>0	sets the timeout interval to the number of milliseconds to elapse before the application calls the function that *callbackfunc_ptr* indicates.

The *timeout* parameter is a 4-byte variable. This parameter is operating-system dependent: it can be a variable with an **int**, **long**, or **short** data type.

callbackfunc_ptr   A pointer to the user-defined callback function.

Usage:

Once you register a callback function with **sqlbreakcallback()**, the application calls this function at three different points in the execution of an SQL request. The value in the *status* argument of the callback function indicates the point at which the application calls the function. The following table summarizes the *status* values.

When callback function is called	Value of status argument
When the database server begins processing an SQL request	*status* = 1
While the database server executes an SQL request, when the timeout interval has elapsed	*status* = 2
When the database server completes the processing of an SQL request	*status* = 0

When you call the callback function with a *status* value of 2, the callback function can determine whether the database server can continue processing with one of following actions:

- It can call the **sqlbreak()** function to cancel the SQL request.
- It can omit the call to **sqlbreak()** to continue the SQL request.

The callback function, and any of its subroutines, can contain only the following Informix ESQL/C control functions: **sqldone()**, **sqlbreak()**, and **sqlbreakcallback()**. For more information about the callback function, see 12-24.

If you call **sqlbreakcallback()** with a timeout value of zero, the callback function executes immediately. The callback function executes over and over again unless it contains a call to **sqlbreakcallback()** to redefine the callback function with one of the following actions:

- It disassociates the callback function to discontinue the calling of the callback function, as follows:
  ```
  sqlbreakcallback(-1L, (void *)NULL);
  ```
- It defines some other callback function or resets the timeout value to a nonzero value, as follows:
  ```
  sqlbreakcallback(timeout, callbackfunc_ptr);
  ```

Important: Small timeout values might adversely affect the performance of your application.

For more information about the timeout interval, see 12-24.

You must establish a database server connection before you call the **sqlbreakcallback()** function. The callback function remains in effect for the duration of the connection or until the **sqlbreakcallback()** function redefines the callback function.

Return codes:

0	The call to **sqlbreakcallback()** was successful.
<0	The call to **sqlbreakcallback()** was not successful.

The sqldetach() function

The **sqldetach()** function detaches a process from the database server. You generally call this function when an application forks a new process to begin a new stream of execution.

Syntax:

```
mint sqldetach();
```

Usage:

If an application spawns one or more processes after it initiates a connection to a database server, all the child processes inherit that database server connection from the parent process (the application process that spawned the child). However, the database server still assumes that this connection has only one process. If one database server connection tries to serve both the parent and child processes at the same time, problems can result. For example, if both processes send messages to do something, the database server has no way of knowing which messages belong to which process. The database server might not receive messages in an order that makes sense and might thereby generate an error (such as error -408).

In this situation, call the **sqldetach()** function from the child process. The **sqldetach()** function detaches the child process from the connection that the parent process establishes (which the child inherits). This action drops all database server connections in the child process. The child process can then establish its own connection to a database server.

Use the **sqldetach()** function with the **fork()** system call. When you spawn a child process from an application process with a database server connection, sequence the function calls as follows:

1. Call **fork()** from the parent process to create a copy of the parent process (the *child* process). Now both parent and child share the same connection to the database server.
2. Call **sqldetach()** from the child process to detach the child process from the database server. This call closes the connection in the child process.

Tip: You cannot use **sqldetach()** after a **vfork()** call because **vfork()** does not execute a true process fork until the **exec()** function is called. Do not use **sqldetach()** after the parent process uses an **exec()**; when **exec()** starts the child process, the child process does not inherit the connection that the parent process established.

A call to the **sqldetach()** function does not affect the database server sessions of the parent process. Therefore, after **sqldetach()** executes in the child process, the parent process retains any open cursors, transactions, or databases, and the child process has neither database server sessions nor database server connections.

When you call the **sqlexit()** function from the parent process, the function drops the connection in the parent process but does not affect the connections in the child process. Similarly, when you call **sqlexit()** from the child process, the function drops only the child connections; it does not affect the parent connections. The **sqlexit()** function rolls back any open transactions before it closes the connection.

If you execute the DISCONNECT statement from a child process, you disconnect the process from database server connections and terminate the database server sessions that correspond to those connections. The DISCONNECT fails if any transactions are open.

If the child process application has only one implicit connection before it calls **sqldetach()**, execution of the next SQL statement or of the **sqlstart()** library function reestablishes an implicit connection to the default database server. If the application has made one or more explicit connections, you must issue a CONNECT statement before you execute any other SQL statements.

The **sqldetach** demonstration program illustrates how to use the **sqldetach()** function.

Return codes:

0   The call to **sqldetach()** was successful.

<0   The call to **sqldetach()** was not successful.

Example:

The **sqldetach.ec** file in the **demo** directory contains this sample program.

```
/*
   * sqldetach.ec *

   This program demonstrates how to detach a child process from a
   parent process using the ESQL/C sqldetach() library function.
*/

main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        mint pa;
    EXEC SQL END DECLARE SECTION;

    printf("SQLDETACH Sample ESQL Program running.\n\n");

    printf("Beginning execution of parent process.\n\n");
    printf("Connecting to default server...\n");
    EXEC SQL connect to default;
    chk("CONNECT");
    printf("\n");

    printf("Creating database 'aa'...\n");
    EXEC SQL create database aa;
    chk("CREATE DATABASE");
    printf("\n");

    printf("Creating table 'tab1'...\n");
    EXEC SQL create table tab1 (a integer);
    chk("CREATE TABLE");
    printf("\n");

    printf("Inserting 4 rows into 'tab1'...\n");
    EXEC SQL insert into tab1 values (1);
    chk("INSERT #1");
    EXEC SQL insert into tab1 values (2);
    chk("INSERT #2");
    EXEC SQL insert into tab1 values (3);
    chk("INSERT #3");
    EXEC SQL insert into tab1 values (4);
    chk("INSERT #4");
```

```
    printf("\n");

    printf("Selecting rows from 'tab1' table...\n");
    EXEC SQL declare c cursor for select * from tab1;
    chk("DECLARE");

EXEC SQL open c;
    chk("OPEN");

    printf("\nForking child process...\n");
    fork_child();

    printf("\nFetching row from cursor 'c'...\n");
    EXEC SQL fetch c into $pa;
    chk("Parent FETCH");
    if (sqlca.sqlcode == 0)
        printf("Value selected from 'c' = %d.\n", pa);
    printf("\n");

    printf("Cleaning up...\n");
    EXEC SQL close database;
    chk("CLOSE DATABASE");
    EXEC SQL drop database aa;
    chk("DROP DATABASE");
    EXEC SQL disconnect all;
    chk("DISCONNECT");

    printf("\nEnding execution of parent process.\n");
    printf("\nSQLDETACH Sample Program over.\n\n");
}

fork_child()
{
    mint rc, status, pid;

    EXEC SQL BEGIN DECLARE SECTION;
        mint cnt, ca;
    EXEC SQL END DECLARE SECTION;

    pid = fork();
    if (pid < 0)
        printf("can't fork child.\n");

    else if (pid == 0)
    {
        printf("\n*********************************************\n");
        printf("* Beginning execution of child process.\n");
        rc = sqldetach();
        printf("* sqldetach() call returns %d.\n", rc);

        /* Verify that the child is not longer using the parent's
         * connection and has not inherited the parent's connection
         * environment.
         */
        printf("* Trying to fetch row from cursor 'c'...\n");
        EXEC SQL fetch c into $ca;
        chk("* Child FETCH");
        if (sqlca.sqlcode == 0)
            printf("* Value from 'c' = %d.\n", ca);

        /* startup a connection for the child, since
         * it doesn't have one.
         */
      printf("\n* Establish a connection, since child doesn't have one\n");
        printf("* Connecting to database 'aa'...\n");
        EXEC SQL connect to 'aa';
        chk("* CONNECT");
```

```
        printf("* \n");
      printf("* Determining number of rows in 'tab1'...\n");
        EXEC SQL select count(*) into $cnt from tab1;
        chk("* SELECT");
        if (sqlca.sqlcode == 0)
            printf("* Number of entries in 'tab1' =  %d.\n", cnt);
        printf("* \n");

        printf("* Disconnecting from 'aa' database...\n");
        EXEC SQL disconnect current;
        chk("* DISCONNECT");
        printf("* \n");
        printf("* Ending execution of child process.\n");
        printf("*********************************************\n");

        exit();
    }

    /* wait for child process to finish */
    while ((rc = wait(&status)) != pid && rc != -1);

}

chk(s)
char *s;
{
    mint msglen;
    char buf1[200], buf2[200];

    if (SQLCODE == 0)
    {
        printf("%s was successful\n", s);
        return;
    }
    printf("\n%s:\n", s);
    if (SQLCODE)
    {
        printf("\tSQLCODE =  %6d: ", SQLCODE);
        rgetlmsg(SQLCODE, buf1, sizeof(buf1), &msglen);
        sprintf(buf2, buf1, sqlca.sqlerrm);
        printf(buf2);
        if (sqlca.sqlerrd[1])
        {
            printf("\tISAM Error =  %6hd: ", sqlca.sqlerrd[1]);
            rgetlmsg(sqlca.sqlerrd[1], buf1, sizeof(buf1), &msglen);
            sprintf(buf2, buf1, sqlca.sqlerrm);
            printf(buf2);
        }
    }
}
```

Output:

```
SQLDETACH Sample ESQL Program running.

Beginning execution of parent process.

Connecting to default server...
CONNECT was successful

Creating database 'aa'...
CREATE DATABASE was successful

Creating table 'tab1'...
CREATE TABLE was successful

Inserting 4 rows into 'tab1'...
```

```
INSERT #1 was successful
INSERT #2 was successful
INSERT #3 was successful
INSERT #4 was successful

Selecting rows from 'tab1' table...
DECLARE was successful
OPEN was successful

Forking child process...

************************************************
* Beginning execution of child process.
* sqldetach() call returns 0.
* Trying to fetch row from cursor 'c'...

* Child FETCH:
   SQLCODE =    -404: The cursor or statement is not available.

* Establish a connection, since child doesn't have one
* Connecting to database 'aa'...
* CONNECT was successful
*
* Determining number of rows in 'tab1'...
* SELECT was successful
* Number of entries in 'tab1' =  4.
*
* Disconnecting from 'aa' database...
* DISCONNECT was successful
*
* Ending execution of child process.
************************************************
SQLDETACH Sample ESQL Program running.

Beginning execution of parent process.

Connecting to default server...
CONNECT was successful

Creating database 'aa'...
CREATE DATABASE was successful

Creating table 'tab1'...

CREATE TABLE was successful

Inserting 4 rows into 'tab1'...
INSERT #1 was successful
INSERT #2 was successful
INSERT #3 was successful
INSERT #4 was successful

Selecting rows from 'tab1' table...
DECLARE was successful
OPEN was successful

Forking child process...

Fetching row from cursor 'c'...
Parent FETCH was successful
Value selected from 'c' = 1.

Cleaning up...
CLOSE DATABASE was successful
DROP DATABASE was successful
DISCONNECT was successful
```

```
Ending execution of parent process.

SQLDETACH Sample Program over.
```

The sqldone() function

The **sqldone()** function determines whether the database server is currently processing an SQL request.

Syntax:
```
mint sqldone();
```

Usage:

Use **sqldone()** to test the status of the database server in the following situations:
- Before a call to the **sqlbreak()** function to determine if the database server is processing an SQL request.
- In a signal-handler function, before a call to the **longjmp()** system function. Only use **longjmp()** in a signal-handler function if **sqldone()** returns zero (the database server is idle).

When the **sqldone()** function determines that the database server is not currently processing an SQL request, you can assume that the database server does not begin any other processing until your application issues its next request.

You might want to create a defined constant for the **-439** value to make your code more readable. For example, the following code fragment creates the SERVER_BUSY constant and then uses it to test the **sqldone()** return status:
```
#define SERVER_BUSY -439

⋮

if (sqldone() == SERVER_BUSY)
```

Return codes:

0	The database server is not currently processing an SQL request: it is idle.
-439	The database server is currently processing an SQL request.

The sqlexit() function

The **sqlexit()** function terminates all database server connections and frees resources. You can use **sqlexit()** to reduce database overhead in programs that refer to a database only briefly and after long intervals, or that access a database only during initialization.

Syntax:
```
mint sqlexit();
```

Usage:

Only call the **sqlexit()** function when no databases are open. If an open database uses transactions, **sqlexit()** rolls back any open transactions before it closes the database. The behavior of this function is similar to that of the DISCONNECT ALL

statement. However, the DISCONNECT ALL statement *fails* if any current transactions exist. Use the CLOSE DATABASE statement to close open databases before you call **sqlexit()**.

If the application has only one implicit connection before it calls **sqlexit()**, execution of the next SQL statement or of the **sqlstart()** library function reestablishes an implicit connection to the default database server. If the application makes one or more explicit connections, you must issue a CONNECT statement before you execute any other SQL statements.

Return codes:

0	The call to **sqlexit()** was successful.
<0	The call to **sqlexit()** was not successful.

The SqlFreeMem() function

If your application is running on a Windows operating system and using the multi-threaded library, then you must use **SqlFreeMem()** when freeing memory that was allocated by the Informix ESQL/C libraries. If you instead use **free()** you will not get an error but the memory will not be freed.

Do not use **SqlFreeMem()** to free memory that was not allocated by one of the functions in the Informix ESQL/C library.

Syntax:

```
void SqlFreeMem( void *memaddr, int freetype);
```

memaddr   The address of the memory to be freed

freetype   The type of memory to be freed and can have the following values:

> **STRING_FREE**
> > Use for strings that were returned by an Informix ESQL/C function and which your application is expected to free.

> **SQLDA_FREE**
> > Use this to free an **sqlda** structure returned by a DESCRIBE INTO statement. This also frees the associated **sqlvar_struct** structures.
> >
> > If your application allocated the **sqlvar_struct** structures you must free them and set the **sqlda.sqlvar** field to null before calling this function.

> **LOC_BUFFER_FREE**
> > Use to free a LOC_BUFFER returned by select or fetch of a simple large object.

The sqlsignal() function

The **sqlsignal()** function enables or disables signal handling of the signals that the Informix ESQL/C library handles.

Syntax:

```
void sqlsignal(sigvalue, sigfunc_ptr, mode)
   mint sigvalue;
   void (*sigfunc_ptr) (void); short mode;
```

sigvalue	The **mint** value of the particular signal that needs to be trapped (as **signal.h**) defines).
	Currently, this parameter is a placeholder for future functionality. Initialize this argument to **-1**.
sigfunc_ptr	A pointer to the user-defined function, which takes no arguments, to call as a signal handler for the *sigvalue* signal.
	Currently, this parameter is a placeholder for future functionality. Initialize this argument to a null pointer to a function that receives no arguments.
mode	Can be one of three possible modes:

0	Initializes signal handling.
1	Disables signal handling.
2	Re-enables signal handling.

Usage:

The **sqlsignal()** function currently provides handling only for the SIGCHLD signal. In some instances, defunct child processes remain after the application ends. If the application does not clean up these processes, they can cause needless use of process IDs and increase the risk that you run out of processes. This behavior is only apparent when the application uses pipes for client-server communication (that is, the **nettype** field of the **sqlhosts** file is **ipcpip**). You do not need to call **sqlsignal()** for other communication mechanisms (for example, a **nettype** of **tlitcp**).

The *mode* argument of **sqlsignal()** determines the task that **sqlsignal()** performs, as follows:

- Set *mode* to 0 to initialize signal handling.

  ```
  sqlsignal(-1, (void (*)()), 0);
  ```

  When you initialize signal handling with **sqlsignal()**, the Informix ESQL/C library traps the SIGCHLD signal to handle the cleanup of defunct child processes. This initial call to **sqlsignal()** must occur at the *beginning* of your application, before the first SQL statement in the program. If you omit this initial call, you cannot turn on the signal-handling capability later in your program.

- Enable and disable signal handling.

  If you want to have the Informix ESQL/C library perform signal handling for portions of the program and your own code perform signal handling for other portions, you can take the following actions:
  - To disable signal handling, call **sqlsignal()** with *mode* set to 1, at the point where you want your program to handle signals:

    ```
    sqlsignal(-1, (void (*)()), 1);
    ```
  - To re-enable signal handling, call **sqlsignal()** with *mode* set to 2, at the point where you want the IBM Informix ESQL library to handle signals:

    ```
    sqlsignal(-1, (void (*)()), 2);
    ```

  When you initialize SIGCHLD signal handling with **sqlsignal()**, you allow the Informix ESQL/C library to process SIGCHLD cleanup. Otherwise, your application must perform the cleanup for these processes if defunct child processes are a problem.

The sqlstart() function

The **sqlstart()** function starts an implicit default connection. An implicit default connection can support one connection to the default database server (that the **INFORMIXSERVER** environment variable specifies).

Tip: Restrict use of **sqlstart()** to pre-Version 6.0 applications that only use one connection. Informix ESQL/C continues to support this function for compatibility with earlier versions of these applications. For applications of Version 6.0 and later, use the CONNECT statement to establish explicit connections to a default database server.

Syntax:

```
mint sqlstart();
```

Usage:

Informix ESQL/C provides the **sqlstart()** function for pre-Version 6.0 applications that can only support single connections. In this context, possible uses of **sqlstart()** are as follows:

- You only need to verify that the default database server is available but you do not intend to open a database. If the call to **sqlstart()** fails, you can check the return status to verify that the default database server is not available.
- You need to speed up the execution of the DATABASE statement when the application runs over a network. When you put the call to **sqlstart()** in an initialization routine, the application establishes a connection before the user begins interaction with the application. The DATABASE statement can then open the specified database.
- You do not know the name of the actual database to access, or your application plans to create a database. The call to **sqlstart()** can establish the implicit default connection and the application can later determine the name of the database to access or create.

If you have a pre-Version 6.0 application that needs an implicit default connection for any other reason, use the DATABASE statement instead of **sqlstart()**. For applications of Version 6.0 and later, use the CONNECT statement to establish database server connections.

When you call the **sqlstart()** function, make sure that the application has not yet established any connections, implicit or explicit. When the application has established an explicit connection, **sqlstart()** returns error -1811. If an implicit connection was established, **sqlstart()** returns error -1802.

You can call this function several times before you establish an explicit connection, as long as each implicit connection is disconnected before the next call to **sqlstart()**. For information about disconnecting, see "Terminating a connection" on page 12-27. For more information about explicit and implicit connections, see "Establishing a Connection" on page 12-10.

Return codes:

0   The call to **sqlstart()** was successful.

<0   The call to **sqlstart()** was not successful.

The stcat() function

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

Syntax:

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

s	A pointer to the start of the string that **stcat()** places at the end of the destination string.
dest	A pointer to the start of the null-terminated destination string.

Example:

This sample program is in the **stcat.ec** file in the **demo** directory.

```
/*
   * stcat.ec *

   This program uses stcat() to append user input to a SELECT statement.
*/

#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 = ";

main()
{
    char custno[11];

    printf("STCAT Sample ESQL Program running.\n\n");

    printf("Initial SELECT string:\n  '%s'\n", selstmt);

    printf("\nEnter Customer #: ");
    gets(custno);


/*
    * Add custno to "select statement"
    */
    printf("\nCalling stcat(custno, selstmt)\n");
    stcat(custno, selstmt);
    printf("SELECT string is:\n  '%s'\n", selstmt);

    printf("\nSTCAT Sample Program over.\n\n");
}
```

Output:

```
STCAT Sample ESQL Program running.

Initial SELECT string:
  'select fname, lname from customer where customer_num = '

Enter Customer #: 104

Calling stcat(custno, selstmt)
```

```
SELECT string is:
  'select fname, lname from customer where customer_num = 104'

STCAT Sample Program over.
```

The stchar() function

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;
   mint count;
```

from	A pointer to the first byte of a null-terminated source string.
to	A pointer to the fixed-length destination string. This argument can point to a location that overlaps the location to which the *from* argument points. In this case, Informix ESQL/C discards the value to which *from* points.
count	The number of bytes in the fixed-length destination string.

Example:

This sample program is in the **stchar.ec** file in the **demo** directory.

```
/*
   * 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] = "123abcdefghijkl;.";

    printf("STCHAR Sample ESQL Program running.\n\n");

    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);

    printf("\nSTCHAR Sample Program over.\n\n");
}
```

Output:

```
STCHAR Sample ESQL Program running.

Source string: [start]
```

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

STCHAR Sample Program over.
```

The stcmpr() function

The **stcmpr()** function compares two null-terminated strings.

Syntax:

```
mint stcmpr(s1, s2)
   char *s1, *s2;
```

s1   A pointer to the first null-terminated string.

s2   A pointer to the second null-terminated string.

> **Important:** *s1* is greater than *s2* when *s1* appears after *s2* in the
> ASCII collation 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:

This sample program is in the **stcmpr.ec** file in the **demo** directory.

```
/*
   * stcmpr.ec *

   The following program displays the results of three string
   comparisons using stcmpr().
*/

#include <stdio.h>

main()
{
    printf("STCMPR Sample ESQL Program running.\n\n");

    printf("Executing: stcmpr(\"aaa\", \"aaa\")\n");
    printf("  Result = %d", stcmpr("aaa", "aaa"));  /* equal */
    printf("\nExecuting: stcmpr(\"aaa\", \"aaaa\")\n");
    printf("  Result = %d", stcmpr("aaa", "aaaa")); /* less */
    printf("\nExecuting: stcmpr(\"bbb\", \"aaaa\")\n");
    printf("  Result = %d\n", stcmpr("bbb", "aaaa")); /* greater */

    printf("\nSTCMPR Sample Program over.\n\n");
}
```

Output:

```
STCMPR Sample ESQL Program running.

Executing: stcmpr("aaa", "aaa")
  Result = 0
Executing: stcmpr("aaa", "aaaa")
  Result = -1
```

```
Executing: stcmpr("bbb", "aaaa")
  Result = 1

STCMPR Sample Program over.
```

The stcopy() function

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   A pointer to the null-terminated string that you want **stcopy()** to copy.

to   A pointer to a location in memory where **stcopy()** copies the string.

Example:

This sample program is in the **stcopy.ec** file in the **demo** directory.

```
/*
   * stcopy.ec *

   This program displays the result of copying a string using stcopy().
*/

#include <stdio.h>

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

    printf("STCOPY Sample ESQL Program running.\n\n");

    printf("Initial string:\n  [%s]\n", string);          /* display dest */
    stcopy("John Doe", &string[15]);                      /* copy */
    printf("After copy of 'John Doe' to position 15:\n  [%s]\n",
        string);

    printf("\nSTCOPY Sample Program over.\n\n");
}
```

Output:

```
STCOPY Sample ESQL Program running.

Initial string:
  [abcdefghijklmnopqrstuvwxyz]
After copy of 'John Doe' to position 15:
  [abcdefghijklmnoJohn Doe]

STCOPY Sample Program over.
```

The stleng() function

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

Syntax:

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

string   A pointer to a null-terminated string.

Usage:

The length does not include the null terminator.

Example:

This sample program is in the **stleng.ec** file in the **demo** directory.

```
/*
   * stleng.ec *

   This  program uses stleng to find strings that are greater than 35
   characters in length.
*/

#include <stdio.h>

char *strings[] =
   {
   "Your First Season's Baseball Glove",
   "ProCycle Stem with Pearl Finish",
   "Athletic Watch w/4-Lap Memory, Olympic model",
   "High-Quality Kickboard",
   "Team Logo Silicone Swim Cap - fits all head sizes",
   };

main(argc, argv)
int argc;
char *argv[];
{
    mint length, i;

    printf("STLENG Sample ESQL Program running.\n\n");

printf("Strings with lengths greater than 35:\n");
    i = 0;
    while(strings[i])
      {
      if((length = stleng(strings[i])) > 35)
         {
         printf("  String[%d]: %s\n", i, strings[i]);
         printf("  Length: %d\n\n", length);
         }
      ++i;
      }
    printf("\nSTLENG Sample Program over.\n\n");
}
```

Output:
```
STLENG Sample ESQL Program running.

Strings with lengths greater than 35:
  String[2]: Athletic Watch w/4-Lap Memory, Olympic model
  Length: 44

  String[4]: Team Logo Silicone Swim Cap - fits all head sizes
  Length: 49

STLENG Sample Program over.
```

Appendix C. Examples for Smart-Large-Object Functions

The examples in this appendix apply only if you are using IBM Informix as your database server. The examples in this appendix illustrate how to use the Informix ESQL/C library functions to access smart large objects.

Prerequisites

The examples in this section depend on the existence of the following, alternate **catalog** table for the **stores7** database. The examples also depend on the presence of an sbspace, **s9_sbspc**, that stores the contents of the BLOB and CLOB columns, **picture** and **advert_descr**, in the alternate **catalog** table.

```
-- create table that uses smart large objects (CLOB & BLOB) to
-- store the catalog advertisement data.

CREATE TABLE  catalog
        (
        catalog_num     SERIAL8 (10001) primary key,
        stock_num       SMALLINT,
        manu_code       CHAR(3),
        unit            CHAR(4),
        advert          ROW (picture BLOB, caption VARCHAR(255, 65)),
        advert_descr    CLOB,
    FOREIGN KEY (stock_num, manu_code) REFERENCES stock constraint aa
        )
        PUT advert IN (s9_sbspc)
           (EXTENT SIZE 100),
        advert_descr IN (s9_sbspc)
           (EXTENT SIZE 20, KEEP ACCESS TIME)
```

The following example illustrates typical commands to create an sbspace. The values of specific options can vary. You must replace PATH with the complete file name of the file that you allocate for the sbspace.

```
touch s9_sbspc
onspaces -c -S s9_sbspc -g 4 -p PATH -o 0 -s 2000
```

For more information about how to create an sbspace, and particularly on the **onspaces** utility, see your *IBM Informix Administrator's Guide*.

Figure C-1 illustrates the format of entries in a load file that you might use to load data into the alternate **catalog** table. A load file contains data that the LOAD statement loads into a table. Each line in Figure C-1 loads one row in the table. Figure C-1 shows only a sample of code that you can use to load the **catalog** table. For more information about the LOAD statement, see the *IBM Informix Guide to SQL: Syntax*:

```
0|1|HRO|case|ROW(/tmp/cn_1001.gif,"Your First Season's Baseball Glove")|0,62,
    /tmp/catalog.des|
0|1|HSK|case|ROW(NULL,"All Leather, Hand Stitched, Deep Pockets, Sturdy Webbing That Won't
    Let Go")||
0|1|SMT|case|ROW(NULL,"A Sturdy Catcher's Mitt With the Perfect Pocket")||
0|2|HRO|each|ROW(NULL,"Highest Quality Ball Available, from the Hand-Stitching to the
    Robinson Signature")||
0|3|HSK|case|ROW(NULL,"High-Technology Design Expands the Sweet Spot")||
0|3|SHM|case|ROW(NULL,"Durable Aluminum for High School and Collegiate Athletes")||
0|4|HSK|case|ROW(NULL,"Quality Pigskin with Norm Van Brocklin Signature")||
```

Figure C-1. Example of load file for alternate catalog table

The code in Figure C-2 illustrate the format of information in the **catalog.des** file, to which Figure C-1 refers. The entry for **advert_descr** (0,62,/tmp/catalog.des) in Figure C-1 specifies the offset, length, and file name from which the description is loaded. The offset and length are hexadecimal values.

```
Brown leather. Specify first baseman's or infield/outfield style.  Specify right- or
    left-handed.

Double or triple crankset with choice of chainrings. For double crankset, chainrings from
    38-54 teeth. For triple crankset, chainrings from 24-48 teeth.

No buckle so no plastic touches your chin. Meets both ANSI and Snell standards for impact
    protection.7.5 oz. Lycra cover.

Fluorescent yellow.

Super shock-absorbing gel pads disperse vertical energy into a horizontal plane for
    extraordinary cushioned comfort. Great motion control. Mens only. Specify size
```

Figure C-2. Example of load file for alternate catalog table

This appendix contains the following example programs.

Program	Description	See
create_clob.ec	Inserts a row that contains a CLOB column into the alternate catalog table that is described in C-1.	C-2
get_lo_info.ec	Appends the price from the stock table of the **stores7** database to the **advert_descr** column of the alternate **catalog** table that is described in C-1.	C-5
upd_lo_descr.ec	Obtains the price of catalog items for which the **advert_descr** column is not null and appends the price to the description.	C-8

The create_clob.ec program

The **create_clob** program demonstrates how to perform the following tasks on a smart large object:

- Create a new smart large object with user-defined storage characteristics.
- Insert the new smart large object into a database column.

Storage characteristics for the example

The **create_clob** program creates an **advert_descr** smart large object that has the following user-defined storage characteristics:

- Logging is on: LO_LOG
- Keep last access time (default from **advert_descr** column):
  LO_KEEP_ACCESSTIME
- Integrity is high
- Allocation extent size is 10 KB

```
EXEC SQL include int8;
EXEC SQL include locator;

EXEC SQL define BUFSZ 10;

extern char statement[80];

main()
{
EXEC SQL BEGIN DECLARE SECTION;
    int8 catalog_num, estbytes, offset;
    int error, numbytes, lofd, ic_num, buflen = 256;
    char buf[256], srvr_name[256], col_name[300];
    ifx_lo_create_spec_t *create_spec;
    fixed binary 'clob' ifx_lo_t descr;
EXEC SQL END DECLARE SECTION;

    void nullterm(char *);
    void handle_lo_error(int);

    EXEC SQL whenever sqlerror call whenexp_chk;
    EXEC SQL whenever sqlwarning call whenexp_chk;

    printf("CREATE_CLOB Sample ESQL program running.\n\n");
    strcpy(statement, "CONNECT stmt");
    EXEC SQL connect to 'stores7';
    EXEC SQL get diagnostics exception 1
        :srvr_name = server_name;
    nullterm(srvr_name);

/* Allocate and initialize the LO-specification structure
*/
    error = ifx_lo_def_create_spec(&create_spec);
    if (error < 0)
    {
        strcpy(statement, "ifx_lo_def_create_spec()");
        handle_lo_error(error);
    }

/* Get the column-level storage characteristics for the
 * CLOB column, advert_descr.
 */
    sprintf(col_name, "stores7@%s:catalog.advert_descr",
            srvr_name);
    error = ifx_lo_col_info(col_name, create_spec);
    if (error < 0)
    {
        strcpy(statement, "ifx_lo_col_info()");
        handle_lo_error(error);
    }

/* Override column-level storage characteristics for
 * advert_desc with the following user-defined storage
 * characteristics:
 * no logging
 * extent size = 10 kilobytes
 */
```

```
            ifx_lo_specset_flags(create_spec,LO_LOG);
            ifx_int8cvint(BUFSZ, &estbytes);
            ifx_lo_specset_estbytes(create_spec, &estbytes);

        /* Create an LO-specification structure for the smart large object
         */

        if ((lofd = ifx_lo_create(create_spec, LO_RDWR,
            &descr, &error)) == -1)
        {
            strcpy(statement, "ifx_lo_create()");
            handle_lo_error(error);
        }
        /* Copy data into the character buffer 'buf' */

        sprintf(buf, "%s %s",
            "Pro model infielder's glove. Highest quality leather and
            stitching. "
            "Long-fingered, deep pocket, generous web.");

        /* Write contents of character buffer to the open smart
        * large object that lofd points to.  */

        ifx_int8cvint(0, &offset);
        numbytes = ifx_lo_writewithseek(lofd, buf, buflen,
            &offset, LO_SEEK_SET, &error);
        if ( numbytes < buflen )
        {
            strcpy(statement, "ifx_lo_writewithseek()");
            handle_lo_error(error);
        }

    /* Insert the smart large object into the table */
        strcpy(statement, "INSERT INTO catalog");
        EXEC SQL insert into catalog values (0, 1, 'HSK', 'case', ROW(NULL,
        NULL),
          :descr);

    /* Need code to find out what the catalog_num value was assigned to new
     * row */
    /* Close the LO file descriptor */
        ifx_lo_close(lofd);

    /* Select back the newly inserted value. The SELECT
    * returns an LO-pointer structure, which you then use to
    * open a smart large object to get an LO file descriptor.
    */
        ifx_getserial8(&catalog_num);
        strcpy(statement, "SELECT FROM catalog");
        EXEC SQL select advert_descr into :descr from catalog
            where catalog_num = :catalog_num;

    /* Use the returned LO-pointer structure to open a smart
    * large object and get an LO file descriptor.
    */
        lofd = ifx_lo_open(&descr, LO_RDONLY, &error);
        if (error < 0)
        {
            strcpy(statement, "ifx_lo_open()");
            handle_lo_error(error);
        }
    /* Use the LO file descriptor to read the data in the
    * smart large object.
    */
```

```
        ifx_int8cvint(0, &offset);
        strcpy(buf, "");
        numbytes = ifx_lo_readwithseek(lofd, buf, buflen,
            &offset, LO_SEEK_CUR, &error);
        if (error || numbytes == 0)
        {
            strcpy(statement, "ifx_lo_readwithseek()");
            handle_lo_error(error);
        }
        if(ifx_int8toint(&catalog_num, &ic_num) !=  0)
            printf("\nifx_int8toint failed to convert catalog_num to int");
        printf("\nContents of column \'descr\' for catalog_num:
        %d \n\t%s\n",
            ic_num, buf);
        /* Close open smart large object */
        ifx_lo_close(lofd);
        /* Free LO-specification structure */
        ifx_lo_spec_free(create_spec);
}

void handle_lo_error(error_num)
int error_num;
{
    printf("%s generated error %d\n", statement, error_num);
    exit(1);
}

void nullterm(str)
char *str;
{
    char *end;

    end = str + 256;
    while(*str != ' ' && *str != '\0' && str < end)
    {
        ++str;
    }
    if(str >= end)
        printf("Error: end of str reached\n");
    if(*str == ' ')
        *str = '\0';
}

/* Include source code for whenexp_chk() exception-checking
 * routine
 */

EXEC SQL include exp_chk.ec;
```

The get_lo_info.ec program

This program retrieves information about smart large objects stored in a BLOB column.

```
#include <time.h>

EXEC SQL define BUFSZ 10;

extern char statement[80];

main()
{
    int error, ic_num, oflags, cflags, extsz, imsize, isize, iebytes;
    time_t time;
```

```
                struct tm *date_time;
                char col_name[300], sbspc[129];

                EXEC SQL BEGIN DECLARE SECTION;
                fixed binary 'blob' ifx_lo_t picture;
                char srvr_name[256];
                ifx_lo_create_spec_t *cspec;
                ifx_lo_stat_t *stats;
                ifx_int8_t size, c_num, estbytes, maxsize;
                int lofd;
                long atime, ctime, mtime, refcnt;
                EXEC SQL END DECLARE SECTION;

                void nullterm(char *);
                void handle_lo_error(int);

                imsize = isize = iebytes = 0;
                EXEC SQL whenever sqlerror call whenexp_chk;
                EXEC SQL whenever sqlwarning call whenexp_chk;

                printf("GET_LO_INFO Sample ESQL program running.\n\n");
                strcpy(statement, "CONNECT stmt");
                EXEC SQL connect to 'stores7';
                EXEC SQL get diagnostics exception 1
                    :srvr_name = server_name;
                nullterm(srvr_name);

                EXEC SQL declare ifxcursor cursor for
                    select catalog_num, advert.picture
                    into :c_num, :picture
                    from catalog
                    where advert.picture is not null;

            EXEC SQL open ifxcursor;
                while(1)
                {
                    EXEC SQL fetch ifxcursor;
                    if (strncmp(SQLSTATE, "00", 2) != 0)
                    {
                        if(strncmp(SQLSTATE, "02", 2) != 0)
                            printf("SQLSTATE after fetch is %s\n", SQLSTATE);
                        break;
                    }
                    /* Use the returned LO-pointer structure to open a smart
                     * large object and get an LO file descriptor.
                     */
                    lofd = ifx_lo_open(&picture, LO_RDONLY, &error);
                    if (error < 0)
                    {
                        strcpy(statement, "ifx_lo_open()");
                        handle_lo_error(error);
                    }
                    if(ifx_lo_stat(lofd, &stats) < 0)
                  {
                        printf("\nifx_lo_stat() < 0");
                        break;
                    }
                    if(ifx_int8toint(&c_num, &ic_num) !=  0)
                        ic_num = 99999;
                    if((ifx_lo_stat_size(stats, &size)) < 0)
                        isize = 0;
                    else
                        if(ifx_int8toint(&size, &isize) !=  0)
                        {
```

```
                printf("\nFailed to convert size");
                isize = 0;
            }
        if((refcnt = ifx_lo_stat_refcnt(stats)) < 0)
            refcnt = 0;
        printf("\n\nCatalog number %d", ic_num);
        printf("\nSize is %d, reference count is %d", isize, refcnt);

        if((atime = ifx_lo_stat_atime(stats)) < 0)
            printf("\nNo atime available");
        else
        {
            time = (time_t)atime;
            date_time = localtime(&time);
            printf("\nTime of last access: %s", asctime(date_time));
        }
        if((ctime = ifx_lo_stat_ctime(stats)) < 0)
            printf("\nNo ctime available");
        else
        {
            time = (time_t)ctime;
            date_time = localtime(&time);
            printf("Time of last change: %s", asctime(date_time));
        }

    if((mtime = ifx_lo_stat_mtime_sec(stats)) < 0)
            printf("\nNo mtime available");
        else
        {
            time = (time_t)mtime;
            date_time = localtime(&time);
         printf("Time to the second of last modification: %s",
            asctime(date_time));
        }
      if((cspec = ifx_lo_stat_cspec(stats)) == NULL)
        {
            printf("\nUnable to access ifx_lo_create_spec_t structure");
            break;
        }
        oflags = ifx_lo_specget_def_open_flags(cspec);
        printf("\nDefault open flags are: %d", oflags);
        if(ifx_lo_specget_estbytes(cspec, &estbytes) == -1)
        {
            printf("\nifx_lo_specget_estbytes() failed");
            break;
        }
        if(ifx_int8toint(&estbytes, &iebytes) !=  0)
        {
            printf("\nFailed to convert estimated bytes");
        }
        printf("\nEstimated size of smart LO is: %d", iebytes);
        if((extsz = ifx_lo_specget_extsz(cspec)) == -1)
    {
            printf("\nifx_lo_specget_extsz() failed");
            break;
        }
        printf("\nAllocation extent size of smart LO is: %d", extsz);
        if((cflags = ifx_lo_specget_flags(cspec)) == -1)
        {
            printf("\nifx_lo_specget_flags() failed");
            break;
        }
        printf("\nCreate-time flags of smart LO are: %d", cflags);
        if(ifx_lo_specget_maxbytes(cspec, &maxsize) == -1)
```

```
                    {
                        printf("\nifx_lo_specget_maxsize() failed");
                        break;
                    }
                    if(ifx_int8toint(&maxsize, &imsize) !=  0)
                    {
                        printf("\nFailed to convert maximum size");
                        break;
                    }
                    if(imsize == -1)
                        printf("\nMaximum size of smart LO is: No limit");
                    else
                        printf("\nMaximum size of smart LO is: %d\n", imsize);
                    if(ifx_lo_specget_sbspace(cspec, sbspc, sizeof(sbspc)) == -1)
                        printf("\nFailed to obtain sbspace name");
                    else
                        printf("\nSbspace name is %s\n", sbspc);

            }
                /* Close smart large object */
                ifx_lo_close(lofd);
                ifx_lo_stat_free(stats);
            EXEC SQL close ifxcursor;
                EXEC SQL free ifxcursor;
            }

        void handle_lo_error(error_num)
        int error_num;
        {
            printf("%s generated error %d\n", statement, error_num);
            exit(1);
        }

        void nullterm(str)
        char *str;
        {
            char *end;

            end = str + 256;
            while(*str != ' ' && *str != '\0' && str < end)
            {
                ++str;
            }
            if(str >= end)
                printf("Error: end of str reached\n");
            if(*str == ' ')
                *str = '\0';
        }
        /* Include source code for whenexp_chk() exception-checking
         * routine
         */

        EXEC SQL include exp_chk.ec;
```

The upd_lo_descr.ec program

This program obtains the price of catalog items for which the **advert_descr** column is not null and appends the price to the description.

```
EXEC SQL include sqltypes;

EXEC SQL define BUFSZ 10;

extern char statement[80];
```

```
main()
{
    int error, isize;
    char format[] =  "<<<,<<<.&&";
    char decdsply[20], buf[50000], *end;

    EXEC SQL BEGIN DECLARE SECTION;
    dec_t price;
    fixed binary 'clob' ifx_lo_t descr;
    smallint stockno;
    char srvr_name[256], mancd[4], unit[5];
    ifx_lo_stat_t *stats;
    ifx_int8_t size, offset, pos;
    int lofd, ic_num;
    EXEC SQL END DECLARE SECTION;

    void nullterm(char *);
    void handle_lo_error(int);

    isize = 0;
    EXEC SQL whenever sqlerror call whenexp_chk;
    EXEC SQL whenever sqlwarning call whenexp_chk;

    printf("UPD_LO_DESCR Sample ESQL program running.\n\n");
    strcpy(statement, "CONNECT stmt");
    EXEC SQL connect to 'stores7';
    EXEC SQL get diagnostics exception 1
        :srvr_name = server_name;
    nullterm(srvr_name);

/* Selects each row where the advert.picure column is not null and
* displays
*  status information for the smart large object.
*/
    EXEC SQL declare ifxcursor cursor for
        select catalog_num, stock_num, manu_code, unit, advert_descr
        into :ic_num, :stockno, :mancd, :unit, :descr
        from catalog
        where advert_descr is not null;

EXEC SQL open ifxcursor;
    while(1)
    {
        EXEC SQL fetch ifxcursor;
        if (strncmp(SQLSTATE, "00", 2) != 0)
        {
            if(strncmp(SQLSTATE, "02", 2) != 0)
                printf("SQLSTATE after fetch is %s\n", SQLSTATE);
            break;
      }
        EXEC SQL select unit_price into :price
            from stock
            where stock_num = :stockno
            and manu_code = :mancd
            and unit = :unit;
        if (strncmp(SQLSTATE, "00", 2) != 0)
        {
            printf("SQLSTATE after select on stock: %s\n", SQLSTATE);
            break;
        }
        if(risnull(CDECIMALTYPE, (char *) &price))  /* NULL? */
            continue;       /* skip to next row */
        rfmtdec(&price, format, decdsply); /* format unit_price */
```

```
                   /* Use the returned LO-pointer structure to open a smart
                    * large object and get an LO file descriptor.
                    */
                   lofd = ifx_lo_open(&descr, LO_RDWR, &error);
                   if (error < 0)
                   {
                       strcpy(statement, "ifx_lo_open()");
                       handle_lo_error(error);
                   }
                   ifx_int8cvint(0, &offset);
                   if(ifx_lo_seek(lofd, &offset, LO_SEEK_SET, &pos) < 0)
                   {
                       printf("\nifx_lo_seek() < 0\n");
                       break;
                   }
                   if(ifx_lo_stat(lofd, &stats) < 0)
                   {
                       printf("\nifx_lo_stat() < 0");
                       break;
                   }
                   if((ifx_lo_stat_size(stats, &size)) < 0)
                   {
                       printf("\nCan't get size, isize = 0");
                       isize = 0;
                   }
                   else
                       if(ifx_int8toint(&size, &isize) !=  0)
                       {
                           printf("\nFailed to convert size");
                           isize = 0;
                       }
                   if(ifx_lo_read(lofd, buf, isize, &error) < 0)
                   {
                       printf("Read operation failed\n");
                       break;
                   }
           end = buf + isize;
                   strcpy(end++, "(");
                   strcat(end, decdsply);
                   end += strlen(decdsply);
                   strcat(end++, ")");
                   if(ifx_lo_writewithseek(lofd, buf, (end - buf), &offset,
                   LO_SEEK_SET,
                       &error) < 0)
                   {
                       printf("Write error on LO: %d", error);
                       continue;
                   }
               printf("\nNew description for catalog_num %d is: \n%s\n", ic_num,
               buf);
           }
           /* Close smart large object */
           ifx_lo_close(lofd);
           ifx_lo_stat_free(stats);
           /* Free LO-specification structure */
           EXEC SQL close ifxcursor;
           EXEC SQL free ifxcursor;
       }

       void handle_lo_error(error_num)
       int error_num;
       {
           printf("%s generated error %d\n", statement, error_num);
           exit(1);
```

```
}

void nullterm(str)
char *str;
{
    char *end;

    end = str + 256;
    while(*str != ' ' && *str != '\0' && str < end)
    {
        ++str;
    }
    if(str >= end)
        printf("Error: end of str reached\n");
    if(*str == ' ')
        *str = '\0';
}

/* Include source code for whenexp_chk() exception-checking
 * routine
 */

EXEC SQL include exp_chk.ec;
```

Appendix D. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft Windows navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format. For more information about the dotted decimal format, go to "Dotted Decimal Syntax Diagrams."

You can view the publications for IBM Informix in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Dotted Decimal Syntax Diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

?   Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

!   Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines

2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*   Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data-area, you know that you can include more than one data area or you can include none. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.

2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.

3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

+   Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A.

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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

S

IBM ®

Printed in USA

Spine information:

IBM Informix   **Version 3.50**

IBM Informix ESQL/C Programmer's Manual

IBM