

INFORMIX[®]-ESQL/C

Programmer's Manual

INFORMIX-OnLine Dynamic Server, Version 7.2x
INFORMIX-OnLine Workgroup Server, Version 7.2x
INFORMIX-OnLine XPS, Version 8.1x
INFORMIX-SE, Version 7.2x,
INFORMIX-Universal Server, Version 9.1x

Version 9.13
October 1997
Part No. 000-4161

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

Copyright © 1981-1997 by Informix Software, Inc. or their subsidiaries, provided that portions may be copyrighted by third parties, as set forth in documentation. All rights reserved.

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

INFORMIX®; INFORMIX®-OnLine Dynamic Server™; DataBlade®

The following are worldwide trademarks of the indicated owners or their subsidiaries, registered in the United States of America as indicated by “®,” and in numerous other countries worldwide:

Adobe Systems Incorporated: PostScript®

Hewlett-Packard: Hewlett-Packard®, HP®

Microsoft Corporation: Microsoft®, MS®, MS-DOS®, CodeView®, MS Windows™; Windows™; Windows NT™; ODBC™; Visual Basic™; Visual C++™

Microsoft Memory Management Product: HIMEM.SYS

(“DOS” as used herein refers to MS-DOS and/or PC-DOS operating systems.)

Open Software Foundation, Inc.: Open Software Foundation™; OSF™; OSF™ DCE

Sun Microsystems, Inc.: Sun™

X/OpenCompany Ltd.: UNIX®, X/Open®

All other marks or symbols are registered trademarks or trademarks of their respective owners.

Documentation Team: Bob Berry, Twila Booth, Evelyn Eldridge-Diaz, John Francis, Smita Joshi

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

RESTRICTED RIGHTS/SPECIAL LICENSE RIGHTS

Software and documentation acquired with US Government funds are provided with rights as follows: (1) if for civilian agency use, with Restricted Rights as defined in FAR 52.227-19; (2) if for Dept. of Defense use, with rights as restricted by vendor's standard license, unless superseded by negotiated vendor license as prescribed in DFAR 227.7202. Any whole or partial reproduction of software or documentation marked with this legend must reproduce the legend.

Table of Contents

Introduction

About This Manual	3
Types of Users	3
Software Dependencies	4
Assumptions About Your Locale.	4
Demonstration Database	5
Major Features	5
Documentation Conventions	6
Typographical Conventions	7
Icon Conventions	8
Command-Line Conventions	10
Additional Documentation	12
On-Line Manuals	13
Printed Manuals	13
Error Message Files	13
Documentation Notes, Release Notes, Machine Notes	14
Compliance with Industry Standards	15
Informix Welcomes Your Comments	15

Section I What Is INFORMIX-ESQL/C?

Chapter 1 Programming with INFORMIX-ESQL/C

What Is INFORMIX-ESQL/C?	1-5
ESQL/C Executable Files for Windows	1-6
ESQL/C Library Functions	1-7
Creating an ESQL/C Program	1-8

Compiling an ESQL/C Program	1-9
Default Compilation Order	1-11
Running the C Preprocessor First	1-12
Using ESQL/C Preprocessor Directives	1-12
The include Directive	1-13
The define and undef Directives	1-14
The ifdef, ifndef, elif, else, and endif Directives	1-16
Embedding SQL Statements	1-17
Handling Case Sensitivity in Embedded SQL Statements	1-18
Using Quotes and Escape Characters	1-19
Adding Comments	1-21
Specifying Host Variables	1-21
Using ESQL/C Header Files	1-22
Declaring and Using Host Variables in SQL Statements	1-27
Declaring a Host Variable	1-27
Assigning a Value to a Host Variable	1-32
Using Host Variables in Data Structures	1-36
Using Indicator Variables	1-42
Using the esql Command	1-47
Requirements for Using esql	1-48
Syntax of the esql Command.	1-48
Options That Affect Preprocessing.	1-58
Running the C Preprocessor Before the ESQL/C Preprocessor	1-66
Preprocessor Options Specific to Windows Environments.	1-73
Compiling and Linking Options of the esql Command	1-75
Syntax of the Compiling/Linking Options for UNIX	1-75
Syntax of the Compiling/Linking Options for Windows Environments.	1-77
Special Compile Options for Windows Environments	1-81
General Options That Affect Linking	1-86
Special Linking Options for Windows Environments	1-87
Windows Operating System Processor Options Available to the esql Command	1-90
Accessing the ESQL Client-Interface DLL in Windows Environments.	1-94

Setting and Retrieving Environment Variables in Windows	
Environments	1-97
ifx_putenv()	1-100
ifx_getenv()	1-102
InetLogin Structure	1-103
Using Host Variables in Windows Environments	1-112
Declaring Variables with Non-ANSI Storage-Class	
Modifiers.	1-112
Using Global ESQL/C Variables	1-115
A Sample INFORMIX-ESQL/C Program	1-116
Compiling the demo1 Program	1-116
Guide to demo1.ec File	1-116

Chapter 2 **INFORMIX-ESQL/C Data Types**

Choosing Data Types for Host Variables	2-3
Defined Constants for Data Types	2-7
Header Files for Data Types	2-11
The BOOLEAN Data Type	2-13
Data Conversion	2-15
Fetching and Inserting with Host Variables	2-15
Arithmetic Operations	2-19
Data-Type Alignment Library Functions	2-23
risnull()	2-24
rsetnull()	2-27
rtypalign()	2-30
rtypsize()	2-33
rtypname()	2-36
rtypwidth()	2-39

Chapter 3 **Working with Character and String Data Types**

Character Data Types	3-3
The char Data Type	3-4
The fixchar Data Type	3-5
The string Data Type	3-5
The varchar Data Type	3-6
The lvarchar Data Type	3-9
Fetching and Inserting Character Data Types	3-11
Fetching and Inserting CHAR Data	3-11
Fetching and Inserting VARCHAR Data	3-12
Fetching and Inserting lvarchar Data	3-17
Fetching and Inserting with an ANSI-Compliant Database	3-18

Character and String Library Functions	3-19
bcmp()	3-21
bcopy()	3-24
bfill()	3-26
byleng()	3-28
ldchar()	3-30
rdownshift()	3-32
rstd()	3-34
rstoi()	3-36
rstol()	3-38
rupshift()	3-40
stcat()	3-42
stchar()	3-44
stcmp()	3-46
scopy()	3-48
stleng()	3-50

Chapter 4 **Working with the DECIMAL Data Type**

The SQL DECIMAL Data Type	4-3
The decimal Data Type	4-4
Implicit Data Conversion for decimal Values	4-7
Operations on decimal Values	4-7
Data Conversion for decimal Values	4-7
The decimal Library Functions	4-8
decadd()	4-9
deccmp()	4-11
deccopy()	4-13
deccvasc()	4-15
deccvdbl()	4-18
deccvint()	4-20
deccvlong()	4-22
decdiv()	4-24
dececv() and decfcvt()	4-26
decmul()	4-32
decround()	4-34
decsub()	4-37
dectoasc()	4-39
dectodbl()	4-42
dectoint()	4-44
dectolong()	4-47
dectrunc()	4-49

Chapter 5

Working with Numeric Data Types

The Integer Data Types	5-4
The Floating-Point Data Types	5-5
Declaring float Host Variables	5-5
Implicit Data Conversion	5-6
The INT8 and SERIAL8 SQL Data Types	5-7
The int8 Data Type.	5-7
Implicit Data Conversion for int8 Values	5-8
Operations on int8 Values	5-8
Data Conversion for int8 Values	5-8
The int8 Library Functions	5-9
ifx_getserial8()	5-10
ifx_int8add()	5-12
ifx_int8cmp()	5-13
ifx_int8copy()	5-14
ifx_int8cvasc()	5-15
ifx_int8cvdbl()	5-16
ifx_int8cvdec()	5-17
ifx_int8cvint()	5-18
ifx_int8cvlong()	5-19
ifx_int8div()	5-20
ifx_int8mul()	5-21
ifx_int8sub()	5-22
ifx_int8toasc()	5-23
ifx_int8todbl()	5-24
ifx_int8todec()	5-25
ifx_int8toint()	5-26
ifx_int8tolong()	5-27
Formatting Numeric Strings	5-28
Numeric-Formatting Functions	5-34
rfmtdec()	5-35
rfmtdouble()	5-38
rfmtlong()	5-41

Chapter 6

Working with Time Data Types

The SQL DATE Data Type	6-3
Formatting Date Strings	6-4
DATE Library Functions	6-5
rdatestr()	6-6
Return Codes	6-6
rdayofweek()	6-8

rdefmtdate()	6-10
rfmtdate()	6-14
rjulmdy()	6-17
rleapyear()	6-19
rmdyjul()	6-21
rstrdate()	6-23
rtoday()	6-26
The SQL DATETIME and INTERVAL Data Types	6-28
The datetime Data Type	6-29
The interval Data Type	6-30
Macros for datetime and interval Data Types	6-31
Fetching and Inserting DATETIME and INTERVAL Values	6-33
ANSI SQL Standards for DATETIME and INTERVAL Values	6-36
Operations on datetime and interval Values	6-36
Data Conversion for datetime and interval Values	6-37
DATETIME and INTERVAL Library Functions	6-38
dtaddinv()	6-40
dtcurrent()	6-42
dctvasc()	6-44
dctvfmtasc()	6-47
dtextend()	6-51
dtsub()	6-54
dtsubinv()	6-57
dttoasc()	6-59
dttofmasc()	6-62
incvasc()	6-65
incvfmtasc()	6-68
intoasc()	6-71
intofmtasc()	6-74
invdivdbl()	6-77
invdivinv()	6-80
invxtend()	6-82
invmuldbl()	6-85

Chapter 7 Working with INFORMIX-Universal Server Smart Large Objects

Data Structures for Smart Large Objects	7-6
Declaring a Host Variable	7-7
The LO-Specification Structure	7-7
The LO-Pointer Structure	7-16
The LO File Descriptor	7-20

Creating a Smart Large Object	7-21
Accessing a Smart Large Object	7-22
Selecting a Smart Large Object	7-22
Opening a Smart Large Object.	7-23
Deletion of a Smart Large Object	7-28
Modifying a Smart Large Object	7-28
Closing a Smart Large Object	7-29
Obtaining Status of a Smart Large Object	7-29
Obtaining a Valid LO-Pointer Structure	7-30
Allocating and Accessing an LO-Status Structure	7-30
Deallocating the LO-Status Structure	7-32
Reading and Writing Smart Large Objects on an Optical Disc	7-32
The ESQL / C API for Smart Large Objects	7-33
ifx_lo_alter()	7-36
ifx_lo_close().	7-38
ifx_lo_col_info()	7-39
ifx_lo_copy_to_file()	7-41
ifx_lo_copy_to_lo()	7-44
ifx_lo_create()	7-46
ifx_lo_def_create_spec()	7-49
ifx_lo_filename()	7-50
ifx_lo_from_buffer()	7-52
ifx_lo_open().	7-53
ifx_lo_read()	7-55
ifx_lo_readwithseek().	7-57
ifx_lo_release()	7-59
ifx_lo_seek()	7-61
ifx_lo_spec_free()	7-63
ifx_lo_specget_estbytes()	7-64
ifx_lo_specget_extsz()	7-66
ifx_lo_specget_flags().	7-67
ifx_lo_specget_maxbytes()	7-69
ifx_lo_specget_sbospace().	7-71
ifx_lo_specset_estbytes().	7-73
ifx_lo_specset_extsz().	7-75
ifx_lo_specset_flags().	7-77
ifx_lo_specset_maxbytes()	7-79
ifx_lo_specset_sbospace().	7-80
ifx_lo_stat()	7-81
ifx_lo_stat_atime().	7-83
ifx_lo_stat_cspec().	7-84
ifx_lo_stat_ctime().	7-86

ifx_lo_stat_free()	7-87
ifx_lo_stat_mtime_sec()	7-88
ifx_lo_stat_refcnt()	7-89
ifx_lo_stat_size()	7-91
ifx_lo_tell()	7-92
ifx_lo_to_buffer()	7-93
ifx_lo_truncate()	7-95
ifx_lo_write()	7-96
ifx_lo_writewithseek()	7-98
The create_clob Program	7-100
Storage Characteristics for the Example	7-100
Source for the create_clob.ec File	7-101

Chapter 8 Working with Simple Large Objects

Choosing a Large-Object Data Type	8-4
Programming with Simple Large Objects	8-5
Declaring a Host Variable for a Simple Large Object.	8-5
Accessing the Locator Structure.	8-6
Locating Simple Large Objects in Memory	8-11
Allocating the Memory Buffer	8-12
Selecting a Simple Large Object into Memory	8-15
Inserting a Simple Large Object from Memory.	8-17
Locating Simple Large Objects in Files	8-19
Locating Simple Large Objects in Open Files	8-21
Locating Simple Large Objects in Named Files	8-26
Using User-Defined Simple-Large-Object Locations	8-31
Selecting a Simple Large Object into a User-Defined Location	8-33
Inserting a Simple Large Object into a User-Defined Location	8-34
Creating the User-Defined Simple-Large-Object Functions	8-35
Reading and Writing Simple Large Objects to an Optical Disc	8-40
The dispcat_pic Program	8-44
Loading the Simple-Large-Object Images	8-45
Compiling the dispcat_pic Program	8-48
Guide to the dispcat_pic.ec File	8-50
Guide to the prdesc.c File	8-66
Guide to the inpfuns.c File	8-68

Chapter 9

Working with INFORMIX-Universal Server Complex Data Types

Accessing a Collection	9-4
Declaring Collection Variables.	9-5
Managing Memory for Collections	9-12
Operating on a Collection Variable	9-13
Saving into a Collection Column	9-40
Operating on a Collection Column	9-41
ifx_cl_card()	9-45
Accessing Row Types.	9-48
Declaring Row Variables	9-49
Managing Memory for Rows	9-57
Operating on a Row Variable	9-58
Saving into a Row-Type Column	9-71
Accessing a Typed Table.	9-72
Operating on a Row-Type Column	9-73

Chapter 10

Working with INFORMIX-Universal Server Opaque Data Types

The SQL Opaque Data Type	10-3
Accessing the External Format of an Opaque Type	10-6
Declaring lvarchar Host Variables	10-7
Using lvarchar Host Variables.	10-11
Accessing the Internal Format of an Opaque Type	10-17
Accessing a Fixed-Length Opaque Type	10-18
Accessing a Varying-Length Opaque Type	10-23
The lvarchar pointer and var binary Library Functions	10-32
ifx_var_alloc()	10-33
ifx_var_dealloc()	10-35
ifx_var_flag()	10-37
ifx_var_getdata()	10-39
ifx_var_getlen().	10-40
ifx_var_isnull()	10-41
ifx_var_setdata()	10-42
ifx_var_setlen()	10-44
ifx_var_setnull()	10-46
Accessing Predefined Opaque Data Types	10-48

Section II Database Server Communication

Chapter 11 Exception Handling

Obtaining Diagnostic Information After an SQL Statement	11-4
Types of Diagnostic Information	11-4
Types of Status Variables	11-6
Exception Handling with SQLSTATE	11-6
Using GET DIAGNOSTICS	11-7
Using the SQLSTATE Variable	11-12
Checking for Exceptions with SQLSTATE	11-19
Exception Handling with the sqlca Structure	11-27
Fields of the sqlca Structure	11-28
Using the SQLCODE Variable	11-32
Checking for Exceptions with sqlca	11-34
Displaying Error Text	11-40
Choosing an Exception-Handling Strategy	11-41
Checking After Each SQL Statement	11-41
The WHENEVER Statement	11-45
Library Functions for Retrieving Error Messages	11-47
rgetlmsg()	11-48
rgetmsg()	11-51
Displaying Error Text in a Windows Environment	11-54
A Program That Uses Exception Handling	11-55
Compiling the Program	11-55
Guide to the getdiag.ec File	11-56
Guide to the exp_chk.ec File	11-59

Chapter 12 Working with the Database Server

The Client-Server Architecture of ESQL/C Applications	12-3
The Client-Server Connection	12-5
Sources of Connection Information on a UNIX Operating System	12-6
Sources of Connection Information in a Windows Environment	12-7
Connecting to a Database Server	12-15
Identifying the Database Server.	12-23
Interacting with the Database Server	12-26
Determining Features of the Database Server	12-26
Switching Between Multiple Database Connections.	12-28
Identifying an Explicit Connection.	12-30

Obtaining Available Databases	12-32
Checking the Status of the Database Server	12-32
Detaching from a Connection	12-32
Interrupting an SQL Request	12-33
Terminating a Connection	12-40
Optimized Message Transfers	12-41
Restrictions on Optimized Message Transfers	12-41
Enabling Optimized Message Transfers	12-42
Error Handling with Optimized Message Transfers	12-44
Using Database Server Control Functions	12-46
ifx_getcur_conn_name().	12-47
sqgetdbs().	12-48
sqlbreak().	12-51
sqlbreakcallback().	12-53
sqldetach().	12-56
sqldone().	12-63
sqlexit().	12-64
sqlsignal().	12-65
sqlstart().	12-67
The timeout Program	12-69
Compiling the Program	12-69
Guide to the timeout.ec File	12-70
Example Output	12-88
Using ESQL / C Connection Library Functions in a Windows	
Environment	12-92
GetConnect().	12-94
ReleaseConnect().	12-97
SetConnect().	12-99

Chapter 13 Using Informix Libraries

Choosing a Version of the Informix General Libraries	13-5
The Informix General Libraries	13-5
The esql Command	13-6
Linking Static Informix General Libraries	13-7
Linking Shared Informix General Libraries	13-8
Choosing Between Shared and Static Versions	13-10
Compatibility of Pre-existing ESQL / C Applications with	
Current Library Versions	13-12
Using the ifx_getversion Utility	13-13
Checking the API Version of a Library	13-14

Creating Thread-Safe ESQL/C Applications	13-16
Characteristics of Thread-Safe ESQL/C Code	13-17
Programming a Thread-Safe ESQL/C Application	13-17
Linking Thread-Safe Libraries	13-26
Using ESQL/C Thread-Safe Decimal Functions	13-30
ifx_dececv(), ifx_deccvt()	13-31
A Sample Thread-Safe Program	13-33
Source Listing	13-33
Sample Output	13-39
Creating a Dynamic Thread Library on UNIX Operating Systems	13-40
Data Types	13-43
Registering the Dynamic Thread Functions.	13-47
Setting the \$THREADLIB Environment Variable	13-50
Creating the Shared Library	13-50
Compiling with the -thread and -l Preprocessor Options	13-51

Section III Dynamic SQL

Chapter 14 Using Dynamic SQL

Using Dynamic SQL	14-4
Assembling and Preparing the SQL Statement.	14-5
Executing the SQL Statement	14-10
Freeing Resources	14-13
Using a Database Cursor	14-14
Receiving More Than One Row	14-14
Sending More Than One Row	14-18
Naming the Cursor	14-19
Optimizing Cursor Execution	14-20
Optimizing OPEN, FETCH, and CLOSE	14-35
Restrictions on OPTOFC	14-36
Enabling the OPTOFC Feature	14-37
Using OPTOFC and Deferred-PREPARE Together	14-38
SQL Statements That Are Known at Compile Time	14-40
Executing Non-SELECT Statements	14-40
Executing SELECT Statements	14-43
Executing User-Defined Routines in Universal Server	14-46
Executing Statements with Input Parameters	14-51
SQL Statements That Are Not Known at Compile Time	14-58

Chapter 15

Determining SQL Statements

Using Dynamic-Management Structures	15-4
A System-Descriptor Area	15-5
An sqlda Structure.	15-10
Using the DESCRIBE Statement	15-14
Determining Statement Type	15-16
Determining the Data Type of a Column	15-19
Checking for a WHERE Clause	15-22
Determining Statement Information at Runtime	15-22
Handling an Unknown Select List	15-23
Handling an Unknown Column List	15-24
Determining Unknown Input Parameters.	15-25
Determining Return Values Dynamically	15-26
Handling Statements That Contain User-Defined Data Types	15-27
Using a Fetch Array	15-33
Allocating Memory for the Fetch Arrays	15-37
Obtaining Values from Fetch Arrays	15-42
Freeing Memory for a Fetch Array	15-45

Chapter 16

Using a System-Descriptor Area

Managing a System-Descriptor Area	16-4
Allocating Memory for a System-Descriptor Area	16-5
Initializing the System-Descriptor Area	16-6
Assigning and Obtaining Values from a System-Descriptor Area	16-8
Specifying Input Parameter Values	16-10
Putting Column Values into a System-Descriptor Area	16-11
Freeing Memory Allocated to a System-Descriptor Area.	16-12
Using a System-Descriptor Area	16-13
Handling an Unknown Select List	16-13
Executing a SELECT That Returns Multiple Rows	16-15
Executing a Singleton SELECT	16-22
Handling Unknown Return Values	16-22
Executing a Noncursor Function	16-23
Executing a Cursor Function	16-28
Handling an Unknown Column List	16-29
Executing a Simple Insert	16-30
Executing an INSERT That Is Associated with a Cursor	16-35

Handling a Parameterized SELECT Statement	16-36
Executing a Parameterized SELECT That Returns Multiple Rows.	16-37
Executing a Parameterized Singleton SELECT.	16-43
Handling a Parameterized User-Defined Routine	16-43
Executing a Parameterized Function	16-44
Executing a Parameterized Procedure	16-45
Handling a Parameterized UPDATE or DELETE Statement	16-46
The dyn_sql Program	16-47
Compiling the Program	16-47
Guide to the dyn_sql.ec File	16-48

Chapter 17 **Using an sqlda Structure**

Managing an sqlda Structure	17-4
Defining an sqlda Structure	17-5
Allocating Memory for the sqlda Structure	17-6
Initializing the sqlda Structure	17-7
Allocating Memory for Column Data	17-10
Assigning and Obtaining Values from an sqlda Structure . . .	17-13
Specifying Input Parameter Values	17-14
Putting Column Values into an sqlda Structure	17-14
Freeing Memory Allocated to an sqlda Structure	17-15
Using an sqlda Structure	17-16
Handling an Unknown Select List	17-17
Executing a SELECT That Returns Multiple Rows	17-18
Executing a Singleton SELECT	17-26
Handling Unknown Return Values	17-26
Executing a Noncursor Function	17-28
Executing a Cursor Function.	17-28
Handling an Unknown Column List	17-30
Executing a Simple Insert	17-31
Executing an INSERT That Is Associated with a Cursor	17-31
Handling a Parameterized SELECT Statement	17-32
Executing a Parameterized SELECT That Returns Multiple Rows.	17-34
Executing a Parameterized Singleton SELECT.	17-41
Handling a Parameterized User-Defined Routine	17-41
Executing a Parameterized Function	17-42
Executing a Parameterized Procedure	17-42
Handling a Parameterized UPDATE or DELETE Statement	17-43

Appendix A	List of ESQL/C Library Functions
Appendix B	The Demonstration Database
	Index

Introduction

About This Manual.	3
Types of Users	3
Software Dependencies	4
Assumptions About Your Locale.	4
Demonstration Database	5
Major Features	5
Documentation Conventions	6
Typographical Conventions	7
Icon Conventions	8
Comment Icons	8
Feature, Product, and Platform Icons	9
Command-Line Conventions	10
How to Read a Command-Line Diagram	12
Additional Documentation	12
On-Line Manuals	13
Printed Manuals	13
Error Message Files	13
Documentation Notes, Release Notes, Machine Notes	14
Compliance with Industry Standards	15
Informix Welcomes Your Comments.	15

Read this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

This manual explains how to use INFORMIX-ESQL/C, the Informix implementation of embedded Structured Query Language (SQL) for C, to create client applications with database-management capabilities. This manual is a complete guide to the features of ESQL/C that enable you to interact with the database server, access databases, manipulate the data in your program, and check for errors.

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

Types of Users

This manual is for C programmers who want to use ESQL/C to embed SQL statements in their programs. The manual assumes that you know C programming and are familiar with the structure of relational databases.

If you have limited experience with relational databases, SQL, or your operating system, see the [Getting Started](#) manual for your database server for a list of supplementary manuals.

Software Dependencies

This manual assumes that you are using INFORMIX-ESQL/C, Version 9.13, on either a Windows NT, Windows 95, or UNIX platform. In addition, you must use one of the following database servers:

- INFORMIX-OnLine Dynamic Server, Version 7.2x
- INFORMIX-OnLine Workgroup Server, Version 7.2x
- INFORMIX-OnLine XPS, Version 8.1x
- INFORMIX-SE, Version 7.2x
- INFORMIX-Universal Server, Version 9.1x

Assumptions About Your Locale

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.

This manual assumes 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(s). For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the [INFORMIX-GLS Programmer's Manual](#) and the [Informix Guide to GLS Functionality](#).

Demonstration Database

Your Informix software includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. Informix includes source files for INFORMIX-ESQL/C demonstration programs that access and manipulate the data stored in **stores7**. Most of the examples in this manual are based on the **stores7** demonstration database.

For information on how to install **stores7** and access the code examples, see [Appendix B, “The Demonstration Database,”](#) of this manual.

Major Features

The Introduction to each Informix product manual contains a list of major features for that product. Major features for Informix products also appear in release notes.

Informix has implemented the following major new features in Version 9.13 of INFORMIX-ESQL/C:

- The ability to have the C preprocessor run prior to the **esql** preprocessor. This ability makes the following functionalities possible:
 - The ability to declare any valid C syntax inside ESQL/C BEGIN DECLARE-END DECLARE sections
 - The ability to declare ANSI-C style function parameters as host variables
 - Ability to define host variables with **#define** and **typedef** statements defined in either C library header files or in user-created C language header files
- Support for user-defined dynamic thread libraries

- The following functions:
 - **ifx_cl_card()**
Returns the number of elements in a collection
 - **ifx_lo_from_buffer()**
Copies bytes from a user-defined buffer into smart large object
 - **ifx_lo_release()**
Releases resources associated with a temporary smart large object
 - **ifx_lo_to_buffer()**
Copies bytes from smart large object into a user-defined buffer
- The **lvvarchar** pointer variable, which enables you to insert or select opaque type columns that can be represented in a character string format and which can vary in size from one row in the table to another

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Command-line conventions

Typographical Conventions

This manual uses the following standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
<i>italics</i>	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
monospace	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
◆	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information.



Tip: When you are instructed to “enter” characters or to “execute” a command, immediately press RETURN after the entry. When you are instructed to “type” the text or to “press” other keys, no RETURN is required.

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
	The <i>warning</i> icon identifies vital instructions, cautions, or critical information.
	The <i>important</i> icon identifies significant information about the feature or operation that is being described.
	The <i>tip</i> icon identifies additional details or shortcuts for the functionality that is being described.

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-, product-, or platform-specific information.

Icon	Description
	Identifies information that is specific to a user-defined routine (UDR) that is written in an external language (such as C)
	Identifies information that is specific to the Informix Global Language Support (GLS) feature.
	Identifies information that is specific to INFORMIX-Universal Server.
	Identifies information that is specific to INFORMIX-OnLine Dynamic Server.
	Identifies information that is specific to INFORMIX-OnLine Workgroup Server.
	Identifies information that is specific to INFORMIX-SE.
	Identifies information that is specific to a user-defined routine (UDR) that is written in Informix Stored Procedure Language.
	Identifies information that is specific to the UNIX operating system.
	Identifies information that is specific to the Windows NT environment.
	Identifies information that is specific to the Windows 95 environment.
	Identifies information that is specific to both Windows NT and Windows 95 environments.
	Identifies information that is specific to INFORMIX-OnLine XPS.
	Identifies functionality that conforms to X/Open specifications for dynamic SQL. This functionality is available when you compile your SQL API with the -xopen flag.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature-, product-, or platform-specific information.

Command-Line Conventions

This section defines and illustrates the format of commands that are available in Informix products. These commands have their own conventions, which might include alternative forms of a command, required and optional parts of the command, and so forth.

INFORMIX-ESQL/C supports a variety of command-line options. Each valid command-line option is illustrated in a diagram in [Chapter 1, “Programming with INFORMIX-ESQL/C.”](#)

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

You might encounter one or more of the following elements on a command-line path.

Element	Description
command	This required element is usually the product name or other short word that invokes the product or calls the compiler or preprocessor script for a compiled Informix product. It might appear alone or precede one or more options. You must spell a command exactly as shown and use lowercase letters.
<i>variable</i>	A word in italics represents a value that you must supply, such as a database, file, or program name. A table following the diagram explains the value.
-flag	A flag is usually an abbreviation for a function, menu, or option name or for a compiler or preprocessor argument. You must enter a flag exactly as shown, including the preceding hyphen.

(1 of 2)

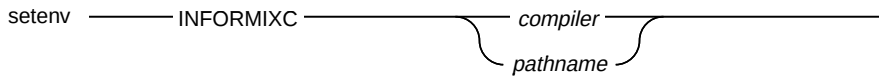
Element	Description
.ext	A filename extension, such as .sql or .cob , might follow a variable that represents a filename. Type this extension exactly as shown, immediately after the name of the file. The extension might be optional in certain products.
(. , ; + * - /)	Punctuation and mathematical notations are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
Privileges p. 5-17 Privileges	A reference in a box represents a subdiagram. Imagine that the subdiagram is spliced into the main diagram at this point. When a page number is not specified, the subdiagram appears on the same page.
— ALL —	A shaded option is the default action.
→ → →	Syntax within a pair of arrows indicates a subdiagram.
—	The vertical line terminates the command.
-f OFF ON	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
, variable	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items.
, 3 size	A gate ($\sqrt{3}$) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify <i>size</i> no more than three times within this statement segment.

(2 of 2)

How to Read a Command-Line Diagram

Figure 1 shows a command-line diagram that uses some of the elements that are listed in the previous table.

Figure 1
Example of a Command-Line Diagram



To construct a command correctly, start at the top left with the command. Then follow the diagram to the right, including the elements that you want. The elements in the diagram are case sensitive.

Figure 1 diagrams the following steps:

1. Type the word `setenv`.
2. Type the word `INFORMIXC`.
3. Supply either a compiler name or pathname.
After you choose *compiler* or *pathname*, you come to the terminator.
Your command is complete.
4. Press RETURN to execute the command.

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes, release notes, and machine notes

On-Line Manuals

An Answers OnLine CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine.

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com.

Please provide the following information:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. The **finderr** utility displays these error messages on the screen. See the Introduction to the *Informix Error Messages* manual for a detailed description of these error messages.

UNIX

To read the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). For a detailed description of these scripts, see the Introduction to the *Informix Error Messages* manual. ♦

WIN NT/95

Informix Find Error is a graphical tool. This utility has been created with Microsoft help facilities. For more information, see the Introduction to the *Informix Error Messages* manual. ♦

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following on-line files supplement the information in this manual. For UNIX, these files are located in the **\$INFORMIXDIR/release/en_us/0333** directory. For Windows, these files are located in the **\$INFORMIXDIR\release\en_us\04e4** directory.

On-Line File	Purpose
ESQLCDOC_9.1	The documentation notes file describes features that are not covered in this manual or that have been modified since publication. For Windows, click the Documentation Notes icon.
CLIENTS_9.1	The CLIENTS_2.0 file lists the release-notes files for the 2.0 Client SDK. These release-notes files describe feature differences from earlier versions of Informix products and how these differences might affect current products. These files also contain information about any known problems and their workarounds. In addition, the file for ESQL/C contains a section on limits that describes the maximum values for INFORMIX-ESQL/C. For Windows, click the Release Notes icon.
ESQLC_9.1	The machine notes file describes any special actions that are required to configure and use Informix products on your computer. ♦

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

UNIX

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. 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, on Informix database servers. In addition, many features of Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send email, our address is:

`doc@informix.com`

Or send a facsimile to the Informix Technical Publications Department at:

650-926-6571

We appreciate your feedback.

What Is INFORMIX-ESQL/C?

Section I



Programming with INFORMIX-ESQL/C

What Is INFORMIX-ESQL/C?	1-5
ESQL/C Executable Files for Windows	1-6
ESQL/C Library Functions.	1-7
Creating an ESQL/C Program	1-8
Compiling an ESQL/C Program	1-9
ESQL/C Preprocessor	1-9
C Preprocessor and Compiler	1-10
Default Compilation Order.	1-11
Running the C Preprocessor First	1-12
Using ESQL/C Preprocessor Directives.	1-12
The include Directive.	1-13
The define and undef Directives	1-14
The ifdef, ifndef, elif, else, and endif Directives	1-16
Embedding SQL Statements.	1-17
Handling Case Sensitivity in Embedded SQL Statements	1-18
Using Quotes and Escape Characters	1-19
Adding Comments	1-21
Specifying Host Variables	1-21
Using ESQL/C Header Files	1-22
Declaring Function Prototypes	1-25
Including Header Files	1-26
Declaring and Using Host Variables in SQL Statements	1-27
Declaring a Host Variable	1-27
Host-Variable Names	1-28
Host-Variable Data Types	1-28
Initial Host-Variable Values	1-29
Scope of Host Variables	1-29
Sample Host-Variable Declarations	1-31

Assigning a Value to a Host Variable	1-32
SQL Identifiers.	1-32
Null Values in Host Variables.	1-35
Using Host Variables in Data Structures	1-36
Arrays of Host Variables	1-36
C Structures as Host Variables	1-37
C typedef Expressions as Host Variables	1-38
Pointers as Host Variables	1-39
Function Parameters as Host Variables	1-40
Using Indicator Variables	1-42
Values in Indicator Variables	1-42
Declaring Indicator Variables.	1-44
Associating an Indicator Variable with a Host Variable	1-45
An Example Using Indicator Variables	1-46
Using the esql Command	1-47
Requirements for Using esql	1-48
Syntax of the esql Command	1-48
The esql Command for UNIX.	1-49
The esql Command for Windows Environments	1-52
Options That Affect Preprocessing	1-58
Checking the Version Number	1-58
Associating Options with Files	1-58
Preprocessing Without Compiling or Linking	1-58
Generating Thread-Safe Code	1-59
Checking for ANSI-Standard SQL Syntax	1-59
Defining and Undefined Definitions While Preprocessing	1-60
Checking for Missing Indicator Variables	1-62
Naming the Location of Include Files	1-62
Numbering Lines	1-64
Setting the Scope of Cursor Names and Statement IDs	1-64
Redirecting Errors and Warnings	1-64
Suppressing Warnings	1-65
Using X/Open Standards	1-65
Running the C Preprocessor Before the ESQL/C Preprocessor	1-66
Options for Running the C Preprocessor First	1-67
CPFIRST Environment Variable	1-67
The Order of Compilation When the C Preprocessor Runs First.	1-68

Defining Host Variables Based on C #defines and typedefs	1-69
Allowing All Valid C Declaration Syntax Inside the EXEC SQL Declare Section.	1-71
Excluding Statements Inside C Header Files	1-71
SQL Keyword Protection.	1-72
SQL Keyword Protection and the \$ Symbol	1-72
Preprocessor Options Specific to Windows Environments.	1-73
Line Wrapping	1-73
Changing Error and Warning Displays	1-74
Setting Tab Stops	1-74
Compiling and Linking Options of the esql Command	1-75
Syntax of the Compiling/Linking Options for UNIX	1-75
Naming the Executable File on UNIX	1-76
Passing Options to the C Compiler	1-76
Specifying a Particular C Compiler	1-77
Syntax of the Compiling/Linking Options for Windows Environments	1-77
Naming the Executable File in Windows Environments	1-79
Passing Options to the C Compiler in Windows Environments	1-80
Specifying a Particular C Compiler in Windows Environments	1-80
Compiling Without Linking.	1-80
Special Compile Options for Windows Environments	1-81
Using a Project File.	1-81
Creating a Response File	1-82
Implicit Options Invoked by the esql Preprocessor in Windows Environments	1-82
General Options That Affect Linking	1-86
Linking Other C Source and Object Files	1-86
Specifying Versions of Informix General Libraries	1-87
Special Linking Options for Windows Environments	1-87
Passing Arguments to the Linker	1-87
Passing Arguments to the Resource Compiler	1-88
ESQL/C Dynamic Link Libraries	1-88
Using the Same Runtime Routines for Version Independence	1-89

Windows Operating System Processor Options Available to the esql Command.	1-90
Syntax of the Options for Windows Environments	1-90
Accessing the ESQL Client-Interface DLL in Windows Environments	1-94
Accessing the Import Library.	1-95
Locating a DLL	1-95
Building an Application DLL.	1-96
Setting and Retrieving Environment Variables in Windows Environments	1-97
ifx_putenv()	1-100
ifx_getenv()	1-102
InetLogin Structure	1-103
Fields of the InetLogin Structure	1-104
InetLogin Field Values	1-109
Precedence of Configuration Values	1-111
Using Host Variables in Windows Environments	1-112
Declaring Variables with Non-ANSI Storage-Class Modifiers	1-112
Using Global ESQL / C Variables	1-115
A Sample INFORMIX-ESQL / C Program	1-116
Compiling the demo1 Program	1-116
Guide to demo1.ec File	1-116

T

his chapter covers the following topics:

- What is INFORMIX-ESQL/C?
- Using ESQL/C preprocessor directives
- Embedding SQL statements in C programs
- Declaring and using host variables in SQL statements
- Preprocessing and compiling an ESQL/C program

The last section of this chapter, “[A Sample INFORMIX-ESQL/C Program](#)” on [page 1-116](#), presents an annotated sample program that is called **demo1**. The **demo1** sample program illustrates the basic programming concepts that this chapter introduces.

What Is INFORMIX-ESQL/C?

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 ESQL/C preprocessor, which the **esql** command calls, converts each SQL statement and all Informix-specific code to C.

ESQL/C consists of the following software components:

- The ESQL/C libraries of C functions, which provide access to the database server
- The ESQL/C header files, which provide definitions for the data structures, constants, and macros useful to the ESQL/C program
- The **esql** command, which manages the processing of the source code to convert a C file that contains SQL statements into an object file

GLS

WIN NT/95

WIN NT/95

WIN NT

- The **finderr** utility on the UNIX system and the **INFORMIX FIND ERROR** Windows-based utility that enable you to obtain information about Informix-specific error messages.
- The Informix GLS *locale* and *code-set-conversion files*, which provide locale-specific information. For more information on these files, see the [Informix Guide to GLS Functionality](#). ♦
- The ESQL client-interface *dynamic link libraries* (DLLs), which enable an ESQL/C application to run in a Windows environment. ♦

ESQL/C Executable Files for Windows

For Windows environments, the 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 [Informix Client Products Installation Guide](#) for Windows.
- The **ILOGIN** utility is a demonstration program that displays a dialog box with fields for the connection parameters, for testing a connection to the database server (uses the **stores7** database).
- The **REGCOPY** utility is a Windows-based utility that makes an ESQL/C program into a Windows NT service. For more information, see the [Informix Client Products Installation Guide](#) for Windows. ♦
- The **ESQLMF.EXE** multibyte filter changes escape characters in multibyte strings into hexadecimal literals.

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

ESQL/C Library Functions

The ESQL/C library contains a set of C functions that you can use in your application. 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, respectively.
- 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 cancelling 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. ♦

IUS

To access a database server through an ESQL/C application

1. Write embedded SQL statements and calls to ESQL/C library functions into your C code to create an ESQL/C program.
2. Convert the ESQL/C program to a C executable file with the **esql** command.
3. Repeat step 2 as needed to remove programming errors.

The following sections provide detailed instructions for steps 1 and 2.

Creating an ESQL/C Program

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

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

For information on C preprocessor directives and C language statements, refer to a C programming text. This chapter provides an overview of the ESQL/C preprocessor directives ([page 1-12](#)), embedded SQL statements ([page 1-17](#)), and host variables ([page 1-27](#)).

Your ESQL/C source code filename can have either of the following forms:

- `esqlc_source.ec`
- `esqlc_source.ecp`

The particular suffix that your 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 on the **.ecp** suffix and the non-default order of compilation, see [“Running the C Preprocessor Before the ESQL/C Preprocessor” on page 1-66](#). ♦

Compiling an ESQL/C Program

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

ESQL/C Preprocessor

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

- In Stage 1, the 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 2, the ESQL/C preprocessor processes any conditional compilation directives (**ifdef**, **ifndef**, **else**, **elif**, **endif**) and translates embedded SQL statements to ESQL/C function calls and special data structures.

Stages 1 and 2 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 1-58](#).

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

- the filename of the ESQL/C source file has the **.ec** or the **.ecp** extension.
 - the **INFORMIXDIR** and **PATH** environment variables are set correctly.
- For information on 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 1-47](#).



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

C Preprocessor and Compiler

The **esql** command does not itself compile and link the ESQL/C program. Instead, 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. The **esql** command calls the C compiler to initiate this process, but its main task is to translate ESQL/C code to C code.

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

If you choose the option of running the C preprocessor on the ESQL/C source file before the ESQL/C preprocessor, you can use the C language **#define** and **typedef** directives to define ESQL/C host variables. For more information, see [“Running the C Preprocessor Before the ESQL/C Preprocessor” on page 1-66](#).

The C compiler takes the following actions:

- Compiles the C language statements to object code
- Links to ESQL/C libraries as well as any other files or libraries you specify
- Creates an executable file

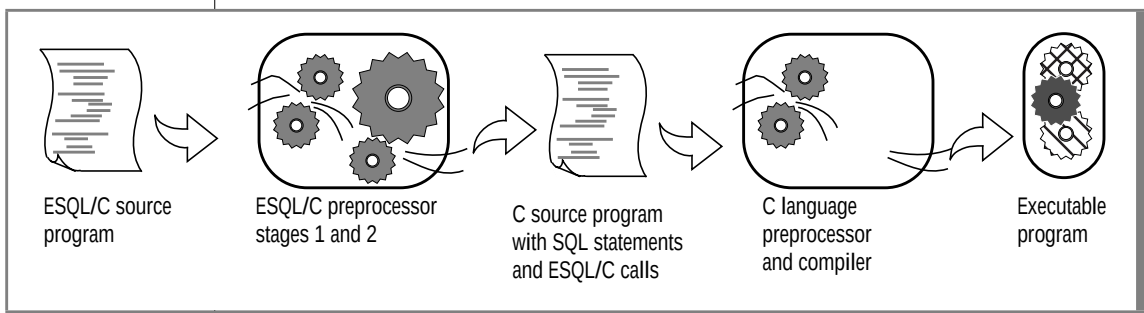
For information about command-line options that affect the compilation and linking steps, see [“Specifying a Particular C Compiler” on page 1-77](#) and [“General Options That Affect Linking” on page 1-86](#).

Default Compilation Order

Once you have created an ESQL/C program file, you run the **esql** command on that file. By default, the ESQL/C preprocessor runs first and translates the embedded SQL statements in the program into ESQL/C function calls that communicate with the database server. The 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 ESQL/C program as you would any C program. When the program runs, it calls the ESQL/C library procedures; they set up communications with the database server to carry out the SQL operations.

Figure 1-1 illustrates the process by which an ESQL/C program becomes an executable program.

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





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

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

Running the C Preprocessor First

With this release, you can either let the compilation of your ESQL/C source file follow the default order or let the C preprocessor run on the source file first, and then pass that file to the ESQL/C preprocessor. This feature is useful in case you want your ESQL/C program to access C preprocessor directives. For more information on this option, see [“Running the C Preprocessor Before the ESQL/C Preprocessor” on page 1-66](#).

Using ESQL/C Preprocessor Directives

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

- The **include** directive expands 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 ESQL/C preprocessor directives:

- The EXEC SQL keywords:
`EXEC SQL preprocessor_directive;`
- The dollar sign (\$) notation:
`$preprocessor_directive;`



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

Important: The EXEC SQL keywords conform to ANSI standards.

The ESQL/C preprocessor works in two stages. In Stage 1, it acts as a preprocessor for the ESQL/C code. In Stage 2, it converts all of the embedded SQL code to C code. The ESQL/C preprocessor handles the preprocessor directives at different stages, as follows:

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

The include Directive

The **include** directive allows you to specify a file to include within your ESQL/C program. When it reaches an **include** directive, the ESQL/C preprocessor places the contents of the specified file into the ESQL/C source file. Stage 1 of the 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*;

In either of these formats, replace *filename* with the name of the file to be included in your program. You can specify *filename* with or without quotation marks around the filename. However, if you use a full pathname, you *must* enclose the pathname in quotation marks. The following example shows how to use full pathnames on a Windows system.

```
EXEC SQL include decimal.h;

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

If you omit the quotes around the filename, ESQL/C changes the filename to lowercase characters. If you omit the pathname, the ESQL/C preprocessor checks the preprocessor search path for the file. For more information on this search path, see [“Naming the Location of Include Files” on page 1-62](#).

You can use **include** for the following types of files:

- An ESQL/C header file

You do not have to use the **.h** file extension for an 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 ESQL/C header files:

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

For a complete list of ESQL/C header files, see [page 1-22](#).

- Other user-defined files

You must specify the exact name of the file that you wish to include. The compiler does not assume the **.h** extension when you include a header file that is not an 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 "/work/esqlcstuff/constant_defs";
#include typedefs.h;
EXEC SQL include "/work/include/typedefs.h";
```

You *must* use the **include** directive if the file you specify contains SQL statements.



Tip: To include system header files, use the standard **#include** directive of the C preprocessor. The **#include** of C includes a file after ESQL/C preprocessing.

The define and undef Directives

The ESQL/C preprocessor allows you to create simple variables that are available *only* to the ESQL/C preprocessor. Informix calls these variables *definitions*. The 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 ESQL/C source file.

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

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

The 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;
```

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 definitions from the **esql** command line with the **-ED** and **-EU** options. For more information about these options, see [“Defining and Undefining Definitions While Preprocessing”](#) on page 1-60.

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

The 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**.
- 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 have previously defined the name USETRANSACTIONS with a **define** directive:

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

The following example uses the dollar sign (\$) format to perform the same task:

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

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

The ESQL/C preprocessor processes these directives rather than the C preprocessor (which processes **#ifdef** and **#ifndef**). The ESQL/C preprocessor processes conditional compilation definitions in Stage 2 of the preprocessing.

Embedding SQL Statements

An 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 12, “Working with the Database Server.”](#)

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. 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 [Informix Guide to SQL: Syntax](#).

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

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

Handling Case Sensitivity in Embedded SQL Statements

Figure 1-2 describes how the ESQL/C preprocessor treats uppercase and lowercase letters.

Figure 1-2
Case Sensitivity in ESQL/C Files

ESQL/C Identifier	Case Sensitive	Example
Host Variable	Yes	ESQL/C treats the variables fname and Fname as distinct variables: <pre>EXEC SQL BEGIN DECLARE SECTION; char fname[16], lname[16]; char Fname[16]; EXEC SQL END DECLARE SECTION;</pre> This sample does <i>not</i> generate a warning from the preprocessor.
SQL Keyword	No	Both CONNECT statements are valid ways of establishing a connection to the stores7 database: <pre>EXEC SQL CONNECT TO 'stores7'; or EXEC SQL connect to 'stores7';</pre> In examples given in this manual, SQL keywords appear as lowercase characters.
Statement Identifiers Cursor Names	No	The following example shows the creation of statement IDs and cursor names: <pre>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;</pre> This code produces errors because the statement IDs st and ST are duplicates, as are the cursor names curname and CURNAME.

Using Quotes and Escape Characters

An *escape character* indicates to the 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 illegal. For example, the following attempt to find a single quote does not conform to ANSI standards:

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

In nonembedded tools such as DB-Access, you can escape a quote 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 quote, you can enclose this double quote with single quotes, as follows:

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

However, ESQL/C uses quotation marks differently than it uses nonembedded tools. Figure 1-3 shows a SELECT statement with a WHERE clause that contains a double quote enclosed with single quotes.

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

Figure 1-3
A SELECT
Statement with an
Invalid WHERE
Clause

For the WHERE clause in [Figure 1-3 on page 1-19](#), the ESQL/C preprocessor does not process a double quote; it just passes it on to the C compiler. When the C compiler receives the string ' "' (double quote enclosed with single quotes), it interprets the first single quote as the start of a string and the double quote as the end of a string. The compiler cannot match the single quote that remains and therefore generates an error.

To resolve this misinterpretation, you can escape the double quote with the C escape character, the backslash (\). The correct syntax for the query in [Figure 1-3](#) is as follows:

```
EXEC SQL select coll from tabl where coll = '\'';
```

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. To search for the string "\"" (where the double quotes are *part of* the string), enter *five* backslashes in the following query:

```
EXEC SQL select coll from tabl where coll = '\"\\\\\\\"';
```

Figure 1-4 shows the string after it has passed through each of the processing steps.

Figure 1-4
Escaped Query String As It Is Processed

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

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

Adding Comments

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

- You can use a double dash (--) comment indicator on any ESQL/C line (one that begins with either EXEC SQL or \$ and terminates with a semicolon). The comment continues to the end of the line.

For example, the following line of code includes a DATABASE statement that opens the **stores7** database and a comment to that effect:

```
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 ESQL/C line, as shown in the following example:

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

Specifying Host Variables

Host variables are normal C variables that you use in embedded SQL statements. When you use a host variable in an SQL statement, you must precede its name with a symbol to distinguish it from regular C variables. 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 on the syntax of indicator variables, see [“Using Indicator Variables” on page 1-42](#).

Outside an SQL statement, treat a host variable as you would a regular C variable. For more information on how to declare and use host variables, see [“Declaring and Using Host Variables in SQL Statements” on page 1-27](#).

Using ESQL/C Header Files

The Informix installation script stores the header files in the **\$INFORMIXDIR/ incl/esql** directory on a UNIX system and in the **%INFORMIXDIR%\ incl\esql** directory in a Windows environment.

Figure 1-5 shows the header files included with the ESQL/C product.

Figure 1-5
ESQL/C Header Files

Header File	Contains	Additional Information
datetime.h	Structure definition for DATETIME and INTERVAL data types.	Chapter 6
decimal.h	Structure definition for DECIMAL and MONEY data types.	Chapter 4
gls.h	Function prototypes and data structures for the gls functionality.	Informix Guide to GLS Functionality

(1 of 3)

Header File	Contains	Additional Information
locator.h	Structure definition for BYTE and TEXT data types.	Chapter 7
sqlca.h	Structure definition that ESQL/C uses to store error-status codes. This structure allows you to check for the success or failure of SQL statements. The esql preprocessor automatically includes this file when it preprocesses your program.	Chapter 10
sqlda.h	Structure definition for value pointers and descriptions of dynamically defined variables.	Chapter 12
sqlhdr.h	The definitions of cursor-related structures. This file includes the sqlda.h header file. <i>The preprocessor automatically includes this file when it preprocesses your program.</i>	“Including Header Files” on page 1-26
sqlapi.h	Function prototypes for ESQL/C library APIs.	
sqlproto.h	The function prototypes of all ESQL/C library functions for use with source that is <i>not</i> fully ANSI C compliant.	“Declaring Function Prototypes” on page 1-25
sqlstype.h	The definitions of strings that correspond to SQL statements. ESQL/C uses these strings when your program contains the DESCRIBE statement.	Chapter 12
sqltypes.h	The integer constants that correspond to ESQL/C language and SQL data types. ESQL/C uses these constants when your program contains a DESCRIBE statement.	Chapter 12
sqlxtype.h	The integer constants that correspond to C language and SQL data types that ESQL/C uses when you are in X/Open mode. ESQL/C uses these constants when your program contains a DESCRIBE statement.	Chapter 12

(2 of 3)

Header File	Contains	Additional Information
value.h	The value structures that the Informix product NewEra uses.	NewEra document- tation
varchar.h	The macros that you can use with the VARCHAR data type.	Chapter 2
xa.h	Function prototypes and data structures for TP/XA applications.	TP/XA Programmer's Manual

(3 of 3)

Figure 1-6 shows the ESQL/C header files specific to INFORMIX-Universal Server.

Figure 1-6
ESQL/C Header Files for INFORMIX-Universal Server

Header File	Contents	Additional Information
collect.h	Definitions of data structures for complex types in ESQL/C.	Chapter 9
ifxgls.h	Function prototypes for the GLS Application Programming Interface	
int8.h	The definition of the structure that stores the INT8 data type.	Chapter 5



The **sqlxtype.h** header file is not available in the Windows environments.

Figure 1-7 shows the ESQL/C header files specific to the Windows environment.

Figure 1-7
ESQL/C Header Files for Windows Environments

Header File	Contents	Additional Information
ifxcexp.h	The ifxcexp.c file 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 1-89
login.h	The definition of the InetLogin and HostInfoStruct structures, which enable you to customize configuration information for the application. <i>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.</i>	Chapter 3



Declaring Function Prototypes

ESQL/C provides the **sqlproto.h** header file to declare function prototypes for all ESQL/C library functions. These function prototypes are required in an 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 ESQL/C functions prevents an ANSI C compiler from generating warnings.

Warning: Although you can use an ANSI C compiler, the 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 ESQL/C statements, you can include this file in either of the following ways:

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

```
EXEC SQL include sqlproto;
```

- With the C **#include** preprocessor directive:

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

Including Header Files

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

- The **sqlca.h** file, which allows your program to check the success or failure of your ESQL/C statements with the SQLSTATE or SQLCODE variable
- The **sqlhdr.h** file provides cursor-related structures for your ESQL/C program. This header file automatically includes the **sqlda.h** header file. ♦

Warning: *Although you can now use an ANSI C compiler, the 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 ESQL/C program, you must use the **include** preprocessor directive. However, you only need to include an 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. Here are some additional examples:

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

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

For information on the **include** directive, see [“The include Directive” on page 1-13](#).

WIN NT/95



Declaring and Using Host Variables in SQL Statements

In an ESQL/C application that uses embedded SQL statements, the SQL statements can refer to the contents of *host variables*. A *host variable* is a C program variable that you use in an embedded SQL statement. Informix uses the term *host variable* because you declare the variable in the host program, the C program. The SQL statements are *guests* in the host program. They use host variables to transfer information between the C host program and the database.

You can use host variables in ESQL/C expressions in the same way that you use literal values. To use a host variable in an SQL statement, you must follow these steps in your ESQL/C program:

1. Declare the host variable within 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 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 ESQL/C program as ordinary 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 1-31](#).

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. 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 1-18](#).)

GLS



You can use non-ASCII (non-English) characters in ESQL/C host-variable names if your client locale supports these non-ASCII characters. For more information on how the client locale affects host-variable names, see the [Informix Guide to GLS Functionality](#). ♦

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 standard 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, refer to [Chapter 2, “INFORMIX-ESQL / C Data Types.”](#) In addition, the [Informix Guide to SQL: Reference](#) contains information on 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 1-36](#).

Initial Host-Variable Values

ESQL/C allows you to declare host variables with normal C initializer expressions. However, initializers with character strings cannot contain embedded semicolons or ESQL/C keywords. 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};
EXEC SQL END DECLARE SECTION;
```

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

Scope of Host Variables

The *scope of reference*, or simply the *scope*, of a host variable is that portion of the program in which the host variable can be accessed. The placement of the 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 braces ({ }).

Declaring a Host Variable

For example, the host variable **blk_int** in Figure 1-8 is valid only in the block of code between the 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 1-8
*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 that govern regular C variables govern the scope of ESQL/C host variables as well:

- A host variable is an automatic variable unless you explicitly define it as an **external** or **static** variable.
- 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

Figure 1-9 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 1-9
*Declaring Host
Variables with the
EXEC SQL Syntax*

Figure 1-10 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 1-10
*Declaring Host
Variables with the
Dollar Sign (\$) Notation*

For information on how to use a host variable in an SQL statement, see [page 1-21](#).

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. Null values are special values that indicate an *unknown* value.

Host variables can appear within an SQL statement as syntax allows. (For information about the syntax of SQL statements, see the [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 1-21.

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 database 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.

The identifier name can consist of letters, numbers, and underscores (_). The first character must be a letter. In most contexts, an identifier can have from 1 to 18 characters. (For more information on naming conventions for SQL identifiers, see the Identifier segment in the [Informix Guide to SQL: Syntax](#).)

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

Important: When you use double quotes (" ") to delimit identifiers, you conform to ANSI standards; single quotes (' ') 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 [Informix Guide to SQL: Syntax](#).)
- an identifier that contains nonalphabetic characters.

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

- At compile time, the 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 quotes (" ") to delimit strings. If you want to indicate a quoted string, enclose the text with single quotes (' ').

Delimited identifiers are case sensitive. All database object names that you place within quotes maintain their case. Keep in mind that ESQL/C restricts identifier names to a maximum of 18 characters.

Figure 1-11 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;

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

Figure 1-11
*Using Delimited
Identifiers for a
Cursor Name*

In [Figure 1-11 on page 1-33](#), you could 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 single quotes for the preprocessor to treat it as a string. If you enclose the statement in double quotes, the preprocessor treats it as an identifier.

To declare a cursor name that contains a double quote, 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 quote in the cursor name:

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

strcpy("\\"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 quote.

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

- The cursor name must contain *two* double quotes.

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

Figure 1-12 shows the string that contains the cursor name as it is processed.

Figure 1-12
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 like " " and avoids conflict with the syntax for INFORMIX-SE database names.
- You *cannot* use a delimited identifier for a storage identifier, like 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, 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, respectively.

For a description of these library functions, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#)

- Indicator variables are special 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 1-42.](#)

Using Host Variables in Data Structures

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

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

Arrays of Host Variables

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 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 on specific statements, consult the [Informix Guide to SQL: Syntax](#).

C Structures as Host Variables

ESQL/C supports the declaration of a C structure (**struct**) as a host variable. You can use the components of the structure within 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;
        charfname[32];
        charlname[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 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 Expressions as Host Variables

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

Pointers as Host Variables

You can use a pointer as a host variable as long as your program uses the pointer to input data to an SQL statement. For example, Figure 1-13 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 1-13
*Declaring a
Character Pointer to
Input Data*

Figure 1-14 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 1-14
*Declaring an Integer
Pointer to Input
Data*

If you use a pointer that is a host variable 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, Figure 1-15 shows a code fragment with Kernighan & 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 1-15
*Using EXEC SQL to
Declare Host
Variables as
Parameters to a
Kernighan &
Ritchie-Style
Function
Declaration*

You can also declare parameter host variables with the dollar sign (\$) notation. For example, Figure 1-16 shows the function header in Figure 1-15, with the dollar sign (\$) notation.

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

Figure 1-16
*Using \$ 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. It is also possible to 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 1-17 on page 1-41](#) 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.

```

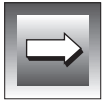
int * f(
EXEC SQL BEGIN DECLARE SECTION;
    PARAMETER char s[20],
    PARAMETER int id,
    PARAMETER int (*f) (double)
EXEC SQL END DECLARE SECTION;
)
{
    select fname into :s from customer
        where customer_num = :id;

    :
    :
}

```

Figure 1-17
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 of the EXEC SQL declare sections to use common header files for C and ESQL/C source files. For more information on how to use common header files between C and ESQL/C source files, see [“Running the C Preprocessor Before the ESQL/C Preprocessor” on page 1-66](#).



Important: If you want to define ESQL/C host variables which 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 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 Indicator Variables

When an SQL statement returns a value, it puts this value in the associated host variable. In some situations, you can use an *indicator variable* to provide additional information about the value that is returned. An indicator variable is an ESQL/C variable that is associated with a host variable. When you pair a host variable and an indicator variable in this way, Informix calls the host variable the *main variable*. When ESQL/C returns a value into the main variable, it can also set the indicator variable so that your program can determine more about the value.

The indicator variable provides additional information in the following situations:

- If you associate a host variable with an aggregate function or with a database column that allows null values, and the host variable receives a null value, this value is often not distinct from other values.
- If the host variable is a character array and its value has been truncated, the size of the variable does not correctly indicate the size of the returned value.

This section describes how your program can use indicator variables to determine additional information about a returned value. Specifically, it contains sections about the following topics:

- How ESQL/C sets indicator variables to indicate the preceding two conditions
- How your program can declare an indicator variable
- How your program can associate an indicator variable with a host variable
- An example program that uses indicator variables

Values in Indicator Variables

ESQL/C can set an indicator variable in the following two cases:

- To indicate that the associated host variable contains a *null value*
- To indicate that a value in a host-variable character array has been *truncated*

Indicating Null Values

When an ESQL/C statement returns a null value to a host variable (through the INTO clause of a SELECT or FETCH statement), the actual value in the main variable 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, ESQL/C sets the indicator variable to -1.
Your program can check for this value to determine whether a value is null.
- If you did *not* define an indicator variable, the behavior of ESQL/C at runtime depends on how you compile the program:
 - If you compile the program with the **-icheck** preprocessor option, ESQL/C generates an error when the database server returns a null value and sets **sqlca.sqlcode** to a negative value. (See [Chapter 7, “Working with INFORMIX-Universal Server Smart Large Objects.”](#))
 - If you compile the program *without* the **-icheck** option, ESQL/C does not generate an error when the database server returns a null value. Therefore, your program has no way to recognize when the database server returns a null value.

If the return code is not null, ESQL/C sets the indicator variable to 0.

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.

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 in this case returns a zero (0).



Important: If you use the DATASKIP feature of the database server and you turn this feature on, an aggregate function might also return null if all the fragments that are on-line are empty or if all fragments are off-line. When DATASKIP is off, the values of the indicator variables (or the **risnull()** function) are not valid if the operation is not successful; therefore, check the SQLCODE value after the operation and before you check for null values.

The DATASKIP feature is not available for 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, ESQL/C:

- sets the SQLSTATE variable to "01004" to signal the occurrence of truncation. (For more information on SQLSTATE, see [“List of SQLSTATE Class Codes” on page 11-15.](#)) 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, ESQL/C still sets SQLSTATE and **sqlca.sqlwarn** to signal the truncation. However, your program has no way to determine how much data has been truncated.

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

Declaring Indicator Variables

You declare indicator variables in the same way as host variables. You put the declaration within an ESQL/C block, as shown in the following example:

```
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 1-27.](#)

Indicator variables can be of 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 that is called **name**. You could declare a **short** integer-indicator variable that is called **nameind**, as the following example shows:

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


GLS

You can use non-ASCII (non-English) characters in ESQL/C indicator-variable names if your client locale supports these non-ASCII characters. For more information on how the client locale affects host-variable names, see the [Informix Guide to GLS Functionality](#). ♦

Associating an Indicator Variable with a Host Variable

You associate a host variable and its indicator variable when you list them in an SQL statement. You can associate these variables in one of two ways:

- Prefix the indicator variable with a colon (:) and place the keyword **INDICATOR** between the main variable name and the indicator variable name as follows:

```
:hostvar INDICATOR :hostvarind
```

Important: The “**INDICATOR :hostvarind**” format conforms to ANSI standards.

- Place a separator symbol between the main variable name and the indicator variable name. The following separator symbols are valid:

- A colon (:)

```
$hostvar:hostvarind
```

- A dollar sign (\$)

```
$hostvar$hostvarind
```

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
```



An Example Using Indicator Variables

The code segments in Figure 1-18 and [Figure 1-19 on page 1-47](#) 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 last name of the company representative with **customer_num = 105** is shorter than 15 characters, 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.

Figure 1-18 shows an 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 1-18
*Using Indicator
Variables with
EXEC SQL and
the Colon (:) Symbol*

Figure 1-18 uses the INDICATOR keyword to associate the main and indicator variables. This method complies with the ANSI standard.

Figure 1-19 shows an 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 1-19
*Using Indicator
Variables with
the Dollar Sign (\$)
Notation*

Using the esql Command

To create an executable C program from an ESQL/C source file, use the **esql** command. The Informix installation script installs the **esql** command as part of the 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. Compiles the file that results with the C compiler to create an object file.
3. Spawns the resource compiler and links any resource files (.res) that you specify on the esql command line. ♦
4. Links the object file with the ESQL/C libraries and your own libraries.

You can use the **esql** command to perform all these steps. Alternatively, you can use the **esql** command to perform step 1, then compile and link the files with your C compiler directly. For more information about step 1, see [“Compiling an ESQL/C Program” on page 1-8](#).

WIN NT/95

Requirements for Using esql

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

- the filename of the ESQL/C source file has the **.ec** extension. You can also use the **.ecp** extension if you want the C preprocessor to run before the ESQL/C preprocessor.
- you set the environment variable **INFORMIXDIR** correctly and that 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 [Informix Guide to SQL: Reference](#) or the [Informix Client Products Installation Guide](#) for your operating system.

Syntax of the esql Command

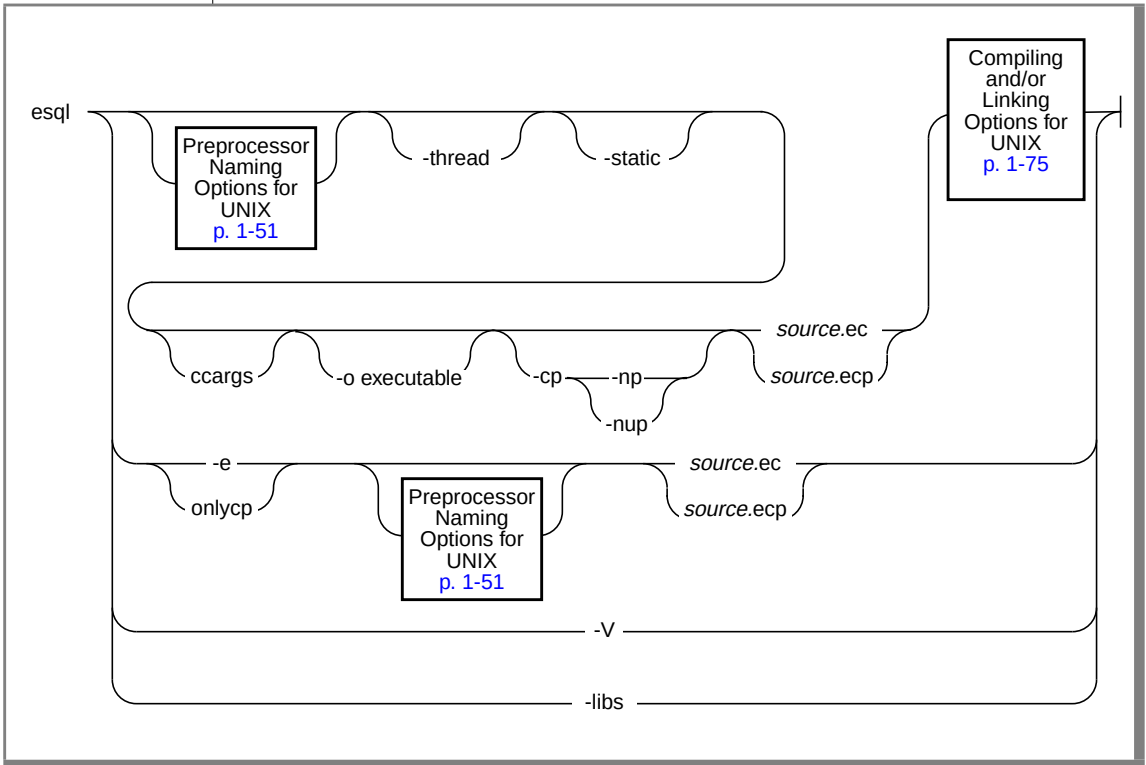
The following sections describe the syntax available for the **esql** command. This section organizes the command-line options by the processing phase that they affect:

- 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.

UNIX

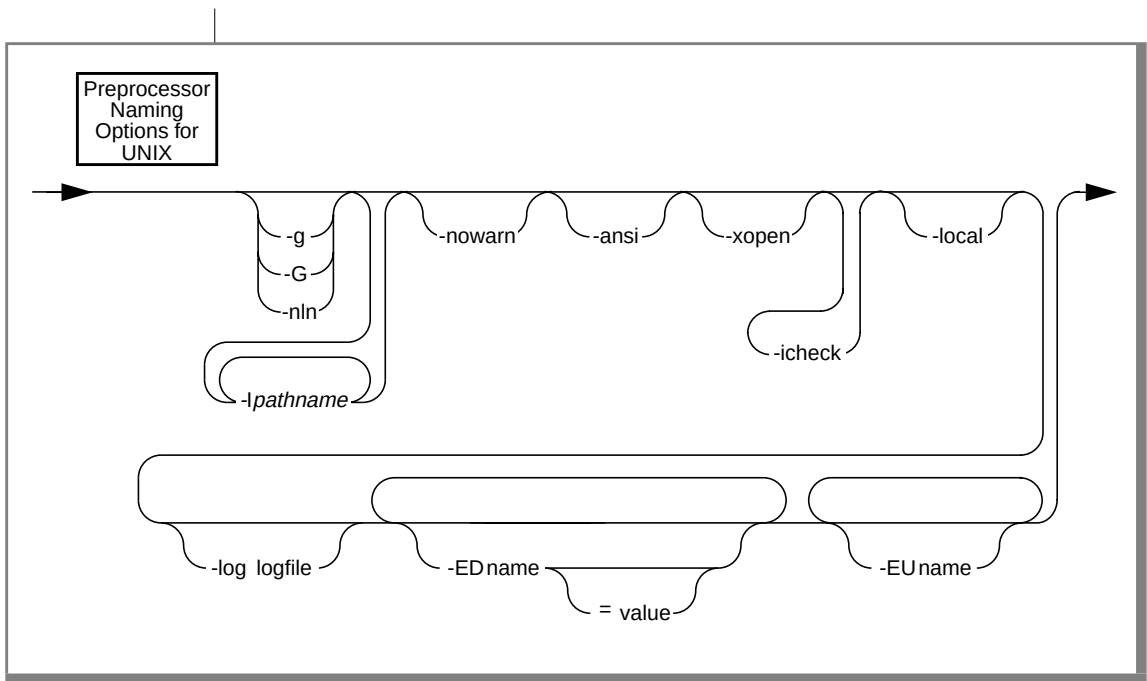
The *esql* Command for UNIX

The following diagram gives the **esql** command options for the UNIX operating system. For more information on the **-cp**, **-np**, **-nup**, and **-cponly** options, see [“Running the C Preprocessor Before the ESQL / C Preprocessor”](#) on page 1-66.



- cp** runs the C preprocessor before the ESQL / C preprocessor. By default this option gives protection to SQL keywords from interpretation by the C preprocessor.
- e** preprocesses only, without compiling or linking. This flag produces a C source file with a **.c** file extension.

-libs	displays the names of all of the libraries that the esql script uses when it links a program that is precompiled. This display is for informational purposes.
-np	No protect mode used with the -cp option. There is no SQL keyword protection against interpretation by the C preprocessor when the C preprocessor runs before the ESQL/C preprocessor.
-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 the <i>source.icp</i> file.
-o <i>executable name</i>	specifies the name of the executable file.
-static	links Informix static libraries instead of the default Informix shared libraries.
-thread	links Informix thread-safe libraries instead of the default Informix shared libraries.
-V	prints the version information for your ESQL/C product.
ccargs	represents options that the esql command passes to the C compiler.
onlycp	This mode is like the -cp mode in that it forces the C preprocessor to run before the ESQL/C preprocessor. However, the processing stops after the C preprocessor runs, leaving the result in a <i>.icp</i> file.
<i>source.ec</i>	The ESQL/C source file with the default suffix.
<i>source.ecp</i>	The ESQL/C source file with a special suffix in the name that forces the -cp option, causing the C preprocessor to run on the source file before the ESQL/C preprocessor does.



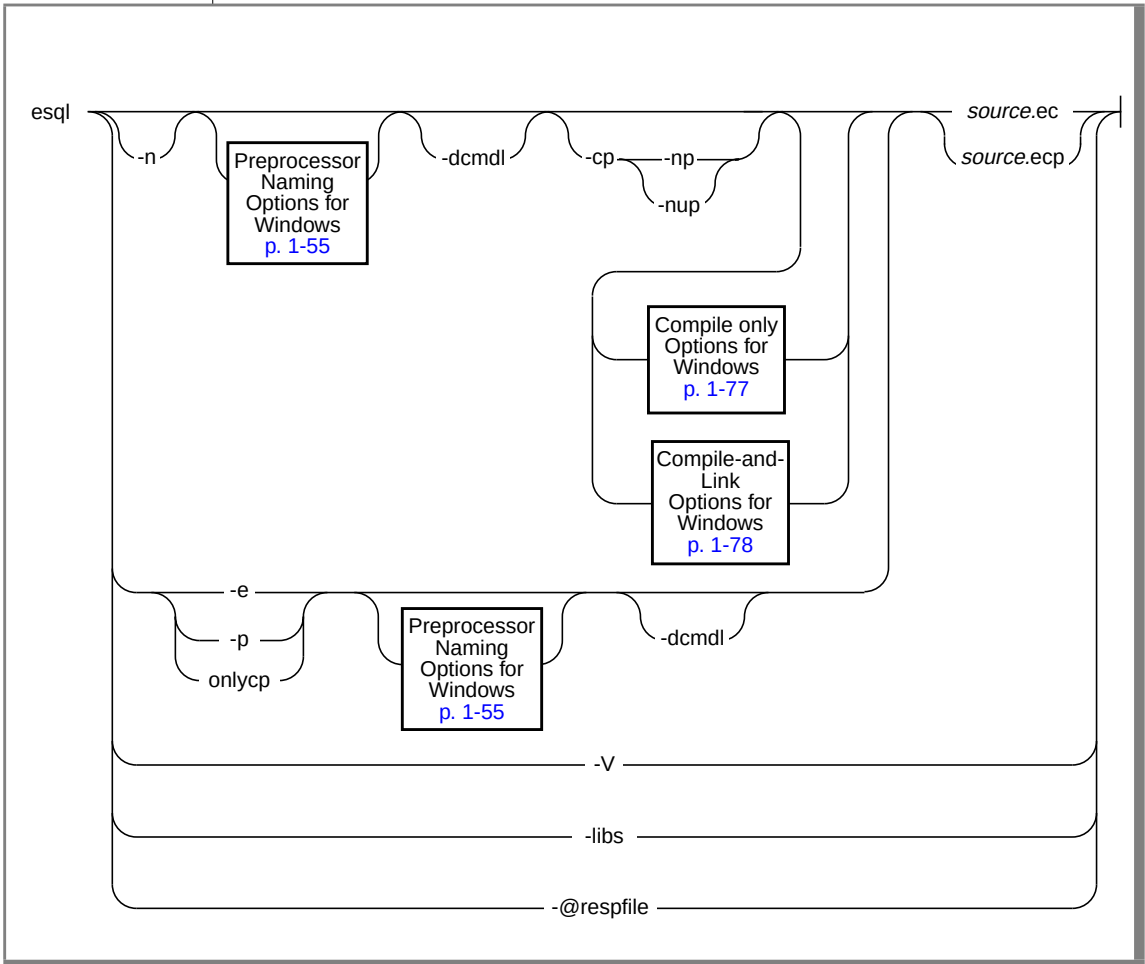
- ansi** checks for Informix extensions to ANSI-standard SQL syntax.
- EDname** defines a user-supplied name flag to the preprocessor that the **undef**, **ifdef**, and **ifndef** directives can use.
- EUname** undefines a specified preprocessor name flag.
- g** numbers every line (used by a debugger).
- G** suppresses line numbers (used by a debugger; same as **-nl**).
- icheck** generates the code to check if a null value is returned to a host variable that does not have an associated indicator variable. Generates an error if such a case exists.

-Ipathname	expands the search range for ESQL/C and C include files. The ESQL/C preprocessor only processes ESQL/C include files (those with the include directive). The esql command passes this option to the C compiler for processing of C include files (those with the #include preprocessor statement).
-local	specifies that the static cursor names and static statement ids that you declare in a file are local to that file. If you do not use the -local option, cursor names and statement IDs, by default, are global entities.
-log logfile	sends the error and warning messages to the specified file instead of to standard output.
-nl	suppresses line numbers (used by a debugger; same as -G).
-nowarn	suppresses warning messages from the preprocessor (has no effect on error messages). ♦

WIN NT/95

The esql Command for Windows Environments

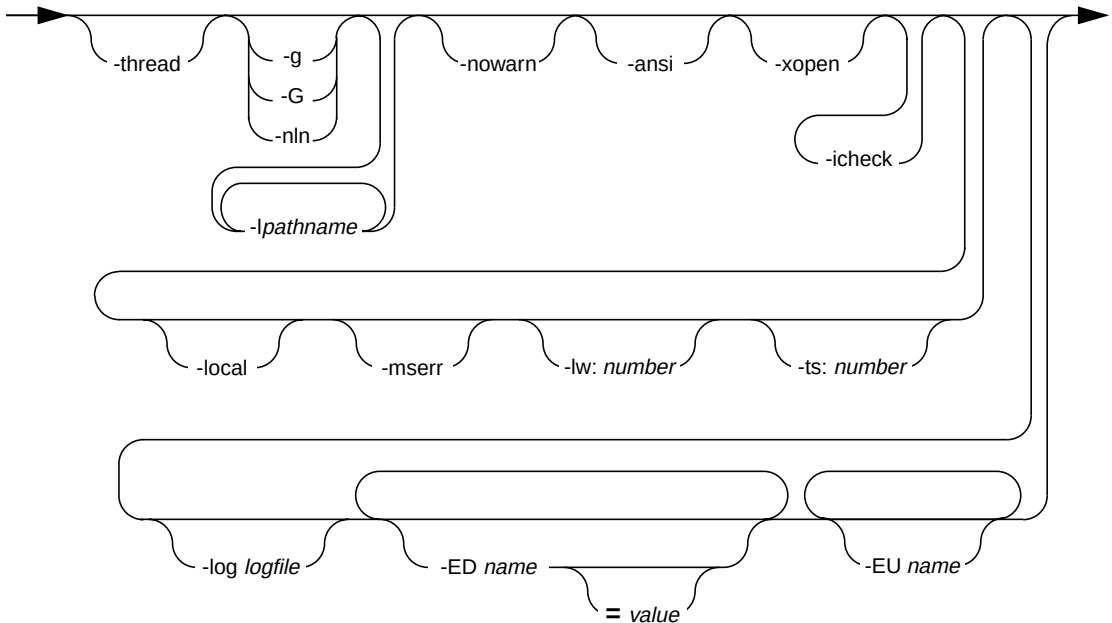
The following diagram shows the general syntax of the ESQL/C command processor, **esql**. For more information on the **-cp**, **-np**, **-nup**, and **onlycp** options, see “Running the C Preprocessor Before the ESQL/C Preprocessor” on page 1-66.



-cp	runs the C preprocessor before the ESQL/C preprocessor. By default this option gives protection to SQL keywords from interpretation by the C preprocessor.
-dcmdl	displays the name of all options passed to the C compiler, the linker, and the resource compiler.
-e	preprocesses only, no compiling or linking (same as -p). It produces a C source file with a .c file extension.
-f filename	specifies the name of the file that contains the names of your ESQL/C source files (.ec).
-libs	displays the names of all the libraries that the esql command uses when it links a precompiled program. This option is for informational purposes.
-n	specifies “no logo.” By default, a copyright message prints.
-np	noprotect mode used with the -cp option. There is no SQL keyword protection against interpretation by the C preprocessor when the C preprocessor runs before the ESQL/C preprocessor.
-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 the <i>source.icp</i> file.
-o exename	specifies the name of the executable file (.exe or .dll).
-p	preprocesses only, no compiling or linking (same as -e). It produces a C source file with a .c file extension.
-V	prints the command processor version information for your ESQL/C preprocessor.
onlycp	This mode is like the -cp mode in that it forces the C preprocessor to run first before the ESQL/C preprocessor. However, the processing stops after the C preprocessor runs, leaving the result in a .icp file.

<i>respfile</i>	specifies the name of the ESQL command-line response file.
<i>source.ec</i>	The ESQL/C source file with the default suffix.
<i>source.ecp</i>	The ESQL/C source file with a special suffix in the name that forces the -cp option, causing the C preprocessor to run on the source file before the ESQL/C preprocessor does.

Preprocessor
Naming
Options for
Windows



-ansi	checks for Informix extensions to ANSI-standard SQL syntax.
-EDname	defines a user-supplied name flag to the preprocessor that the undef , ifdef , and ifndef directives can use.
-EUname	undefines a specified preprocessor name flag.
-EDname=value	lets you assign an initial value to the user-defined <i>name</i> flag; for example: <pre>- EDMACNAME=62</pre> This definition is equivalent to the following define directive appearing at the top of your ESQL/C program: <pre>EXEC SQL define MACNAME 62;</pre>
-g	numbers every line by generating a #line compiler directive for each line in the source file (used by a debugger).
-G	no line numbers generated (used by a debugger; same as -nln).
-Ipathname	expands the search range for ESQL/C and C include files. The ESQL/C preprocessor only processes ESQL/C include files (those with the include directive). This option is passed on to the C compiler for processing of C include files (those with the #include preprocessor directive).
-icheck	generates the code to check for a null value returned to a host variable that does not have an indicator variable associated with it and generates an error if such a case exists.
-local	specifies that the static cursor names and static statement IDs that you declare in a file are local to that file. If you do not use the -local option, cursor names and statement IDs, by default, are global entities.

-log <i>logfile</i>	sends the error and warning messages to the specified file instead of to standard output. This option affects only preprocessor errors and warnings.
-lw: <i>number</i>	tells the preprocessor to perform line wrapping at the column position that <i>number</i> indicates.
-mserr	provides Microsoft-style messages and warnings.
-nln	no line numbers generated (used by a debugger; same as - G).
-nowarn	suppresses warning messages from the preprocessor (has no effect on error messages).
-thread	tells the ESQL/C preprocessor to create thread-safe code.
-ts: <i>number</i>	tells the preprocessor to define tab stops every <i>number</i> columns. By default, the preprocessor sets tab stops at every eighth column position.
-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 sqllda structure). ♦

Options That Affect Preprocessing

An ESQL/C program must be preprocessed before a C compiler can compile it. The 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 Informix version number and serial number for your ESQL/C product, as shown in the following example:

```
esql -V
```

Associating Options with Files

Most processor options that precede a filename affect the file. Suppose you issue the following command:

```
esql -G demo1.ec -ansi demo2.ec
```

The **-G** option affects the **demo1.ec** file, and both the **-ansi** and the **-G** options affect the **demo2.ec** file.

An exception is the **-o** option, which affects only the next item, as [“Naming the Executable File in Windows Environments” on page 1-79](#) describes.

Preprocessing Without Compiling or Linking

By default, the **esql** command causes the 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 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 resides 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 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 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 ESQL/C program, ESQL/C checks for ANSI compatibility. For information on how to set **DBANSIWARN**, see the [Informix Guide to SQL: Reference](#). For details about how to check for runtime warnings, see Chapter 10 of this manual. For details on how to set environment variables, see the [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 ESQL/C to check for ANSI compatibility at runtime.

UNIX

WIN NT/95

With the **-ansi** option, the 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 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 ESQL/C preprocessing.

To create a global definition, use one of the following forms of the **-ED** option:

- Use the **-EDname** syntax to define a Boolean symbol, as follows:

```
esql -EDENABLE_CODE define_ex.ec
```

- Use the **-EDname=value** syntax to define an integer constant, as follows:

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

The **-EDname** is equivalent to the **define** preprocessor directive (**\$define** or **EXEC SQL define**) with *name* at the top of your 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 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 on preprocessor definitions, see [“The define and undef Directives”](#) on page 1-14.

Figure 1-20 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 1-20
*ESQL/C Excerpt
That Uses ifdef,
ifndef, and endif*

For the code fragment shown in Figure 1-20, 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 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, 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 on indicator variables, see [“Using Indicator Variables” on page 1-42](#).

Naming the Location of Include Files

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

- The ESQL/C preprocessor (**esql**) processes *only* 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 ESQL/C compilation completes, the C include files are processed *after* the 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
```

UNIX

The *directory* can reside 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 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. ♦

WIN NT/95

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** ♦

UNIX

For more information on ESQL/C **include** files, see [“The include Directive” on page 1-13](#).

Numbering Lines

By default, the ESQL/C preprocessor gives line numbers to the embedded SQL lines in your program. You can change this default numbering with two command-line options:

- Use the **-g** option if you want to include line numbers for *every* line (C and embedded SQL).
- Use the **-G** or **-nl** options if you do not want *any* line numbers.

Setting the Scope of Cursor Names and Statement IDs

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



Tip: Informix provides the **-local** option primarily for backward compatibility with applications that were created in previous versions of 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 filename. 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 ESQL/C preprocessor generates. The warnings from the compile and link stages still go to the standard output.

Suppressing Warnings

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

Using X/Open Standards

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

If you use X/Open SQL in an 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 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 ESQL/C source file can follow either the default order, where the ESQL/C preprocessor runs first on the source file, or it can allow the C preprocessor to run on the source file before the ESQL/C preprocessor.

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

1. The ESQL/C preprocessor performs the following steps to create a **.c** file from the ESQL/C source file:
 - ❑ Incorporates 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 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 within C system header files or custom C header files to define ESQL/C host variables and constants, nor can you use them for conditional compilation of ESQL/C code. With the default order of compilation, C header files do not get included into the ESQL/C source code until after the ESQL/C preprocessor has run, making these definitions unavailable to the ESQL/C preprocessor.

Options for Running the C Preprocessor First

You can make the C preprocessor run on an ESQL/C source file first, thus expanding any C-dependent **typedefs** or **#defines** inside the ESQL/C source file before the ESQL/C preprocessor is run on that source file. This can be done 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 *source.icp* file.
- Create the ESQL/C source file with a **.ecp** extension. This triggers the **-cp** option by default.
- Set the **CPFIRST** environment variables to **TRUE** (uppercase only) as explained in the following section.

CPFIRST Environment Variable

The **CPFIRST** environment variable specifies whether the C preprocessor should run before the ESQL/C preprocessor on all ESQL/C source files. Set the environment variable to **TRUE** (uppercase only) to run the C preprocessor on all 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.

UNIX

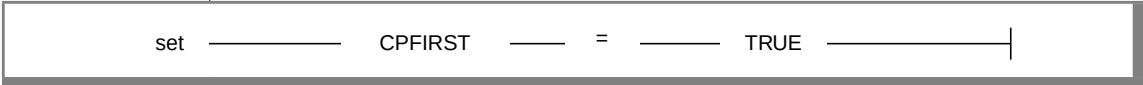
The following diagram shows how to set the **CPFIRST** environment variable in a UNIX C-shell environment.

```
setenv _____ CPFIRST _____ TRUE _____|
```

◆

WIN NT/95

The following diagram shows how to set the **CPFIRST** environment variable in a Windows command-line environment.



```
set CPFIRST = TRUE
```



The Order of Compilation When the C Preprocessor Runs First

When a user chooses to run the C preprocessor on an ESQL/C source file before the ESQL/C preprocessor, the file undergoes the following order of compilation. For details on the phases of SQL keyword protection and unprotection, see [“SQL Keyword Protection” on page 1-72](#).

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 ESQL/C preprocessor runs on the output of the C preprocessor, which no longer has any SQL keyword protection
5. The output of the ESQL/C preprocessor undergoes compilation and linking by the C compiler and linker to produce an executable file.

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 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 1-21, the same C header file is used by both ESQL/C and C source files.

```

/*header file i.h*/
#define LEN 10
typedef NAME char[LEN];
:
:
/*name.ec*/
main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
    #include "i.h"
    NAME name_hostvar;
EXEC SQL END DECLARE SECTION;
:
:
}

/*name.c*/
#include "i.h"
main ()
{...
NAME namevar;
:
:
}

```

Figure 1-21
*ESQL/C and C
 Excerpt That Use a
 Common C Header
 File*

In the example in Figure 1-22, the ESQL/C source file contains a host variable based on a type defined in the **time.h** system-header file.

```
/*time.ec*/
main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
    #include <time.h>
    time_t time_hostvar;
EXEC SQL END DECLARE SECTION;
:
:
}
```

Figure 1-22
*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 ESQL/C source file. However, to define host variables in 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 1-23, which leads to error -33051: Syntax error on identifier or symbol 'name_hostvar with the example in [Figure 1-21 on page 1-69](#) which does not. The only difference is that in the example in [Figure 1-21](#), 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 10;
typedef NAME char[LEN];
:
:
/*name.ec*/
#include "i.h"
main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
    NAME name_hostvar;
EXEC SQL END DECLARE SECTION;
:
:
}
:
:
...Leads to error -33051...
```

Figure 1-23
*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 ESQL/C preprocessor runs on a file, it expands all the contents of header files inside the 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 1-28](#).

Excluding Statements Inside C Header Files

If the ESQL/C preprocessor has problems with certain statements in C header files, you can exclude specific lines from the preprocessing that the ESQL/C preprocessor performs as shown in the example in Figure 1-24.

```
/*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 1-24
*ESQL/C and C
Excerpt That Uses a
Common C Header
File*

SQL Keyword Protection

If the code in the ESQL/C files is passed unprotected to the C preprocessor before it is passed to the 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 ESQL/C preprocessor, the utility **eprotect** runs before the C preprocessor runs on the ESQL/C source file. The **eprotect** utility prepends 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 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 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 ESQL/C preprocessor on the output of the C preprocessor.

SQL Keyword Protection and the \$ Symbol

All SQL statements within 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 prepend the **SQLKEYWORD_** prefix on SQL keywords that appear in SQL statements that begin with the dollar sign (\$) symbol.



Important: If you want to run the C preprocessor on your ESQL/C source file before the ESQL/C preprocessor does, 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 **\$** symbol.

WIN NT/95

Preprocessor Options Specific to Windows Environments

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

Line Wrapping

The 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 ESQL/C preprocessor generates error and warning messages when it processes an 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 ESQL/C source file that caused the error or warning.

Setting Tab Stops

By default, the 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
```



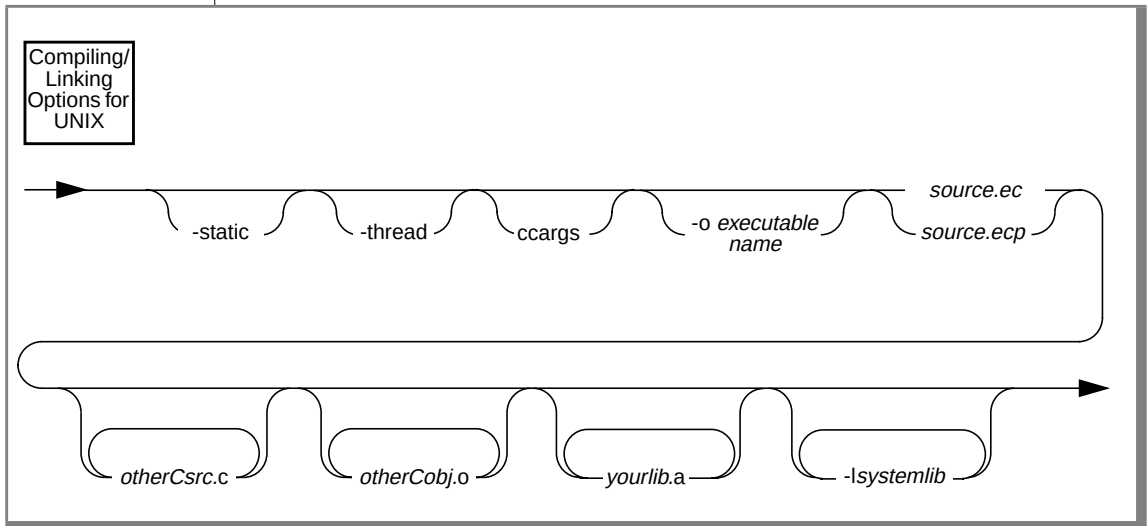
Compiling and Linking Options of the esql Command

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

UNIX

Syntax of the Compiling/Linking Options for UNIX

The following diagram shows the compiling and linking options that can be passed to the **esql** command.



<code>ccargs</code>	are command-line arguments that esql does not interpret but passes to the C compiler.
<code>-lsystemlib</code>	indicates other system libraries that you want to link, where <i>systemlib</i> is a string that is the abbreviated name of the library and is appended to the constant l .
<code>-o executable name</code>	specifies the name of the executable file.
<code>otherCobj.o</code>	specifies a C object file that you want to link with <i>esqlfile.ec</i> .

<i>otherCsrc.c</i>	specifies a C source file that you want to compile and link with <i>esqlfile.ec</i> .
-static	links Informix static libraries instead of the default Informix shared libraries.
-thread	links Informix thread-safe libraries instead of the default Informix shared libraries.
<i>yourlib.a</i>	specifies your own special static library that you want to link.

Naming the Executable File on UNIX

When the **esql** command successfully completes, it creates a C object file for your ESQL/C program. By default, the C compiler names this object file **a.out**. Often, you want the executable file to have the same name as the ESQL/C source file but without the **.ec** extension.

You can explicitly specify the name of the object file with the **-o** option. For example, the following **esql** command produces an executable file that is called **custinpt** (instead of **a.out**):

```
esql -o custinpt custinpt.ec
```

You can also create a shared object file from your source **.ec** file by giving standard C compiler options for shared objects. The **esql** command passes these options to the C compiler:

```
esql -G -o l esqlfile.so esqlfile.ec
```

Passing Options to the C Compiler

If you list options in the command line that the **esql** command does *not* support, it passes the options to the C compiler. For example, the following **esql** command in a UNIX environment passes the **-c** option to the C compiler because the **esql** command does not recognize the **-c** option.

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

For most C compilers, the **-c** option suppresses the loading phase of compilation and forces creation of an object file, even if you compile only one program.

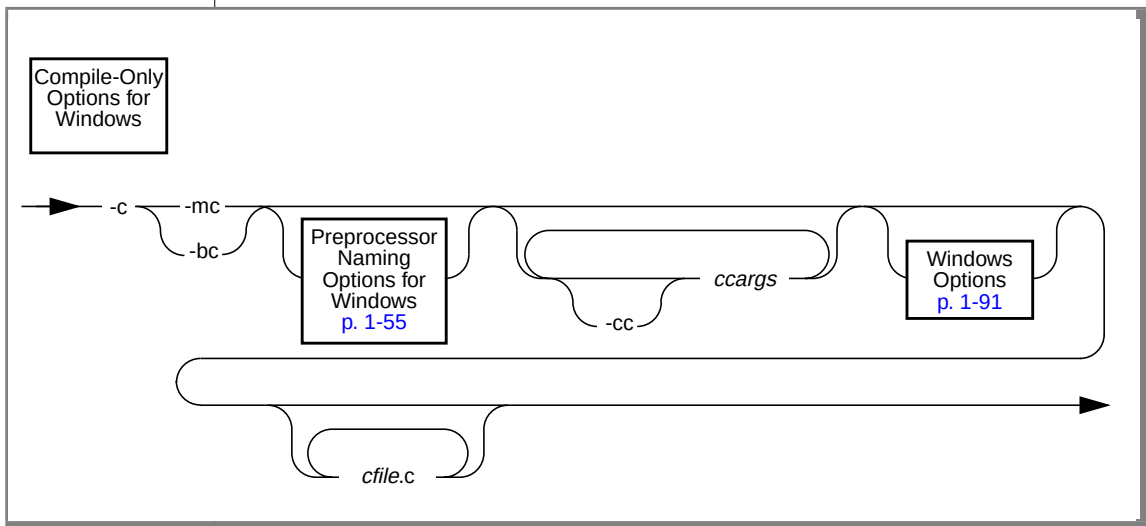
Specifying a Particular C Compiler

The **INFORMIXC** environment variable lets you specify the name or pathname of the C compiler that the **esql** command uses to generate compiled files. If you do not set **INFORMIXC**, the default compiler is **cc**. For a full description of this environment variable, refer to the [Informix Guide to SQL: Reference](#). ♦

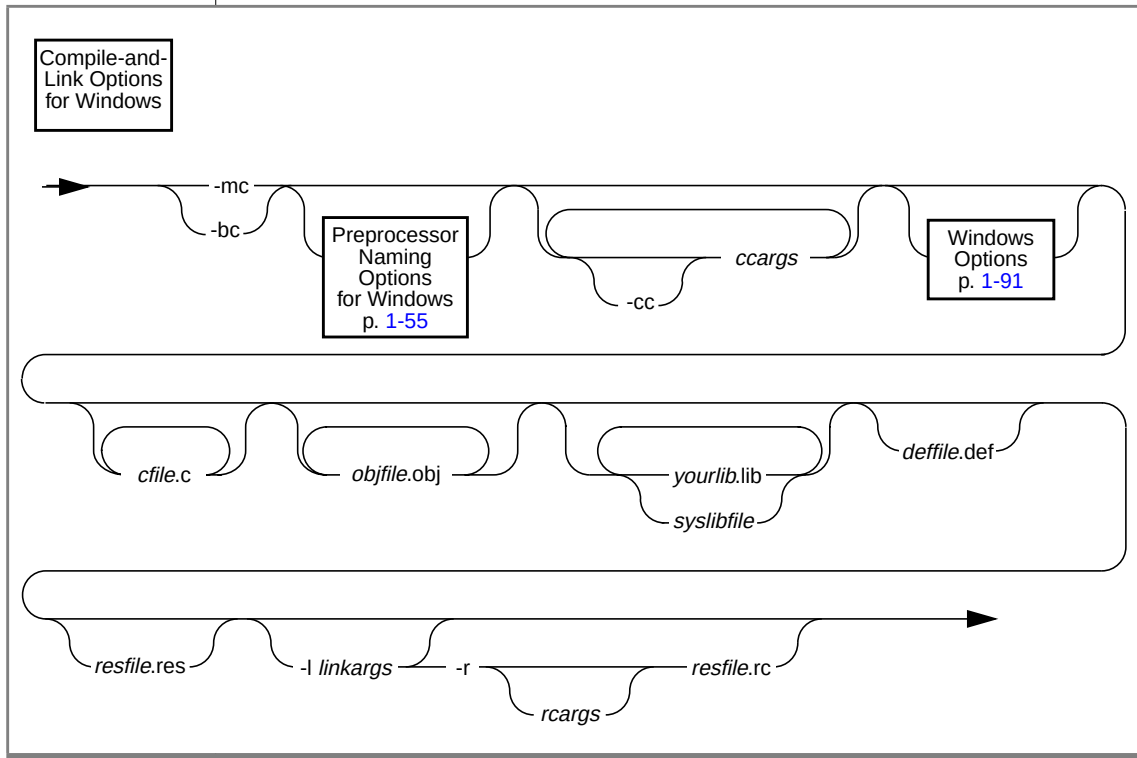
WIN NT/95

Syntax of the Compiling/Linking Options for Windows Environments

The following diagram shows the syntax of the **esql** processor options available when you want to only preprocess and compile an ESQL/C program but do not want to link it.



The following diagram shows the syntax of the **esql** processor options that are available in a Windows environment when you want to preprocess, compile, and link an ESQL/C program.



- c** preprocesses and compiles but does not link.
- bc** invokes the Borland C compiler.
- cc** indicates that you want to pass a C compiler option that has the same name as an ESQL/C processor option. This option tells the ESQL/C processor not to interpret the next option.
- ccargs* are command-line arguments that **esql** does not interpret but passes to the C compiler.
- cfile.c* indicates C source file(s) that you want to compile and link.

deffile.def	indicates a module-definition file that you want to link.
-l linkargs	marks the end of the compiler arguments and the start of linker arguments.
-mc	invokes the Microsoft C compiler, which is the default C compiler.
objfile.obj	indicates other C object file(s) that you want to link.
-r rcargs	marks the end of the linker arguments (or the end of the compiler arguments, if no linker arguments are specified) and the start of resource compiler arguments.
resfile.rc	specifies your own resource file containing resource script statements that you want to send to the resource compiler. The resource compiler compiles the .rc file into an .res file, which is linked into the target ESQL/C program.
resfile.res	specifies your own compiled resource files that you want linked.
syslibfile	indicates other system libraries that you want linked.
yourlib.lib	specifies your own special library that you want linked.

Naming the Executable File in Windows Environments

When **esql** successfully completes, it creates a *target* ESQL/C program. For the ESQL/C program, this target can be either an executable file (**.exe**) or a DLL file (**.dll**). (The **-target** command-line option determines whether **esql** creates an executable or DLL file. For more information, see [“Syntax of the Options for Windows Environments” on page 1-90.](#))

By default, the name of this target file is the name of the first ESQL/C source file on the **esql** command line. Often, you want the executable file to have a name that clearly identifies the purpose of the executable program.

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** (instead of **custinpt.exe**):

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

Passing Options to the C Compiler in Windows Environments

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 a C compiler option that has the same name as an ESQL/C processor option, precede the C compiler option with the **-cc** option. For example, the following **esql** command passes the **-od** option to the C compiler, suppressing interpretation of the option by the ESQL/C processor:

```
esql -cc -od demo1.ec
```

Specifying a Particular C Compiler in Windows Environments

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 reside on your computer before you can compile an ESQL/C program. The default C compiler option, **-mc**, invokes the Microsoft compiler. To choose the Borland compiler, use the **-bc** option.

Compiling Without Linking

By default, the ESQL/C command processor preprocesses, compiles, and links the ESQL/C program and creates either an executable file or a DLL. To suppress the linking of your 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 ESQL/C source file (**.ec**).

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

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



WIN NT/95

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 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 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 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 1-82](#).

Creating a Response File

You can specify the command-line arguments for the ESQL/C command processor in a *response file* and specify the filename for the ESQL/C processor. The ESQL/C response file is a text file with ESQL/C command-line options and filenames, 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 filenames **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 ESQL/C command processor implicitly passes compiler and linker flags to the supported C compilers. [Figure 1-25 on page 1-83](#) 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.

Figure 1-25
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 -etarget -subtype -libtlog -libtlg	-c -Tpe -M -DEF:deffile -subsystem %INFORMIXDIR%\lib\isqlb09a.lib %INFORMIXDIR%\lib\igl4b303.lib %INFORMIXDIR%\lib\iglx303.lib %INFORMIXDIR%\lib\igo4b303.lib c0t32.obj libset
	dll	-target:dll -wd	-c -I%INFORMIXDIR%\incl\esql -etarget -subtype -libtlog -libtlg	-c -Tpd -M -DEF:deffile -subsystem %INFORMIXDIR%\lib\isqlb09a.lib %INFORMIXDIR%\lib\igl4b303.lib %INFORMIXDIR%\lib\iglx303.lib %INFORMIXDIR%\lib\igo4b303.lib c0d32.obj libset

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

Term	Definition
<i>deffile</i>	name of a .def file. (The -DEF option executes only if you specify a .def file on the command line.)
<i>libset</i>	library set (depends on whether the application is WINDOWS or CONSOLE).
<i>libtlg</i>	-D_RTLDLL for a dynamic library or “ ” for a shared library.
<i>libtlog</i>	-WM for a multithread library or “ ” for a single-thread library.
<i>proctype</i>	type of processor (X86).
<i>subsystem</i>	ap for a console subsystem or aa for a Windows subsystem.
<i>subtype</i>	WC for a console executable, W for a Windows executable file, WCD for a console DLL, or WD for a Windows DLL.
<i>systype</i>	type of subsystem (WINDOWS or CONSOLE).
<i>t</i>	X for a console subsystem and W for a Windows subsystem.
<i>target</i>	name of the executable file (name of first .ec file or the name specified by the -o command line option).
<i>threadtype</i>	type of thread option (ML, MT, MD, depending on the value of the -runtime command-line option).

For more information on the **-target**, **-wd**, and **-we** command-line options, see [page 1-90](#).



Important: As the **Linker** column of the table in [Figure 1-25 on page 1-83](#) indicates, the library file in the **%INFORMIXDIR%\lib** directory is **isqlt09a.lib**.

The library set that the linker uses depends on whether you are creating a Windows or console application. Figure 1-26 lists the library sets that the indicated **esql** options use.

Figure 1-26
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



General Options That Affect Linking

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

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 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 ESQL/C application. ♦

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

```
esql -libs
```

WIN NT/95



Specifying Versions of Informix General Libraries

By default, the **esql** command links the shared libraries for the 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 in the thread-safe versions of the Informix shared libraries.
- The **-static** option tells the preprocessor to link in the static libraries for the Informix general libraries in a UNIX environment. ♦

You can combine these options to tell the preprocessor to link in the thread-safe versions of the Informix static libraries. For more information on this topic, see [“The esql Command” on page 13-6](#).

UNIX

Special Linking Options for Windows Environments

The following sections give linking options that can only be used 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

WIN NT/95

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 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 ESQL/C product includes the following dynamic link libraries (DLLs):

- The ESQL client-interface DLL (**isqlt09a.dll**) contains the ESQL/C library functions that the 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 1-94.
- 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, ESQL/C calls the **sqlauth()** function, which **esqlauth.dll** defines. For more information on **sqlauth()**, see [“Connection Authentication Functionality in a Windows Environment”](#) on page 12-13.
- The Registry DLL, **iregt07b.dll**, is used by the **Setnet32** utility and the INFORMIX-Connect library to set and access configuration information in the Registry.
- The **igl4b304.dll**, **igo4g303.dll**, and **iglxg303.dll** DLLs are required for Global Language Support (GLS). For more information about code-set conversion, see the [Informix Guide to GLS Functionality](#). ♦

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

GLS

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 need to 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 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 ESQL/C library routine or SQL statement:

```
#include "infxexp.c";
```

The **infxexp.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 need to include the **infxexp.c** file only once, in the **main()** routine (once per process), before the first 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. ♦

Windows Operating System Processor Options Available to the esql Command

The options that the **esql** command processor supports for Windows environments enable you to control the following features of your ESQ/C program:

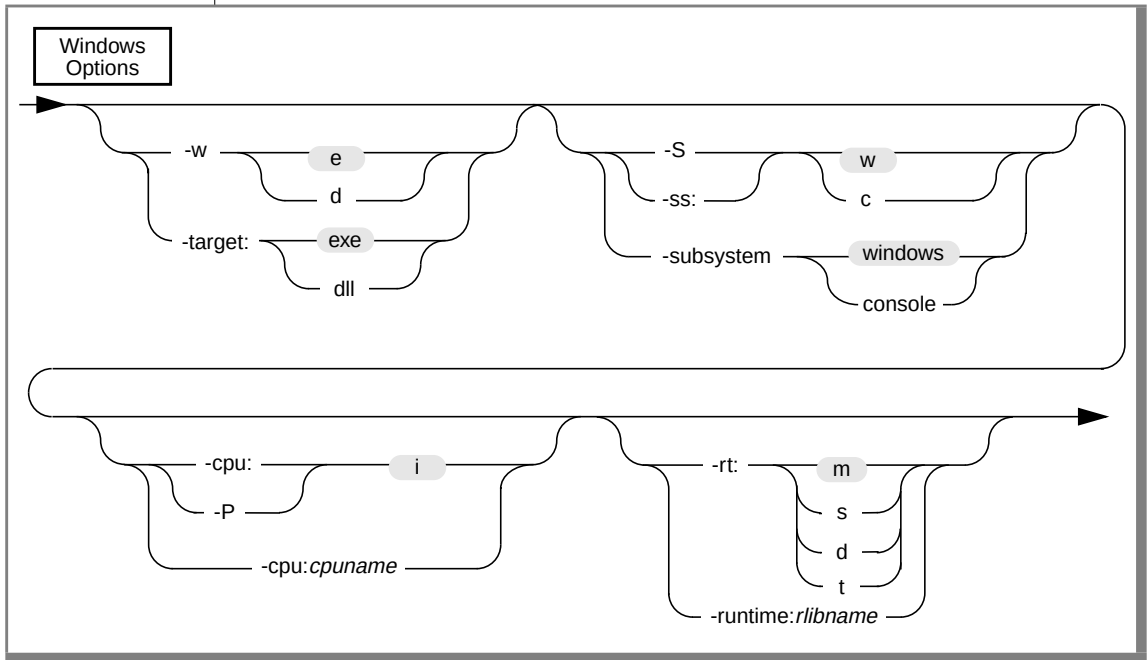
- Whether to create an executable file (**.exe**) or a DLL (**.dll**) for the target ESQ/C application
- Whether to create an ESQ/C application that runs in the Windows environments or on console subsystem
- Which runtime libraries to link into the target ESQ/C application

This section describes the syntax of the **esql** processor options for Windows environments.

Syntax of the Options for Windows Environments

[Figure 1-27 on page 1-91](#) shows the syntax of the processor options that are available when you use **esql** either to preprocess and compile only (**-c** command-line option) or to preprocess, compile, and link an ESQ/C program.

Figure 1-27
Windows Environment Options available to esql



- cpu:** indicates that the target ESQL/C program is to run on the central processing unit (CPU) that *cpuname* specifies. The Borland C compiler supports only Intel 386 CPUs, so it generates the code for a 386 CPU regardless of the **cpu** variable.
- cpu:i** indicates that the CPU type for the target ESQL/C program is Intel 386 (same as **-cpu:i386** and **-Pi**). This option specifies the default CPU type that **esql** uses. Informix maintains this option for backward compatibility.
- cpuname* specifies the name of the type of CPU to use for the target ESQL/C program. Valid value is **i386**, for an Intel 386 CPU. The default CPU type that **esql** uses is **i386**.

-Pi	indicates that CPU type is an Intel 386 (same as -cpu:i and -cpu:i386). This option is case sensitive. ESQL/C supports this option for compatibility with INFORMIX-ESQL/C for Windows NT, Version 5.01. This option specifies the default CPU type that esql uses.
<i>rlibname</i>	specifies the name of the C runtime library to link into the target ESQL/C program. Valid values are libc , libcmt , and msvcrt . The default C runtime library that esql links is msvcrt .
-rt:d	indicates that the target ESQL/C program is to include the multithreaded C msvcrt runtime DLL (same as -runtime:msvcrt). If you do not choose -rt:d , -rt:m , or -rt:s , esql assumes -rt:d , which is the default value for the -rt option. (For Borland C, the runtime DLL is -runtime:cw32mti .)
-rt:m	indicates that the target ESQL/C program is to include the static, multithreaded C libcmt runtime library (same as -runtime:libcmt). (For Borland C, the runtime DLL is -runtime:cw32mt .)
-rt:s	indicates that the target ESQL/C program is to include the static, single-threaded C libc runtime library (same as -runtime:libc). If you do not choose -rt:d , -rt:m , or -rt:s , esql assumes -rt:d . (For Borland C, the runtime DLL is -runtime:cw32 .)
-rt:t	indicates that the target ESQL/C program is to include the single-threaded C CW32i runtime DLL (same as -runtime:cw32i). This option is for the Borland C compiler only.
-runtime:	indicates that the esql processor is to link the runtime library that <i>rlibname</i> specifies into the target ESQL/C program.
-ss:c	creates a console-based application for the target ESQL/C program (same as -Sc and -subsystem:console). If you do not choose -ss:c or -ss:w , esql assumes -ss:w .

-ss:w	creates a Windows-based application for the target ESQL/C program (same as -subsystem:windows and -Sw). This option specifies the default subsystem type that esql uses.
-subsystem:console	creates a console-based application for the target ESQL/C program (same as -Sc and -ss:c). If you do not choose -subsystem:console or -subsystem:windows , esql assumes -subsystem:windows .
-subsystem:windows	creates a Windows-based application for the target ESQL/C program (same as -ss:w and -Sw). This option specifies the default subsystem-type that esql uses.
-Sc	creates a console-based application for the target ESQL/C program (same as -ss:c and -subsystem:console). If you do not choose -Sc or -Sw , esql assumes -Sw . This option is case sensitive. ESQL/C supports this option for compatibility with INFORMIX-ESQL/C for Windows NT, Version 5.01.
-Sw	creates a Windows-based application for the target ESQL/C program (same as -subsystem:windows and -ss:w). This option specifies the default subsystem type that esql uses. This option is case sensitive. ESQL/C supports this option for compatibility with INFORMIX-ESQL/C for Windows NT, Version 5.01.
-target:dll	creates a Windows DLL (.dll) for the target ESQL/C program (same as -wd). If you do not choose -target:exe or -target:dll , esql assumes -target:exe .
-target:exe	creates a Windows executable file (.exe) for the target ESQL/C program (same as -we). This option specifies the default target type that esql uses.
-wd	creates a Windows DLL (.dll) (same as -target:dll). If you do not choose -we or -wd , esql assumes -we .
-we	creates a Windows executable file (.exe) (same as -target:exe). This option specifies the default target type that esql uses. ♦

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 reside in memory. This situation is inefficient.

But if a function is dynamically linked, Windows NT and Windows 95 first checks memory to see if a copy of the function already resides 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 ESQL/C library functions, as well as other internal functions, are contained in the ESQL client-interface DLL. To use these functions in your 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

To enable your ESQL/C application to access the ESQL client-interface DLL, Informix provides the *import* library of the 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 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 ESQL/C program.

Locating a DLL

During the development of your application, the 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.

Windows NT and Windows 95 search directories for your DLL in the following order:

1. The directory from which you loaded the application
2. The current directory (where the executable file exists or the working directory that the Program Item Properties value for the icon specifies)

For Windows NT, the 32-bit Windows environment directory, SYSTEM32. (To get the path of this directory, use the **GetSystemDirectory()** function.) ♦

For Windows 95, the Windows system directory. (To get the path of this directory, use the **GetSystemDirectory()** function.) ♦

WIN NT

WIN 95

WIN NT

3. For Windows NT only, the 16-bit Windows environment system directory, **SYSTEM** ♦
4. The Windows directory
To get the path of this directory, use the **GetWindowsDirectory()** function.
5. Directories that your **PATH** environment variable lists

Building an Application DLL

You can tell the ESQL/C processor to build an ESQL/C program as a DLL (**.dll** file) with the **-target** (or **-wd**) command-line option. Such an ESQL/C program is called an *application DLL*.

To build an ESQL/C program as a DLL, follow the guidelines for general-purpose DLLs. For more information, refer to your system documentation. Compile the 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 demo program in the **%INFORMIXDIR%\demo\wdemo** directory. The 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. ♦

Setting and Retrieving Environment Variables in Windows Environments

You might change the settings of environment variables or create new ones 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 ESQL/C library functions are available for setting and retrieving environment variables.

Function Name	Description
ifx_putenv()	Modifies or removes an existing environment variable or creates a new one
ifx_getenv()	Retrieves the value of an environment variable



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**. Informix recommends 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 1-104](#).

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 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 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 ESQL/C **ifx_getenv()** function to retrieve the variable into the runtime environment segment.

For environment variable entries, observe the following guidelines:

GLS

- If you plan to set any Informix environment variables with **ifx_putenv()**, have the application set all of them before it calls any other ESQL/C library routine, including **ifx_getenv()**, or any SQL statement. The first call to any other 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, 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.*



ifx_putenv()

Purpose

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 is 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 only *varname*= (a null string).

The **ifx_putenv()** function sets Informix variables first and then non-Informix variables. For a list of Informix environment variables, see [“Fields of the InetLogin Structure” on page 1-104](#).

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.

ifx_getenv()

Purpose

The **ifx_getenv()** function retrieves the value of a current environment variable.

Syntax

```
char *ifx_getenv( varname );  
const char *varname;
```

varname is 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 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.



Warning: 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.

InetLogin Structure



Important: Informix supports the **InetLogin** structure for backward compatibility only. For new development, Informix recommends that you use the **ifx_getenv()** and **ifx_putenv()** functions instead.

An ESQ/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 C structure that the **login.h** header file defines. To use this structure in your ESQL/C program, you must include **login.h** in your source file (.ec). (For more information on **login.h**, see [Figure 1-7 on page 1-25.](#))

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

Figure 1-28 defines the fields in the **InetLogin** structure.

Figure 1-28
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 backward compatibility for client applications that are based on earlier versions of Informix products
DbMoney	char[19]	Specifies the value for the DBMONEY environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix products
DbTime	char[81]	Specifies the value for the DBTIME environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix products
DbTemp	char[81]	Specifies the value for the DBTEMP environment variable

(1 of 6)

InetLogin Field	Data Type	Purpose
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
DbNls	char[3]	Specifies the value for the DBNLS environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products
CollChar	char[3]	Specifies the value for the COLLCHAR environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products
Lang	char[81]	Specifies the value for the LANG environment variable for the database locale Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products
Lc_Collate	char[81]	Specifies the value for the LC_COLLATE environment variable for the database locale Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products

(2 of 6)

InetLogin Field	Data Type	Purpose
Lc_CType	char[81]	Specifies the value of the LC_CTYPE environment variable for the database locale Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products
Lc_Monetary	char[81]	Specifies the value of the LC_MONETARY environment variable for the database locale Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products
Lc_Numeric	char[81]	Specifies the value of the LC_NUMERIC environment variable for the database locale Provides backward compatibility for client applications that are based on earlier versions of Informix NLS products
Lc_Time	char[81]	Specifies the value for the LC_TIME environment variable for the database locale Provides backward compatibility for client applications that are based on earlier versions of 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

(3 of 6)

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]	<i>Reserved for future use</i>
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 backward compatibility for client applications that are based on earlier versions of Informix products
DbApiCode	char[24]	Specifies the value for the DBAPICODE environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix products

(4 of 6)

InetLogin Field	Data Type	Purpose
DbAsciiBc	char[2]	Specifies the value for the DBASCIIBC environment variable Provides backward compatibility for client applications that are based on earlier versions of 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 backward compatibility for client applications that are based on 4.x and 4.x versions of Informix Asian Language Support (ALS) products
DbConnect	char[2]	Specifies the value for the DBCCONNECT environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix products
DbCsConv	char[9]	Specifies the value for the DBCSCONV environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix products
DbCsOverride	char[2]	Specifies the value for the DBCISOVERRIDE environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix Asian Language Support (ALS) products
DbCsWidth	char[12]	Specifies the value for the DBCWIDTH environment variable Provides backward compatibility for client applications that are based on earlier versions of Informix products
DbFltMsk	char[4]	Specifies the value for the DBFLTMASK environment variable

(5 of 6)

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

(6 of 6)

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 ESQL/C program must allocate.

InetLogin Field Values

Your application must set **InetLogin** values *before* it executes the SQL statement or ESQL/C library function that needs the configuration information. Informix recommends 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.

Figure 1-29 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 1-29
*User Dialog Box for
Login Parameters*

Figure 1-30 shows a code fragment that sets login values in the **InetLogin** structure. The application could obtain these values from the end user through a dialog box (such as the one in Figure 1-29).

Figure 1-30
Code to Prompt the User for InetLogin Values

```
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);
```



In [Figure 1-30 on page 1-110](#), if the user enters host information, the fragment sets the **InetLogin.Host** and **InetLogin.User** fields for the **mainssvr** database server to the user-specified names of the host name and user name, respectively. If the user does *not* enter host information, ESQL/C uses the HOST and USER Registry values from the subkey for the **mainssvr** database server.

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

Precedence of Configuration Values

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

1. The **InetLogin** structure

If the application uses the **InetLogin** structure, 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 desired configuration information in **InetLogin**, ESQL/C checks for this information in its copy of the Registry information. For more information on how to set the Registry, see the *Informix Client Products Installation and Configuration Guide for Microsoft Windows Environments*.

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 have not set the corresponding Registry value, the application uses its default value.



Important: The first time that the application requires configuration information, ESQL/C reads this information from the Registry and stores it in memory. For subsequent references to Registry information, 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 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, 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**, refer to the “Informix Client Products Installation and Configuration Guide for Microsoft Windows Environments.” ♦



WIN NT/95

Using Host Variables in Windows Environments

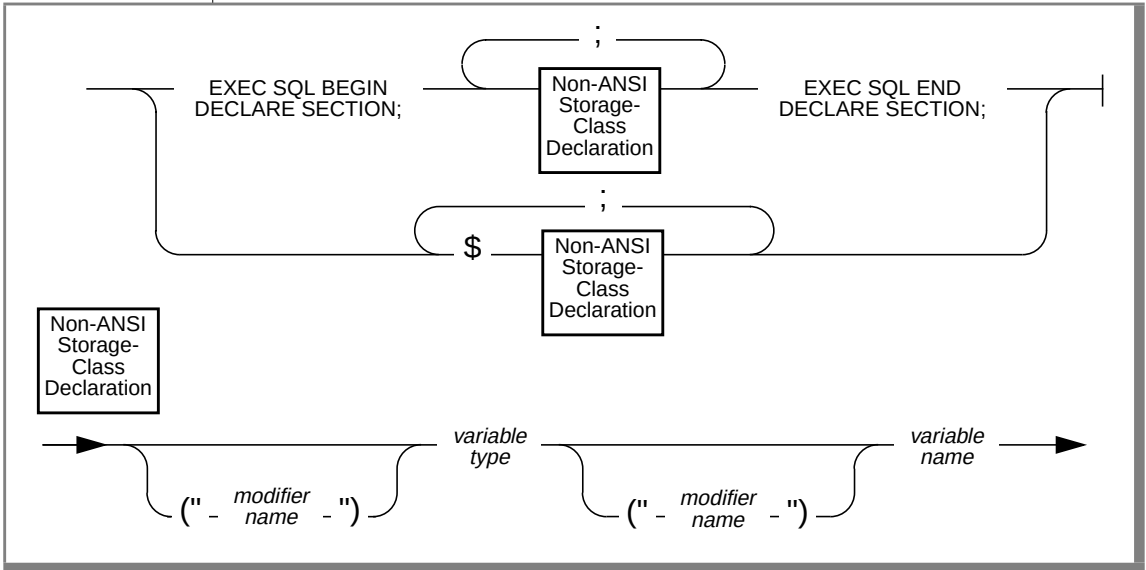
This section describes the following topics about ESQL/C host variables that are unique to the Windows environments:

- How to declare host variables with non-ANSI storage-class modifiers
- How global 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 ESQL/C host-variable declarations, the ESQL/C preprocessor supports the form of the ANSI syntax that [Figure 1-31 on page 1-113](#) shows.

Figure 1-31
ESQL/C Syntax for Non-ANSI Storage-Class Specifiers



Element	Purpose	Restrictions	Syntax
<i>modifier name</i>	The 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.
<i>variable name</i>	The identifier name of the ESQL/C host variable	None.	See “Declaring a Host Variable” on page 1-27 .
<i>variable type</i>	The 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 1-27 .

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 **dlexport**), *var_type* is the data type of the variable, and *var_name* is the variable name.

To enable you to declare ESQL/C host variables as extended storage-class variables, the 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 could declare similar macros to simplify declaration of variables to be exported (or imported) to the dynamic link library (DLL), as follows:

```
#define DLLEXPORT __declspec(dlexport);
:
EXEC SQL BEGIN DECLARE SECTION;
    @("DLLEXPORT") int winHdl;
EXEC SQL END DECLARE SECTION;
```

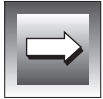
Using Global ESQL/C Variables

In earlier versions of the ESQL/C product, ESQL/C provided several global variables to support different features. Figure 1-32 describes these global variables.

Figure 1-32
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.”
SQLCODE sqlca.sqlcode	An Informix-specific status code as an integer value For more information about SQLCODE, see Chapter 11, “Exception Handling.”
sqlca structure	Informix-specific diagnostic information For more information about this structure, see Chapter 11, “Exception Handling.”
FetBufSize	The size of the fetch buffer For more information on FetBufSize , see Chapter 14, “Using Dynamic SQL.”
InetLogin structure	Environment information for the client ESQL/C application. For more information, see “InetLogin Structure” on page 1-103.

In Windows environments, ESQL/C implements the global variables in Figure 1-32 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 ESQL/C code. You can still use these functions in the same context as their global-variable counterparts. ♦



A Sample 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 the on-line versions of this and other demonstration programs in the **\$INFORMIXDIR/demo/esqlc** directory. This and other demonstration programs included throughout this book are not available for Windows environments.*

Compiling the demo1 Program

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

```
esql -o demo1 demo1.ec
```

The **-o demo1** option causes the executable program to be named **demo1**. Without the **-o** option, the name of the executable program defaults to **a.out**. For more information on the **esql** command, see [“Using the esql Command” on page 1-47](#).

Guide to demo1.ec File

The sample 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 last names of customers whose last name is 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. Your 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

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( "DEM01 Sample ESQL Program running.\n\n");
11 EXEC SQL connect to 'stores7';

12 EXEC SQL declare democursor cursor for
13     select fname, lname
14     into :fname, :lname
15     from customer
16     where lname < 'C';

17 EXEC SQL open democursor;
```

Lines 10 to 11

The **printf()** function displays text to identify the program and to notify the user when the program begins to execute. The **CONNECT** statement initiates a connection to the default database server and opens the **stores7** 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 12 to 16

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 last names of those customers whose last name (**lname**) begins with a letter less than 'C'.

Line 17

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

```

18  for (;;)
19  {
20      EXEC SQL fetch democursor;
21      if (strcmp(SQLSTATE, "00", 2) != 0)
22          break;

23      printf("%s %s\n",fname, lname);
24  }

25  if (strcmp(SQLSTATE, "02", 2) != 0)
26      printf("SQLSTATE after fetch is %s\n", SQLSTATE);

27  EXEC SQL close democursor;
28  EXEC SQL free democursor;

```

Lines 18 to 24

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 of this manual.

Lines 25 to 26

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

Lines 27 to 28

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

```
29 EXEC SQL disconnect current;  
30 printf("\nDEM01 Sample Program over.\n\n");  
31 }
```

Lines 29 to 31

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 line 31 marks the end of the **main()** function and of the program.

INFORMIX-ESQL/C Data Types

Choosing Data Types for Host Variables	2-3
Defined Constants for Data Types	2-7
Header Files for Data Types	2-11
The BOOLEAN Data Type	2-13
Data Conversion.	2-15
Fetching and Inserting with Host Variables	2-15
Converting Numbers and Strings	2-16
Converting Floating-Point Numbers to Strings	2-18
Converting BOOLEAN Values to Characters	2-18
Converting DATETIME and INTERVAL Values	2-18
Converting Between VARCHAR and Character Data Types	2-19
Arithmetic Operations	2-19
Converting Numbers to Numbers	2-19
Operations That Involve a Decimal Value	2-20
Data-Type Alignment Library Functions	2-23
risnull()	2-24
rsetnull()	2-27
rtypalign()	2-30
rtypmsize()	2-33
rtypname()	2-36
rtypwidth()	2-39

T

his chapter contains information about the correspondence between SQL and C data types and how to handle data types in an INFORMIX-ESQL/C program. The chapter covers the following topics:

- 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 use a host variable in an SQL statement, you must declare the host variable with a data type that is compatible with the SQL data type of the data that the database server returns. The [Informix Guide to SQL: Reference](#) describes in detail the SQL data types that are available for database columns. For each table column that you access from a database, you must declare a host variable of the appropriate C or ESQL/C data type. [Figure 2-1 on page 2-4](#) summarizes the correspondence between SQL data types and ESQL/C host-variable types for all database servers. [Figure 2-2 on page 2-5](#) summarizes the SQL data types and the corresponding ESQL/C host variable types that are specific to Universal Server.

Figure 2-1
Correspondence of SQL and ESQL/C Data Types

SQL Data Type	ESQL/C Predefined Data Type	C Language Type
BYTE	loc_t	
CHAR(<i>n</i>) CHARACTER(<i>n</i>)	fixchar array[<i>n</i>] or string array[<i>n</i>+1]	char array[<i>n</i> + 1] or char *
DATE	date	long int
DATETIME	datetime or dtime_t	
DECIMAL DEC NUMERIC MONEY	decimal or dec_t	
FLOAT DOUBLE PRECISION		double
INTEGER INT		4-byte integer
INTERVAL	interval or intrvl_t	
MULTISET(<i>e</i>)	collection	
NCHAR(<i>n</i>)	fixchar array[<i>n</i>] or string array[<i>n</i>+1]	char array[<i>n</i> + 1] or char *
NVARCHAR(<i>m</i>)	varchar[<i>m</i>+1] or string array[<i>m</i>+1]	char array[<i>m</i>+1]
SERIAL		4-byte integer
SMALLFLOAT REAL		float
SMALLINT		2-byte integer
TEXT	loc_t	
VARCHAR(<i>m</i> , <i>x</i>)	varchar[<i>m</i>+1] or string array[<i>m</i>+1]	char array[<i>m</i>+1]

Figure 2-2

Correspondence of SQL and ESQL/C Data Types Specific to INFORMIX-Universal Server

SQL Data Type	ESQL/C Predefined Data Type	C Language Type
BLOB	ifx_lo_t	
BOOLEAN	boolean	
CLOB	ifx_lo_t	
INT8	int8 or ifx_int8_t	8-byte integer
LIST(e)	collection	
LVARCHAR	lvvarchar	char
Opaque data type	lvvarchar, fixed binary, or var binary	
ROW(...)	row	
SERIAL8	int8 or ifx_int8_t	8-byte integer
SET(e)	collection	



To declare a host variable for an SQL data type, you must use the appropriate ESQL/C predefined data type, or structure, for the host variable. [Figure 2-3 on page 2-6](#) shows where you can find the description of how to use these specialized structures for the appropriate host data type for all database servers. [Figure 2-4 on page 2-7](#) shows where you can find the description of how to use these specialized structures that are specific to Universal Server.

Figure 2-3
References for Using Host Data Types

Data Type	Reference
BYTE	Chapter 8, “Working with Simple Large Objects”
CHAR	Chapter 3, “Working with Character and String Data Types”
DATETIME	Chapter 6, “Working with Time Data Types”
DECIMAL	Chapter 4, “Working with the DECIMAL Data Type”
FLOAT	Chapter 5, “Working with Numeric Data Types”
INTEGER	Chapter 5, “Working with Numeric Data Types”
INTERVAL	Chapter 6, “Working with Time Data Types”
LIST	Chapter 9, “Working with INFORMIX-Universal Server Complex Data Types”
MONEY	Chapter 4, “Working with the DECIMAL Data Type”
NCHAR	Chapter 3, “Working with Character and String Data Types”
NVARCHAR	Chapter 3, “Working with Character and String Data Types”
SERIAL	Chapter 5, “Working with Numeric Data Types”
SERIAL8	Chapter 5, “Working with Numeric Data Types”
SMALLFLOAT	Chapter 5, “Working with Numeric Data Types”
SMALLINT	Chapter 5, “Working with Numeric Data Types”
TEXT	Chapter 8, “Working with Simple Large Objects”
VARCHAR	Chapter 3, “Working with Character and String Data Types”

Figure 2-4
References for Using Host Data Types That Are Specific to INFORMIX-Universal Server

Data Type	Reference
BLOB	Chapter 7, “Working with INFORMIX-Universal Server Smart Large Objects”
BOOLEAN	“The BOOLEAN Data Type” on page 2-13
CLOB	Chapter 7, “Working with INFORMIX-Universal Server Smart Large Objects”
INT8	Chapter 5, “Working with Numeric Data Types”
LVARCHAR	Chapter 3, “Working with Character and String Data Types”
MULTISET	Chapter 9, “Working with INFORMIX-Universal Server Complex Data Types”
Opaque	Chapter 10, “Working with INFORMIX-Universal Server Opaque Data Types”
ROW	Chapter 9, “Working with INFORMIX-Universal Server Complex Data Types”
SET	Chapter 9, “Working with INFORMIX-Universal Server Complex Data Types”



Defined Constants for Data Types

The **sqltypes.h** file contains the following two sets of defined integer constants for the data types that Informix database servers use:

- Constants for Informix SQL data types, which apply to data types of columns in an Informix database
- Constants for ESQL/C data types, which apply to data types of host variables in an ESQL/C program

The Informix SQL data types are data types that you assign to a column when you create a table in an Informix database. The DESCRIBE statement uses these data type values for column values it describes. Figure 2-5 shows the SQL data type constants for all database servers. [Figure 2-6 on page 2-9](#) shows the SQL data type constants that are specific to Universal Server.

Figure 2-5
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

Figure 2-6*Constants for Informix SQL Column Data Types That Are Specific to INFORMIX-Universal Server*

SQL Data Type	Defined Constant	Integer Value
INT8	SQLINT8	17
SERIAL8	SQLSERIAL8	18
SET	SQLSET	19
MULTISET	SQLMULTISET	20
LIST	SQLLIST	21
ROW	SQLROW	22
Varying-length opaque type	SQLUDTVAR	40
Fixed-length opaque type	SQLUDTFIXED	41
LVARCHAR	SQLLVARCHAR	43
BOOLEAN	SQLBOOL	45



However, if your programs conform to the X/Open standards (compile with the **-xopen** option), you must use the data-type values that Figure 2-7 shows. Informix defines the constants for these values in the **sqlxtype.h** header file.

Figure 2-7*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

The **sqltypes.h** header file also contains constants for the ESQL/C data types. You assign an ESQL/C data type to a host variable in an ESQL/C program. Figure 2-8 shows these constants.

Figure 2-8
Constants for ESQL/C Host-Variable Data Types

ESQL/C Data Type	Constant	Integer Value
char	CCHARTYPE	100
short int	CSHORTTYPE	101
int (long or short depending on machine type)	CINTTYPE	102
long	CLONGTYPE	103
float	CFLOATTYPE	104
double	CDOUBLETTYPE	105
dec_t or decimal	CDECIMALTYPE	107
fixchar	CFIXCHARTYPE	108
string	CSTRINGTYPE	109
long or date	CDATETYPE	110
dec_t or decimal	CMONEYTYPE	111
datetime or dtime_t	CDTIMETYPE	112
loc_t	CLOCATORTYPE	113
varchar	CVCHARTYPE	114
intrvl_t or interval	CINVTYPE	115
char	CFILETYPE	116
int8	CINT8TYPE	117
collection	CCOLTYPE	118

(1 of 2)

ESQL/C Data Type	Constant	Integer Value
lvvarchar	CLVCHARTYPE	119
fixed binary	CFIXBINTYPE	120
var binary	CVARBINTYPE	121
boolean	CBOOLTYPE	122
row	CROWTYPE	◆ 123 ◆

(2 of 2)

You can use these ESQL/C data types as arguments for some of the functions in the ESQL/C library. When you need to use one of these functions, check the contents of the **sqltypes.h** file for the appropriate value.

Header Files for Data Types

To use an SQL data type, your program must include the appropriate ESQL/C header file. Figure 2-9 shows the relationship between host-variable data types and ESQL/C header files for all database servers. [Figure 2-10 on page 2-13](#) shows the relationship between host-variable data types and ESQL/C header files that are specific to Universal Server.

Figure 2-9
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
BYTE	loc_t	locator.h
CHAR(<i>n</i>) CHARACTER(<i>n</i>)	fixchar array[<i>n</i>] or string array[<i>n</i>+1]	Defined automatically
DATE	date	Defined automatically
DATETIME	datetime or dtime_t	datetime.h

(1 of 2)

SQL Data Type	ESQL/C or C Data Type	ESQL/C Header File
DECIMAL DEC NUMERIC MONEY	decimal or dec_t	decimal.h
FLOAT DOUBLE PRECISION	double	Defined automatically
INTEGER INT	long int	Defined automatically
INTERVAL	interval or intrvl_t	datetime.h
LVARCHAR	lvarchar	Defined automatically
MULTISET(E)	collection	Defined automatically
NCHAR(<i>n</i>)	fixchar array [<i>n</i>] or string array [<i>n</i> +1]	Defined automatically
NVARCHAR(<i>m</i>)	varchar [<i>m</i> +1] or string array [<i>m</i> +1]	Defined automatically
SERIAL	long int	Defined automatically
SMALLFLOAT REAL	float	Defined automatically
SMALLINT	short int	Defined automatically
TEXT	loc_t	locator.h
VARCHAR(<i>m,x</i>)	varchar [<i>m</i> +1] or string array [<i>m</i> +1]	Defined automatically

(2 of 2)

Figure 2-10
SQL Data Types and ESQL/C Header Files That Are Specific to INFORMIX-Universal Server

SQL Data Type	ESQL/C or C Data Type	ESQL/C Header File
BOOLEAN	boolean	Defined automatically
CLOB	ifx_lo_t	locator.h
INT8	int8	int8.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
SERIAL8	int8	int8.h
SET(e)	collection	Defined automatically

◆

The BOOLEAN Data Type

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 [Informix Guide to SQL: Reference](#). This section describes how to manipulate the 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 ESQL/C program, the following values are the only valid values that you can assign to **boolean** host variables:

Boolean Value	Representation
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 */
```



Data Conversion

When a discrepancy exists between the data types of two values, ESQ/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, ESQ/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 on fetching and inserting, see [“Fetching and Inserting with Host Variables.”](#) For more information on arithmetic operations, see [“Arithmetic Operations” on page 2-18.](#)

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 [Figure 2-1 on page 2-4](#), ESQ/C attempts to convert the data types. Similarly, if you try to *insert* a value from a host variable into a database column, ESQ/C might need to convert data types if the host variable and database column do not use the correspondences in [Figure 2-1](#). ESQ/C converts the data types only if the conversion is meaningful.

This section provides the following information on data conversion for fetching and inserting values with host variables:

- How ESQL/C converts between numeric and character data
- How ESQL/C converts floating-point numbers to strings
- How ESQL/C converts BOOLEAN values to characters ♦
- How ESQL/C converts DATETIME and INTERVAL values
- How ESQL/C converts between VARCHAR columns and character data

Converting Numbers and Strings

Before ESQL/C can convert a value from one data type to another, it must determine whether the conversion is meaningful. Figure 2-11 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, ESQL/C returns values that the **Results** column in Figure 2-11 describes.

Figure 2-11
Data Conversion Problems and Results

Conversion	Problem	Result
C → C	Does not fit	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 3-11.
N → C	None	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 2-17.

(1 of 2)

Conversion	Problem	Result
N → C	Does not fit	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, ESQL / C rounds the number. Asterisks appear only when the integer part does not fit.
C → N	None	ESQL / C determines the numeric data type based on the format of the character value; if the character contains a decimal point, ESQL / C converts the value to a DECIMAL value.
C → N	Not a number	The number is undefined; ESQL / C sets sqlca.sqlcode and SQLSTATE to indicate a runtime error.
C → N	Overflow	The number is undefined; ESQL / C sets sqlca.sqlcode and SQLSTATE to indicate a runtime error.
N → N	Does not fit	ESQL / C attempts to convert the number to the new data type. For information on possible errors, see the Informix Guide to SQL: Reference .
N → N	Overflow	The number is undefined; ESQL / C sets sqlca.sqlcode and SQLSTATE to indicate a runtime error.

(2 of 2)

In [Figure 2-11 on page 2-16](#), 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

ESQL/C can automatically convert floating-point column (data type of DECIMAL(*n*), FLOAT, or SMALLFLOAT) values between database columns and host variables of character type **char**, **varchar**, **string**, or **fixchar**. When ESQL/C converts a floating-point value to a character string whose buffer is not large enough to hold the full precision, ESQL/C rounds the value to fit it in the character buffer. For more information, see [“Implicit Data Conversion” on page 5-6](#).

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, see [“The BOOLEAN Data Type” on page 2-13](#). ♦

Converting DATETIME and INTERVAL Values

ESQL/C can automatically convert DATETIME and INTERVAL values between database columns and host variables of character type **char**, **string**, or **fixchar**. 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 6-35](#).

You can use 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 6-37](#).

Converting Between VARCHAR and Character Data Types

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 3-12](#).

Arithmetic Operations

When 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 on data conversion for arithmetic operations:

- How ESQL/C converts numeric values
- How 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, ESQL/C converts the values to the data type that Figure 2-12 indicates and then performs the operation.

Figure 2-12
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

Figure 2-12 shows that if 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), ESQL/C generates a result that has a DECIMAL data type. For more information about DECIMAL values, see [“Operations That Involve a Decimal Value” on page 2-19](#).

Operations That Involve a Decimal Value

The following table shows the numeric data types. Database columns use the SQL data types, and ESQL/C host variables use the corresponding ESQL/C data types.

SQL Data Type	ESQL/C Data Type
INTEGER	LONG INTEGER
SMALLINT	SHORT INTEGER
DECIMAL	DECIMAL
MONEY	DECIMAL
FLOAT	DOUBLE
SMALLFLOAT	FLOAT

When ESQL/C performs arithmetic operations on operands with numeric data types and one of the operands has a decimal value (SQL data type of DECIMAL or an ESQL/C data type of **decimal**), 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 ESQ/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 ESQ/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 ESQ/C **decimal** host variable.

For more information on how the **decimal** data type tracks precision and scale, see [“The decimal Data Type” on page 4-4](#).

To Convert the Non-DECIMAL Numeric Operand

ESQ/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 ESQ/C uses for the non-DECIMAL operand.

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

ESQ/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, ESQ/C adds trailing zeros to the operand with the smaller scale until the scales are equal.

To Obtain 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, ESQL / C converts the FLOAT value to a floating-point decimal of DECIMAL(16). The arithmetic result has a data type of DECIMAL(16).

- When both of the operands are fixed-point decimals, the arithmetic result is also a fixed-point decimal.

Figure 2-13 summarizes the rules for arithmetic operations on operands with definite scale (fixed-point decimals). In Figure 2-13, 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.

Figure 2-13
Precision and Scale of Fixed-Decimal Arithmetic Results

Operation	Precision and Scale of Result
Addition and Subtraction	Precision: $\text{MIN}(32, \text{MAX}(p_1 - s_1, p_2 - s_2) + \text{MAX}(s_1, s_2) + 1)$ Scale: $\text{MAX}(s_1, s_2)$
Multiplication	Precision: $\text{MIN}(32, p_1 + p_2)$ Scale: $s_1 + s_2$; If $(s_1 + s_2) > \text{precision}$, the result is a floating-point decimal number (no scale value).
Division	Precision: 32 Scale: 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, ESQL / C reports an error.

Data-Type Alignment Library Functions

The following 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
risnull()	Checks whether a C variable is null
rsetnull()	Sets a C variable to null
rtyppalign()	Aligns data on proper type boundaries
rtypmsize()	Gives byte size of SQL data types
rtypname()	Returns the name of a specified SQL data type
rtypwidth()	Returns minimum number of characters that a character data type needs to avoid truncation

When you compile your ESQL/C program with the **esql** command, **esql** calls on the linker to link these functions to your program. The following pages describe these functions.

risnull()

The **risnull()** function checks whether a C or an ESQL/C variable contains a null value.

Syntax

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

type is an integer that corresponds to the data type of a C or ESQL/C variable. This *type* can be any data type *except* **var binary** or an **lvvarchar** pointer variable. For more information, see [“Defined Constants for Data Types” on page 2-7](#).

ptrvar is a pointer to the C or ESQL/C variable.

Usage

The **risnull()** function determines whether ESQL/C variables of all data types *except* **var binary** and **lvvarchar** pointer variables contain a null value. To determine whether a **var binary** or **lvvarchar** pointer host variable contains null, use the **ifx_var_isnull()** macro. For more information, see [page 10-41](#).

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;
    long ret, exp_chk();

    EXEC SQL BEGIN DECLARE SECTION;
        long order_num, 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 11-59](#) or refer to the **exp_chk.ec** file for a listing of this exception-handling function.

Example 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.

rsetnull()

The **rsetnull()** function sets a C variable to a value that corresponds to a database null value.

Syntax

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

type is an integer that corresponds to the data type of a C or ESQL/C variable. This *type* can be any data type *except* **var binary** or an **lvvarchar** pointer variable. For more information, see [“Defined Constants for Data Types” on page 2-7](#).

ptrvar is a pointer to the C or ESQL/C variable.

Usage

The **rsetnull()** function sets to null ESQL/C variables of all data types *except* **var binary** and **lvvarchar** pointer host variables. To set a **var binary** or **lvvarchar** pointer host variable to null, use the **ifx_var_setnull()** macro. For more information, see [page 10-46](#).

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;
    long 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 11-59](#) or refer to the **exp_chk.ec** file for a listing of this exception-handling function.

Example 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.

rtypalign()

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

Syntax

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

pos is the current position in a buffer.

type is an integer that corresponds to the data type of a C or ESQL/C variable. This *type* can be any data type *except* **var binary**. For more information, see [“Defined Constants for Data Types” on page 2-7](#).

Usage

The **rtypalign()** and **rtypmsize()** functions are useful when you use an **sqllda** 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, ESQL/C stores the value of *type* in **sqllda.sqlvar->sqltype**.

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

Return Code

>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()
{
    int i, pos;
    long 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\type\t\tlen\tnext\taligned\n");          /* display column hdgs. */
    printf("\t\t\t\t\tpos\tpos\n\n");
    /*
     * For each column in the orders table
     */
    i = 0;
    pos = 0;
    while(i++ < sql_desc->sqlid)
    {
        /* 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", rtypename(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 11-59](#) or refer to the **exp_chk.ec** file for a listing of this exception-handling function.

Example 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.

rtpmsize()

The **rtpmsize()** function returns the number of bytes you must allocate in memory for the specified ESQL/C or SQL data type.

Syntax

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

sqltype is the integer code of the ESQL/C or SQL data type. For more information, see [“Defined Constants for Data Types” on page 2-7](#).

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

Usage

The **rtpalign()** and **rtpmsize()** 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.

Informix provides the **rtpmsize()** function 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 **rtpmsize()** determines sizes for character data, keep the following size information in mind:

- For CCHARTYPE (**char**) and CSTRINGTYPE (**string**), ESQL/C adds one byte to the number of characters for the null terminator.
- For CFIXCHARTYPE (**fixchar**), ESQL/C does not add a null terminator.

You can see an application of the **rtpmsize()** 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[];
{
    int i;
    char db_stmnt[50];
    long 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 sqllda */
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->sqlld; i++, col++)
printf("\t%-20s%-8s%3d\n", col->sqlname, rtypname(col->sqltype),
    rtypmsize(col->sqltype, col->sqlllen));

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 11-59](#) or refer to the **exp_chk.ec** file for a listing of this exception-handling function.

Example 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.

rtypname()

The **rtypname()** function returns a null-terminated string that contains the name of the specified SQL data type.

Syntax

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

sqltype is an integer code for one of the SQL data types. For more information, see [“Defined Constants for Data Types” on page 2-7](#).

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[];
{
    int i;
    long err_chk();
```

```

char db_stmt[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_stmt, "CONNECT TO %s", argv[1]);
exp_chk(db_stmt, 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, rtypename(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 11-59](#) or refer to the **exp_chk.ec** file for a listing of this exception-handling function.

rtypname()

Example 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.

rtypewidth()

The **rtypewidth()** 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

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

sqltype is the integer code of the SQL data type. For more information, see [“Defined Constants for Data Types” on page 2-7](#).

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

Usage

Informix provides the **rtypewidth()** function 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[];
{
    int i, numchars;

    long 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->sqlid; 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

```

For a complete listing of the **exp_chk()** function, see [“Guide to the exp_chk.ec File” on page 11-59](#) or refer to the **exp_chk.ec** file for a listing of this exception-handling function.

Example 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.
```


Working with Character and String Data Types

Character Data Types	3-3
The char Data Type	3-4
The fixchar Data Type	3-5
The string Data Type	3-5
The varchar Data Type	3-6
The lvvarchar Data Type	3-9
An lvvarchar Host Variable of a Fixed Size	3-10
The lvvarchar Pointer Host Variable.	3-10
Fetching and Inserting Character Data Types	3-11
Fetching and Inserting CHAR Data	3-11
Fetching CHAR Data	3-11
Inserting CHAR Data	3-12
Fetching and Inserting VARCHAR Data	3-12
Fetching VARCHAR Data.	3-13
Inserting VARCHAR Data.	3-14
Fetching and Inserting lvvarchar Data	3-17
Fetching lvvarchar Data	3-17
Inserting lvvarchar Data.	3-17
Fetching and Inserting with an ANSI-Compliant Database.	3-18
Character and String Library Functions.	3-19
bycmpr()	3-21
bycopy()	3-24
byfill()	3-26
byleng()	3-28
ldchar()	3-30
rdownshift()	3-32
rstod()	3-34
rstoi()	3-36

rstol()	3-38
rupshift()	3-40
stcat()	3-42
stchar().	3-44
stcmprr()	3-46
stcopy()	3-48
stleng().	3-50

T

his chapter explains how to use character data types in an INFORMIX-ESQL/C program. It contains 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 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 ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#) For information about SQL data types, see the [Informix Guide to SQL: Reference](#).

Character Data Types

ESQL/C supports four 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 ESQL/C predefined data types: **fixchar**, **string**, **varchar**
- The **lvvarchar** 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 [Informix Guide to GLS Functionality](#). ♦

IUS

GLS

The following two conditions determine which character data type to use:

- Whether you want ESQ/C to terminate the character data with the null character
- Whether you want ESQ/C to pad the character data with trailing blanks

Figure 3-1 summarizes the attributes of each of the character data types.

Figure 3-1
ESQ/C Character Data Types

ESQ/C Character Data Type	Null Terminated	Contains Trailing Blanks
char	✓	✓
fixchar		✓
string	✓	
varchar	✓	✓
lvarchar	✓ ♦	

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**, ESQ/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 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**, ESQL/C pads this value with trailing blanks up to the size of the host variable. 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 ESQL/C data type that holds character data that is null terminated and does *not* contain trailing blanks. 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 if the CHAR or VARCHAR column contains an empty string. To distinguish an empty string from a null string, the database server stores an empty string as a single blank character. A **string** host variable stores an empty string as a single blank followed by a null terminator. If the **lname** column in the **customer** table contains an empty string, the following SELECT statement puts the string "b\0" (where b represents a blank and \0 is the null terminator) into the **buffer** host variable.

```
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 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**, 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 ESQL / C libraries. This file defines the names and macro functions shown in Figure 3-2.

Figure 3-2
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.
VLENGTH(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 <i>min</i> and <i>max</i> , 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 **sllen** field of the **slda** 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 3-3 on page 3-8](#) 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()
{
    int vc_code;
    int max, min;
    int hv_length;

    EXEC SQL BEGIN DECLARE SECTION;
        int 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 3-3
*The varchar.ec
Demo Program*

The Ivarchar Data Type

The **Ivarchar** data type is an ESQL/C data type that holds character data of varying lengths. The **Ivarchar** 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 **Ivarchar** data type, 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 **Ivarchar** 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 10-6](#).



Important: You cannot retrieve or store smart large objects (CLOB or BLOB data types) from or to an **Ivarchar** host variable.

To declare an **Ivarchar** host variable for a character column (CHAR, VARCHAR, or LVARCHAR), use the **Ivarchar** keyword as the variable data type, as the following syntax shows.

```
+ Ivarchar variable name [ variable size ] ;
* variable name
```

Element	Purpose	Restrictions
variable name	Name of an Ivarchar variable of a specified size.	
variable size	Number of bytes to allocate for an Ivarchar variable of specified size.	Integer value can be between 1 and 32,768 (32 kilobytes).
*variable name	Name of an Ivarchar 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 Ivarchar pointer and var binary Library Functions” on page 10-32 .

Figure 3-4 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 3-4
Sample Ivarchar Host
Variables



Important: To declare an **lvarchar** host variable for the external format of an opaque data type, use the syntax described in [“Declaring Ivarchar Host Variables” on page 10-7](#).

An Ivarchar 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 will be truncated when the column is fetched. Use an indicator variable to check for truncation.

Because an **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 Ivarchar 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 an **lvarchar** pointer host variable.

For more information on the **ifx_var()** functions and on how to use the ESQL/C **lvarchar** host variable with opaque data types, see [Chapter 10](#), [“Working with INFORMIX-Universal Server Opaque Data Types.”](#) ♦

Fetching and Inserting Character Data Types

You can transfer character data between CHAR and VARCHAR columns and character (**char**, **string**, **fixchar**, **vchar**, or **lvchar**) 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 [Informix Guide to GLS Functionality](#). ♦

Fetching and Inserting CHAR Data

When an application uses a character host variable to fetch or insert a CHAR value, 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**, **vchar**, or **lvchar**) host variable. If the column data does not fit into the character host variable, ESQL/C truncates the data. To notify the user of the truncation, 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 1-42.

Inserting CHAR Data

An application can insert data from a character host variable (**char**, **string**, **fixchar**, **varchar**, or **lvvarchar**) into a database column of type CHAR. When an application inserts a character value into a CHAR database column, ESQL/C pads the value with blanks up to the size of the column. If the column size is less than the length of the host-variable value, ESQL/C truncates the value. In other words, ESQL/C always pads or truncates a character value to the size of that column when it inserts the value into a CHAR database column.

Although **char**, **varchar**, **lvvarchar**, and **string** host variables contain null terminators, ESQL/C never inserts these characters into a database column. (Host variables of type **fixchar** should never contain null characters.)

GLS

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 [Informix Guide to GLS Functionality](#). ♦

ANSI

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 3-18. ♦

Fetching and Inserting VARCHAR Data

When an application uses a character host variable to fetch or insert a VARCHAR value, ESQL/C must ensure that the character value fits into the host variable or database column. When ESQL/C calculates the length of a source item, it does not count trailing spaces. The following sections describe how ESQL/C performs the conversion of VARCHAR data to and from **char**, **fixchar**, and **string** character data types.

GLS

These conversions also apply to NVARCHAR data. For more information on the NVARCHAR data type, see the [Informix Guide to GLS Functionality](#). ♦

Fetching VARCHAR Data

Figure 3-5 shows the conversion of VARCHAR data when an application fetches it into host variables of **char**, **fixchar**, **lvarchar**, and **string** character data types.

Figure 3-5
Converting the VARCHAR Data Type to ESQL/C Character Data Types

Source	Destination	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.

Figure 3-6 shows examples of conversions from VARCHAR column data to character host variables that 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.

Figure 3-6
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

(1 of 2)

Source Type	Contents	Length	Destination Type	Contents	Indicator
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

(2 of 2)

Inserting VARCHAR Data

When an application inserts a value from a **char**, **varchar**, **lvarchar**, or **string** host variable into a VARCHAR column, ESQL/C preserves any trailing blanks, as long as the maximum size of the column is adequate. ESQL/C does not, however, add trailing blanks. If the column size is less than the length of the host-variable value, ESQL/C truncates the value.

Although **char**, **varchar**, **lvarchar**, and **string** host variables contain null terminators, 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.

Figure 3-7 shows the conversion of VARCHAR data when an application inserts it from host variables of **char**, **fixchar**, **lvvarchar**, and **string** character data types.

Figure 3-7
Converting ESQL/C Character Data Types to the VARCHAR Data Type

Source	Destination	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).
lvvarchar	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.

GLS

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 [Informix Guide to GLS Functionality](#). ♦

Figure 3-8 shows examples of conversions from character host variables to VARCHAR column data that ESQL/C might perform during an insert. In this figure, a plus (+) symbol represents a space character.

Figure 3-8
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 *Iv*varchar Data

When an application uses an **lvvarchar** host variable to fetch or insert a data value, ESQL/C must ensure that the value fits into the host variable or database column.

Fetching Ivvarchar Data

An application can fetch data from a database column of type LVARCHAR into a character (**char**, **string**, **fixchar**, **varchar**, or **lvvarchar**) host variable. If the column data does not fit into the host variable, ESQL/C truncates the data. To notify the user of the truncation, 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 1-42](#).

Inserting Ivvarchar Data

An application can insert data from a character host variable (**char**, **string**, **fixchar**, **varchar**, or **lvvarchar**) into a database column of type LVARCHAR. When an application inserts a character value into an LVARCHAR database column and the column size is less than the length of the host-variable value, ESQL/C truncates the value.

If the host variable you use for the insert is a **char** or **varchar**, the database server casts the type to **lvvarchar**.

When you write data to an LVARCHAR column, the database server imposes a limit of 2 kilobytes on the column. If the host variable is an **lvvarchar** data type and the data exceeds 2 kilobytes, the database server returns an error. If the column has an input support function, it must consume any data beyond 2 kilobytes, if necessary, to prevent the database server from returning the error. ♦

ANSI

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** ♦

IUS

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;
:
:
strcpy("Dexter", first);
strcpy("Haven", last);
EXEC SQL insert into customer (fname, lname)
      values (:first, :last);
```

The **strcpy()** 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 ESQ/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 on the **ifx_var()** functions, see [“The lvarchar pointer and var binary Library Functions” on page 10-32](#).

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.

Function Name	Description
bycmp()	Compares two groups of contiguous bytes
bycopy()	Copies bytes from one area to another
byfill()	Fills an area you specify with a character
byleng()	Counts the number of bytes in a string
ldchar()	Copies a fixed-length string to a null-terminated string
rdownshift()	Converts all letters to lowercase
rstod()	Converts a string to a double value
rstoi()	Converts a string to a short integer value
rstol()	Converts a string to a long integer value

(1 of 2)

Function Name	Description
strupshift()	Converts all letters to uppercase
strcat()	Concatenates one string to another
stchar()	Copies a null-terminated string to a fixed-length string
stcmpwr()	Compares two strings
stcopy()	Copies one string to another string
stleng()	Counts the number of bytes in a string

(2 of 2)

When you compile your ESQL/C program with the **esql** preprocessor, it calls on the linker to link these functions to your program. The following pages describe each of these functions.

bycmprr()

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

Syntax

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

- byte1* is a pointer to the location at which the first group of contiguous bytes starts.
- byte2* is a pointer to the location at which the second group of contiguous bytes starts.
- length* is the number of bytes that you want **bycmprr()** to compare.

Usage

The **bycmprr()** 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 **bycmprr()** function returns an integer whose value (0, -1, or +1) indicates the result of the comparison between the two groups of bytes.

The **bycmprr()** 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.

Example

This sample program is in the **bycmp.ec** file in the **demo** directory.

```

/*
 * bycmp.ec *

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

#include <stdio.h>

main()
{
    int x;

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

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

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

    printf("BYCMPR Sample ESQL Program running.\n\n");

    /* strings */
    printf("Comparing strings: String 1=%s\tString 2=%s\n", string1, string2);
    printf("  Executing: bycmp(string1, string2, sizeof(string1))\n");
    x = bycmp(string1, string2, sizeof(string1));
    printf("  Result = %d\n", x);

    /* ints */
    printf("Comparing numbers: Number 1=%d\tNumber 2=%d\n", number1, number2);
    printf("  Executing: bycmp( (char *) &number1, (char *) &number2, ");
    printf("sizeof(number1))\n");
    x = bycmp( (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: bycmp(string3, string4, sizeof(string3))\n");
    x = bycmp(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: bycmp(&string1[2], &string2[2], 2)\n");
    x = bycmp(&string1[2], &string2[2], 2);
    printf("  Result = %d\n", x);

    printf("\nBYCMPR Sample ESQL Program over.\n\n");
}

```

Example Output

BYCMPR Sample ESQL Program running.

Comparing strings: String1=abcdef String 2=abcdeg
Executing: bycmp(string1, string2, sizeof(string1))
Result = -1

Comparing numbers: Number 1=12345 Number 2=12367
Executing: bycmp((char *) &number1, (char *) &number2, sizeof(number1))
Result = -1

Comparing strings with non-printable characters:
Octal string 1=40300 Octal string 2=40310
Executing: bycmp(string3, string4, sizeof(string3))
Result = 1

Comparing bytes: Byte string 1=cd Byte string 2=cd
Executing: bycmp(&string1[2], &string2[2], 2)
Result = 0

BYCMPR Sample ESQL Program over.

bycopy()

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

Syntax

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

from is a pointer to the first byte of the group of bytes that you want **bycopy()** to copy.

to is 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, ESQL/C does not preserve the value to which *from* points.

length is the number of bytes that you want **bycopy()** to copy.



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

Example

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()
{
    int number1 = 12345;
    int 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");
}

```

Example 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.

```

byfill()

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

Syntax

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

to is a pointer to the first byte of the memory area that you want **byfill()** to fill.

length is the number of times that you want **byfill()** to repeat the character within the area.

ch is 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.

Example

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");
}

```

Example Output

```

BYFILL Sample ESQL Program running.

String = xxxxxxxxxxxxxxxxxxxx

Filling string with five 's' characters:
Result = sssssxxxxxxxxxxxxxx

Filling string with two 's' characters starting at position 16:
Result = sssssxxxxxxxxxxssx

Filling entire string with 'b' characters:
Result = bbbbbbbbbbbbbbbbbbb

BYFILL Sample Program over.

```

byleng()

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

Syntax

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

from is a pointer to a fixed-length string (not null-terminated).

count is the number of bytes in the fixed-length string. This does not include trailing blanks.

Example

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

    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");
}
```

Example Output

```
BYLENG Sample Program running.

Initial string:
  Length = 10, String = 'xxxxxxxxx          '

After copying two 's' characters starting at position 16:
  Length = 18, String = 'xxxxxxxxx          ss '

BYLENG Sample Program over.
```

ldchar()

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;  
    int count;  
    char *to;
```

from is a pointer to the fixed-length source string.

count is the number of bytes in the fixed-length source string.

to is 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\tDestination: [%s]\n", src2, dest);

    printf("\nLDCHAR Sample Program over.\n\n");
}

```

Example Output

```
LDCHAR Sample ESQL Program running.
```

```
Source: [abcd  ]
Destination: [abcd]
```

```
Source: [abcd g ]
Destination: [abcd g]
```

```
LDCHAR Sample Program over.
```

rdownshift()

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

Syntax

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

s is 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 [*Informix Guide to GLS Functionality*](#). ♦

GLS

Example

This sample program is in the **rdownshift.ec** file in the **demo** directory.

```
/*
 * rdownshift.ec *

The following program uses rdownshift() on a string containing alpha,
numeric and punctuation characters.
*/

#include <stdio.h>

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

    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");
}
```

Example Output

```
RDOWNSHIFT Sample ESQL Program running.

    Input string...: [123ABCDEFGHGIJK'.;]
    After downshift: [123abcdefghijklmnopqrstuvwxyz]

RDOWNSHIFT Sample Program over.
```

rstod()

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

Syntax

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

string is a pointer to a null-terminated string.

double_val is a pointer to a **double** value that holds the converted value.

Return Codes

=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()
{
    int 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");
}

```

Example 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.

```

rstoi()

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

Syntax

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

string is a pointer to a null-terminated string.

ival is a pointer to an **integer** value that holds the converted value.

Usage

The legal range of values is from -32767 to 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()
{
    int err;
    int 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;
        if (risnull(CSHORTTYPE, (char *) &s)) /* assign to a SHORT variable */
            printf("\tResult = NULL\n\n"); /* and then test for NULL */
        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");
}

```

Example 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.

```

rstol()

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

Syntax

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

string is a pointer to a null-terminated string.

long_int is a pointer to a **long integer** value that holds the converted value.

Usage

The legal range of values is from -2,147,483,647 to 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()
```

```

{
    int err;
    long l;

    printf("RSTOL Sample ESQL Program running.\n\n");

    l = 0;
    printf("Converting String 'abc':\n");
    if((err = rstrtol("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 = rstrtol("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 = rstrtol("", &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");
}

```

Example 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.

```

rupshift()

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

Syntax

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

s is 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 [Informix Guide to GLS Functionality](#). ♦

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[] = "123abcdefghijklmnopqrstuvwxyz.";

    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");
}

```

Example Output

```

RUPSHIFT Sample ESQL Program running.

    Input  string: 123abcdefghijklmnopqrstuvwxyz.
    After upshift: 123ABCDEFGHIJKLMNOPQRSTUVWXYZ.

RUPSHIFT Sample Program over.

```

stcat()

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

Syntax

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

s is a pointer to the start of the string that **stcat()** places at the end of the destination string.

dest is 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");
}

```

Example 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.

```

stchar()

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

Syntax

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

from is a pointer to the first byte of a null-terminated source string.

to is 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, ESQL/C discards the value to which *from* points.

count is 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");
}

```

Example Output

```
STCHAR Sample ESQL Program running.
```

```
Source string: [start]
```

```
Destination string before stchar: [123abcdefghijkl;.]
```

```
Destination string  after stchar: [start
```

```
]

```

```
STCHAR Sample Program over.
```

strcmp()

The **strcmp()** function compares two null-terminated strings.

Syntax

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

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

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



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("STCMR 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("\nSTCMR Sample Program over.\n\n");
}
```

Example Output

```
STCMR Sample ESQL Program running.

Executing: stcmpr("aaa", "aaa")
  Result = 0
Executing: stcmpr("aaa", "aaaa")
  Result = -1
Executing: stcmpr("bbb", "aaaa")
  Result = 1

STCMR Sample Program over.
```

strcpy()

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

Syntax

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

from is a pointer to the null-terminated string that you want **strcpy()** to copy.

to is a pointer to a location in memory where **strcpy()** copies the string.

Example

This sample program is in the **strcpy.ec** file in the **demo** directory.

```
/*
 * strcpy.ec *
 */
This program displays the result of copying a string using strcpy().
#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 */
    strcpy("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");
}
```


Example Output

```
STCOPY Sample ESQL Program running.  
  
Initial string:  
[abcdefghijklmnopqrstuvwxyz]  
After copy of 'John Doe' to position 15:  
[abcdefghijklmnopqrstuvwxyzJohn Doe]  
  
STCOPY Sample Program over.
```

stleng()

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

Syntax

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

string is 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[];
{
    int 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");
}

```

Example 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.
```

Working with the DECIMAL Data Type

The SQL DECIMAL Data Type	4-3
The decimal Data Type	4-4
Implicit Data Conversion for decimal Values	4-7
Operations on decimal Values.	4-7
Data Conversion for decimal Values	4-7
The decimal Library Functions	4-8
decadd()	4-9
deccmp()	4-11
deccopy()	4-13
deccvasc()	4-15
deccvdbl()	4-18
deccvint()	4-20
deccvlong()	4-22
decdiv()	4-24
dececv() and decfcvt()	4-26
decmul()	4-32
decround()	4-34
decsub()	4-37
dectoasc()	4-39
dectodbl()	4-42
dectoint()	4-44
dectolong()	4-47
dectrunc()	4-49

T

his chapter explains how to use the **decimal** data type in an INFORMIX-ESQL/C program. It contains the following information:

- A description of the **decimal** data type
- Descriptions of the ESQL/C library functions that you can use to manipulate **decimal** data types and examples of their use

For information about the other numeric data types that ESQL/C supports, see [Chapter 5, “Working with Numeric Data Types.”](#) For information about all the data types available for use in an ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#) For information about SQL data types, see the *Informix Guide to SQL: Reference*.

The SQL DECIMAL Data Type

ESQL/C supports the SQL DECIMAL data type 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* ($\leq p$) digits to the right of the decimal point (the scale).

For a complete description of the DECIMAL data type, see the [Informix Guide to SQL: Reference](#).

The decimal Data Type

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.) Figure 4-1 shows the four parts of the **decimal** structure.

Figure 4-1
Fields in the decimal Structure

Field	Description
dec_exp	The <i>exponent</i> 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 <i>left</i> to position the decimal point (or as a power of 100 for the number of base-100 numbers).
dec_pos	The <i>sign</i> 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 <i>number of digit pairs</i> (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] .

Figure 4-2 shows some sample **decimal** values.

Figure 4-2
Sample decimal Values

Value	decimal Structure Field Values			
	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 ESQL/C stores **decimal** numbers.

Implicit Data Conversion for decimal Values

All operations on **decimal** type numbers take place through the functions that the ESQL/C library provides. Any other operations, modifications, or analyses can produce unpredictable results. ESQL/C provides functions that facilitate conversion of **decimal** data type numbers to and from each C-language data type. The following section describes these functions.

Operations on decimal Values

The ESQL/C **decimal** data type uses its own internal structure to store decimal values. (For more information, see [“The decimal Data Type” on page 4-4](#).) Use the following ESQL/C library functions to perform all operations on **decimal** type numbers.

decadd()	deccopy()	decmul()	decsb()
deccmp()	decdiv()	decround()	dectrunc()

Any other operations, modifications, or comparisons can produce unpredictable results. For more information on how to use these functions, refer to [“The decimal Library Functions” on page 4-8](#).

Data Conversion for decimal Values

ESQL/C provides the following library functions to support conversion of **decimal** values to and from C language data types.

deccvasc()	deccvlong()	dectoasc()	dectoint()
deccvdbl()	dececv()	dectodbl()	dectolong()
deccvint()	decfcvt()		

For more information on how to use these functions, refer to [“The decimal Library Functions” on page 4-8](#).

You can control the number of decimal digits that ESQL/C places in character host variables with an internal ESQL/C global variable that is called **dbfltprec**, or with the **DBFLTMASK** environment variable. For more information, see [“Operations That Involve a Decimal Value” on page 2-20](#).

The decimal Library Functions

The following ESQL/C library functions are available to handle **decimal** data type numbers.

Function Name	Description
decadd()	Adds two decimal numbers
deccmp()	Compares two decimal numbers
deccopy()	Copies a decimal number
deccvasc()	Converts a C char type value to a decimal type value
deccvdbl()	Converts a C double type value to a decimal type value
deccvint()	Converts a C int type value to a decimal type value
deccvlong()	Converts a C long type value to a decimal type value
decdiv()	Divides two decimal numbers
dececv()	Converts a decimal value to an ASCII string
decfcv()	Converts a decimal value to an ASCII string
decmul()	Multiplies two decimal numbers
decround()	Rounds a decimal number
decsub()	Subtracts two decimal numbers
dectoasc()	Converts a decimal type value to an ASCII string
dectodbl()	Converts a decimal type value to a C double type value
dectoint()	Converts a decimal type value to a C int type value
dectolong()	Converts a decimal type value to a C long type value
dectrunc()	Truncates a decimal number

When you compile your ESQL/C program with the **esql** preprocessor, it calls on the link editor to link the library that contains these functions to your program.

Chapter 5, “Working with Numeric Data Types,” describes the function **rfmtdec()**, which allows you to format a decimal number. The following pages describe the functions in the preceding table.

decadd()

The **decadd()** function adds two **decimal** type values.

Syntax

```
int decadd(n1, n2, sum)
        dec_t *n1;
        dec_t *n2;
        dec_t *sum;
```

n1 is a pointer to the **decimal** structure of the first operand.

n2 is a pointer to the **decimal** structure of the second operand.

sum is 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.

Example

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];
```

decadd()

```
main()
{
    int 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);
}
```

Example Output

DECADD Sample ESQL Program running.

1000.6789 + 80 = 1080.6789

DECADD Sample Program over.

deccmp()

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

Syntax

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

- n1* is a pointer to a **decimal** structure of the first number to compare.
- n2* is 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.

Example

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()
{
```

```

int 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);
}

```

Example 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.

```


deccopy()

The **deccopy()** function copies one **decimal** structure to another.

Syntax

```
void deccopy(source, target)
           dec_t *source;
           dec_t *target;
```

source is a pointer to the value held in the source **decimal** structure.

target is 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.

Example

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()

```
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);
}
```

Example Output

```
DECCOPY Sample ESQL Program running.

String = 12345.6789
Executing: deccopy(&num1, &num2)
Destination = 12345.6789

DECCOPY Sample Program over.
```

deccvasc()

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

Syntax

```
int deccvasc(strng_val, len, dec_val)
    char *strng_val;
    int len;
    dec_t *dec_val;
```

strng_val is a pointer to a string whose value **deccvasc()** converts to a **decimal** value.

len is the length of the *strng_val* string.

dec_val is 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.

Example

The file **deccvasc.ec** 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()
{
    int 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;
{
    int 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");
}
```

Example 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.

deccvdbl()

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

Syntax

```
int deccvdbl(dbl_val, np)
    double dbl_val;
    dec_t *dec_val;
```

dbl_val is the **double** value that **deccvdbl()** converts to a **decimal** type value.

dec_val is 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.

Example

The file **deccvdbl.ec** 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()
{
    int 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);
}

```

Example 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.

```

deccvint()

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

Syntax

```
int deccvint(int_val, dec_val)
    int int_val;
    dec_t *dec_val;
```

int_val is the **int** value that **deccvint()** converts to a **decimal** type value.

dec_val is 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.

Example

The file **deccvint.ec** 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()
{
    int 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);
}

```

Example 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.

```

deccvlong()

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

Syntax

```
int deccvlong(lng_val, dec_val)
            long lng_val;
            dec_t *dec_val;
```

lng_val is the **long** value that **deccvlong()** converts to a **decimal** type value.

dec_val is 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.

Example

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()
{
    int x;
    dec_t num;
    long 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 = %ld\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);
}

```

Example 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.

```

decdiv()

The **decdiv()** function divides two **decimal** type values.

Syntax

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

n1 is a pointer to the **decimal** structure of the first operand.

n2 is a pointer to the **decimal** structure of the second operand.

quotient is 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.

Example

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()
{
    int 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);
}

```

Example Output

DECDIV Sample ESQL Program running.

480 / 80 = 6.0

DECDIV Sample Program over.

dececvrt() and decfcvt()

The **dececvrt()** and **decfcvt()** functions are analogous to the subroutines under ECVT(3) in section three of the *UNIX Programmer's Manual*. The **dececvrt()** 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 *dececvrt(dec_val, ndigit, decpt, sign)
    dec_t *dec_val;
    int ndigit;
    int *decpt;
    int *sign;
```

```
char *decfcvt(dec_val, ndigit, decpt, sign)
    dec_t *dec_val;
    int ndigit;
    int *decpt;
    int *sign;
```

- dec_val** is a pointer to a **decimal** structure that contains the **decimal** value you want these functions to convert.
- ndigit** is the length of the ASCII string for **dececvrt()**. It is the number of digits to the right of the decimal point for **decfcvt()**.
- decpt** is a pointer to an integer that is the position of the decimal point relative to the start of the string. A negative value for **decpt* means to the left of the returned digits.
- sign** is 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 **dececvrt()** 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 **dececvrt()** function rounds low-order digits.

The **decfcvt()** function is identical to **dececvrt()**, 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 **dececv()** 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 **dececv.ec** demo program on [page 4-29](#).



Warning: When you write thread-safe ESQL/C applications, do not use the **dececv()** or **decfcv()** library functions. Instead, use their thread-safe equivalents, **ifx_dececv()** and **ifx_decfcv()**. For more information, see [Chapter 13, “Using Informix Libraries.”](#)

dececv() Example

The file **dececv.ec** in the **demo** directory contains the following sample program.

```
/*
 * dececv.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
};
```

dececvrt() and decfcvt()

```
char result[40];

main()
{
    int x;
    int i = 0, f, sign;
    dec_t num;
    char *dp, *dececvrt();

    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 = dececvrt(&num, 20, &f, &sign);    /* to 20-char ASCII string */
        printf(" Output of dececvrt(&num, 20, ...): %c%s decpt: %d sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvrt(&num, 10, &f, &sign);    /* to 10-char ASCII string */
        /* display result */
        printf(" Output of dececvrt(&num, 10, ...): %c%s decpt: %d sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvrt(&num, 4, &f, &sign);    /* to 4-char ASCII string */
        /* display result */
        printf(" Output of dececvrt(&num, 4, ...): %c%s decpt: %d sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvrt(&num, 3, &f, &sign);    /* to 3-char ASCII string */
        /* display result */
        printf(" Output of dececvrt(&num, 3, ...): %c%s decpt: %d sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        dp = dececvrt(&num, 1, &f, &sign);    /* to 1-char ASCII string */
        /* display result */
        printf(" Output of dececvrt(&num, 1, ...): %c%s decpt: %d sign: %d\n",
            (sign ? '-' : '+'), dp, f, sign);

        ++i;                                /* next string */
    }

    printf("\nDECECVT Sample Program over.\n\n");
}
```


dececvf() Example Output

DECECVT Sample ESQL Program running.

```
Input string[0]: 210203.204
Output of dececvf: +2102 decpt: 6 sign: 0
Output of dececvf: +2102032040 decpt: 6 sign: 0
Output of dececvf: +2 decpt: 6 sign: 0
Output of dececvf: +210 decpt: 6 sign: 0

Input string[1]: 4894
Output of dececvf: +4894 decpt: 4 sign: 0
Output of dececvf: +4894000000 decpt: 4 sign: 0
Output of dececvf: +5 decpt: 4 sign: 0
Output of dececvf: +489 decpt: 4 sign: 0

Input string[2]: 443.334899312
Output of dececvf: +4433 decpt: 3 sign: 0
Output of dececvf: +4433348993 decpt: 3 sign: 0
Output of dececvf: +4 decpt: 3 sign: 0
Output of dececvf: +443 decpt: 3 sign: 0

Input string[3]: -12344455
Output of dececvf: -1234 decpt: 8 sign: 1
Output of dececvf: -1234445500 decpt: 8 sign: 1
Output of dececvf: -1 decpt: 8 sign: 1
Output of dececvf: -123 decpt: 8 sign: 1

Input string[4]: 12345.67
Output of dececvf: +1235 decpt: 5 sign: 0
Output of dececvf: +1234567000 decpt: 5 sign: 0
Output of dececvf: +1 decpt: 5 sign: 0
Output of dececvf: +123 decpt: 5 sign: 0

Input string[5]: .001234
Output of dececvf: +1234 decpt: -2 sign: 0
Output of dececvf: +1234000000 decpt: -2 sign: 0
Output of dececvf: +1 decpt: -2 sign: 0
Output of dececvf: +123 decpt: -2 sign: 0

DECECVT Sample Program over.
```

decfcvt() Example

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()
{
    int x;
    dec_t num;
    int 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");
}
```

decfcvt() Example Output

```
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.
```

decmul()

The **decmul()** function multiplies two **decimal** type values.

Syntax

```
int decmul(n1, n2, product)
    dec_t *n1;
    dec_t *n2;
    dec_t *product;
```

n1 is a pointer to the **decimal** structure of the first operand.

n2 is a pointer to the **decimal** structure of the second operand.

product is 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.

Example

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()
{
    int 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);
}

```

Example Output

DECMUL Sample ESQL Program running.

80.2 * 6.0 = 481.2

DECMUL Sample Program over.

decround()

The **decround()** function rounds a **decimal** type number to fractional digits.

Syntax

```
void decround(d, s)
    dec_t *d;
    int s;
```

- d* is a pointer to a **decimal** structure whose value the **decround()** function rounds.
- s* is the number of fractional digits to which **decround()** rounds *d*. Use a positive number for the *d* 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 <i>s</i>	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

Example

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()
{
    int x;
    int 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");
}

```

decround()

Example 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.
```


decsub()

The **decsub()** function subtracts two **decimal** type values.

Syntax

```
int decsub(n1, n2, difference)
    dec_t *n1;
    dec_t *n2;
    dec_t *difference;
```

n1 is a pointer to the **decimal** structure of the first operand.

n2 is a pointer to the **decimal** structure of the second operand.

difference is 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.

Example

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()
{
    int 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);
}
```

Example Output

```
DECSUB Sample ESQL Program running.
```

```
    1000.038782 - 480 = 520.038782
```

```
DECSUB Sample Program over.
```

dectoasc()

The **dectoasc()** function converts a **decimal** type number to a C **char** type value.

Syntax

```
int dectoasc(dec_val, strng_val, len, right)
    dec_t *dec_val;
    char *strng_val;
    int len;
    int right;
```

- dec_val* is a pointer to the **decimal** structure whose value **dectoasc()** converts to a text string.
- strng_val* is a pointer to the first byte of the character buffer where the **dectoasc()** function places the text string.
- len* is the size of *strng_val*, in bytes, minus 1 for the null terminator.
- right* is an integer that indicates the number of decimal places to the right of the decimal point.

Usage

If *right* = -1, the decimal value of **np* 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.

Example

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()
{
    int 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")
}

```

Example 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.

dectodbl()

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

Syntax

```
int dectodbl(dec_val, dbl_val)
           dec_t *dec_val;
           double *dbl_val;
```

dec_val is a pointer to a **decimal** structure whose value **dectodbl()** converts to a **double** type value.

dbl_val is 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.

Example

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()
{
    int 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);
}

```

Example 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.

dectoint()

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

Syntax

```
int dectoint(dec_val, int_val)
           dec_t *dec_val;
           int *int_val;
```

dec_val is a pointer to a **decimal** structure whose value **dectoint()** converts to an **int** type value.

int_val is a pointer to an **int** 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.

Example

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()
{
    int x;
    int 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);
}

```

dectoint()

Example 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.
```

dectolong()

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

Syntax

```
int dectolong(dec_val, lng_val)
           dec_t *dec_val;
           long  *lng_val;
```

dec_val is a pointer to a **decimal** structure whose value **dectolong()** converts to a **long** integer.

lng_val is a pointer to a **long** 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.

Example

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);
}

```

Example 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.

```

detrunc()

The **detrunc()** function truncates a rounded **decimal** type number to fractional digits.

Syntax

```
void detrunc(d, s)
    dec_t *d;
    int s;
```

- d* is a pointer to a **decimal** structure for a rounded number whose value **detrunc()** truncates.
- s* is the number of fractional digits to which **detrunc()** truncates the number. Use a positive number or zero for this argument.

Usage

The following table shows the sample output from **detrunc()** with various inputs.

Value Before Truncation	Value of <i>s</i>	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

Example

The file **detrunc.ec** in the **demo** directory contains the following sample program.

```

/*
 * detrunc.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()
{
    int x;
    int i = 6;          /* number of decimal places to start with */
    dec_t num1;

    printf("DETRUNC 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;
        }
        detrunc(&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("\nDETRUNC Sample Program over.\n\n");
}

```

Example Output

```
DETRUNC 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
```

```
DETRUNC Sample Program over.
```


Working with Numeric Data Types

The Integer Data Types	5-4
The Floating-Point Data Types	5-5
Declaring float Host Variables.	5-5
Implicit Data Conversion	5-6
The INT8 and SERIAL8 SQL Data Types	5-7
The int8 Data Type	5-7
Implicit Data Conversion for int8 Values	5-8
Operations on int8 Values	5-8
Data Conversion for int8 Values	5-8
The int8 Library Functions	5-9
ifx_getserial8()	5-10
ifx_int8add().	5-12
ifx_int8cmp()	5-13
ifx_int8copy()	5-14
ifx_int8cvasc()	5-15
ifx_int8cvdbl()	5-16
ifx_int8cvdec()	5-17
ifx_int8cvint()	5-18
ifx_int8cvlong().	5-19
ifx_int8div()	5-20
ifx_int8mul().	5-21
ifx_int8sub().	5-22
ifx_int8toasc()	5-23
ifx_int8todbl()	5-24

ifx_int8todec()	5-25
ifx_int8toint()	5-26
ifx_int8tolong()	5-27
Formatting Numeric Strings.	5-28
Numeric-Formatting Functions.	5-34
rfmtdec()	5-35
rfmtdouble()	5-38
rfmtlong()	5-41

Informix database servers support the following numeric data types:

- Integer data types: SMALLINT, INTEGER, SERIAL
- Integer data types: INT8, SERIAL8 ♦
- Floating-point data types: SMALLFLOAT and FLOAT
- Fixed-point data types: DECIMAL and MONEY

This chapter describes the following features that INFORMIX-ESQL/C provides to work with numeric data types:

- How to declare host variables for the integer data types
- How to declare host variables for the floating-point data types
- How to manipulate the INT8 and SERIAL8 data types ♦
- How to use numeric-formatting masks to format numeric values

For information about the fixed-point data types, see [Chapter 4, “Working with the DECIMAL Data Type.”](#) For information about all the data types available for use in an ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#) For information about SQL data types, see the *Informix Guide to SQL: Reference*.

IUS

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, SERIAL8	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. A 32-bit platform could implement a **long int** with a different size than 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.

IUS

For more information on the INT8 or SERIAL8 data type, see [page 5-7](#). ♦

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 8 significant digits
FLOAT, DOUBLE PRECISION	double	Double-precision values with up to 16 significant digits
DECIMAL(<i>p</i>)	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 on the DECIMAL data type, see [Chapter 4, “Working with the DECIMAL Data Type.”](#)

Implicit Data Conversion

When an ESQL/C program fetches a floating-point column value (data type of DECIMAL(*n*), FLOAT, or SMALLFLOAT) 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, ESQL/C rounds the number to the precision that the host variable can hold.

In the following code fragment, an 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, 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, ESQL/C rounds the FLOAT value to fit into **prncpl_strng** and the code fragment produces the following output:

```
Value of principal=1234.8764
```

ESQL/C assumes a precision of 16 decimal digits for DECIMAL(*n*), FLOAT, or SMALLFLOAT values. (For DECIMAL(*n,m*), ESQL/C assumes *m* decimal digits.)

The INT8 and SERIAL8 SQL Data Types

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 SQL data types INT8 and SERIAL8, see the [Informix Guide to SQL: Reference](#). This section describes how to manipulate the ESQL/C data type, **int8**.

The int8 Data Type

Use the **int8** data type to declare host variables for database values of type INT8 and SERIAL8. Figure 5-1 shows the fields of the structure **ifx_int8_t**, which represents an INT8 or SERIAL8 value.

Figure 5-1
Fields of the ifx_int8_t Structure

Field Name	Field Type	Purpose
data	<i>unsigned long</i> [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 4-byte unsigned integers. <i>The actual data type of an unsigned 4-byte integer can be machine specific.</i>
sign	<i>short integer</i>	A short integer to hold the sign (NULL, negative, or positive) of the 8-byte integer. <i>The actual data type of a 2-byte integer can be machine specific.</i>

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;
```

Implicit Data Conversion for int8 Values

All operations on **int8** type numbers take place through the functions that the ESQL/C library provides. Any other operations, modifications, or analyses can produce unpredictable results. ESQL/C provides functions that facilitate conversion of **int8** data type numbers to and from each C-language data type. The following section describes these functions.

Operations on int8 Values

The ESQL/C **int8** data type uses its own internal structure to store large integer values. (For more information, see [“The int8 Data Type” on page 5-7.](#)) Use the following ESQL/C library functions to perform all operations on **int8** type numbers.

ifx_int8add()	ifx_int8copy()	ifx_int8mul
ifx_int8cmp()	ifx_int8div()	ifx_int8sub()

Any other operations, modifications, or comparisons can produce unpredictable results. For more information on how to use these functions, refer to [“The int8 Library Functions” on page 5-9.](#)

Data Conversion for int8 Values

ESQL/C provides the following library functions to support conversion of **int8** values to and from other data types.

ifx_int8cvasc()	ifx_int8cvint()	ifx_int8todec()
ifx_int8cvdbl()	ifx_int8cvlong()	ifx_int8toflt()
ifx_int8cvdec()	ifx_int8toasc()	ifx_int8toint()
ifx_int8cvflt()	ifx_int8todbl()	ifx_int8tolong()

For more information on how to use these functions, refer to [“The int8 Library Functions” on page 5-9.](#) ♦

The int8 Library Functions

The following ESQL/C library functions are available to handle **int8** data type numbers.

Function Name	Description
ifx_getserial8()	Returns an inserted SERIAL8 value
ifx_int8add()	Adds two int8 numbers
ifx_int8cmp()	Compares two int8 numbers
ifx_int8copy()	Copies an int8 number
ifx_int8cvasc()	Converts a C char type value to an int8 type value
ifx_int8cvdbl()	Converts a C double type value to an int8 type value
ifx_int8cvflt()	Converts a C float type value to an int8 type value
ifx_int8cvint()	Converts a C int type value to an int8 type value
ifx_int8cvlong()	Converts a C long type value to an int8 type value
ifx_int8div()	Divides two int8 numbers
ifx_int8mul()	Multiplies two int8 numbers
ifx_int8sub()	Subtracts two int8 numbers
ifx_int8toasc()	Converts an int8 type value to a text string
ifx_int8todbl()	Converts an int8 type value to a C double type value
ifx_int8toflt()	Converts an int8 type value to a C float type value
ifx_int8toint()	Converts an int8 type value to a C int type value
ifx_int8tolong()	Converts an int8 type value to a C long type value



ifx_getserial8()

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 is 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.

Example

```
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);
}
```



ifx_int8add()

The **ifx_int8add()** function adds two **int8** type values.

Syntax

```
int ifx_int8add(n1, n2, sum)
    ifx_int8_t *n1;
    ifx_int8_t *n2;
    ifx_int8_t *sum;
```

n1 is a pointer to the **int8** structure that contains the first operand.
n2 is a pointer to the **int8** structure that contains the second operand.
sum is 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. ♦

ifx_int8cmp()

The **ifx_int8cmp()** function compares two **int8** type numbers.

Syntax

```
int ifx_int8cmp(n1, n2)  
    ifx_int8_t *n1;  
    ifx_int8_t *n2;
```

- n1* is a pointer to the **int8** structure that contains the first number to compare.
- n2* is 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.
- INT8UNKNOWNW Either value is null. ♦

ifx_int8copy()

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 is a pointer to the **int8** structure that contains the source **int8** value to copy.

target is 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. ♦

IUS

ifx_int8cvasc()

The **ifx_int8cvasc()** function converts a value held as printable characters in a C **char** type into an **int8** type number.

Syntax

```
int ifx_int8cvasc(strng_val, len, int8_val)
    char *strng_val
    int len;
    ifx_int8_t *int8_val;
```

strng_val is a pointer to a string.

len is the length of the *strng_val* string.

int8_val is 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.

GLS

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 [Informix Guide to GLS Functionality](#). ♦

Return Codes

- 0 The conversion was successful.
- 1213 The string has non-numeric characters.
- 1284 The operation resulted in overflow or underflow. ♦

ifx_int8cvdbl()

The **ifx_int8cvdbl()** function converts a C **double** type number into an **int8** type number.

Syntax

```
int ifx_int8cvdbl(dbl_val, int8_val)  
    double dbl_val;  
    ifx_int8_t *int8_val;
```

dbl_val is the **double** value that **ifx_int8cvdbl()** converts to an **int8** type value.

int8_val is 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. ♦

ifx_int8cvdec()

The **ifx_int8cvdec()** function converts a **decimal** type value into an **int8** type value.

Syntax

```
int ifx_int8cvdec(dec_val, int8_val)  
    dec_t *dec_val;  
    ifx_int8_t *int8_val;
```

dec_val is a pointer to the **decimal** structure that **ifx_int8cvdec()** converts to an **int8** type value.

int8_val is 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. ♦

ifx_int8cvint()

The **ifx_int8cvint()** function converts a C **int** type number into an **int8** type number.

Syntax

```
int ifx_int8cvint(int_val, int8_val)  
    int int_val;  
    ifx_int8_t *int8_val;
```

int_val is the **int** value that **ifx_int8cvint()** converts to an **int8** type value.

int8_val is 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. ♦

ifx_int8cvlong()

The **ifx_int8cvlong()** function converts a C **long** type value into an **int8** type value.

Syntax

```
int ifx_int8cvlong(lng_val, int8_val)  
    long lng_val;  
    ifx_int8_t *int8_val;
```

lng_val is the **long** integer that **ifx_int8cvlong()** converts to an **int8** type value.

int8_val is 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. ♦

ifx_int8div()

The **ifx_int8div()** function divides two **int8** type values.

Syntax

```
int ifx_int8div(n1, n2, quotient)
    ifx_int8_t *n1;
    ifx_int8_t *n2;
    ifx_int8_t *quotient;
```

n1 is a pointer to the **int8** structure that contains the dividend.
n2 is a pointer to the **int8** structure that contains the divisor.
quotient is 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. ♦

ifx_int8mul()

The **ifx_int8mul()** function multiplies two **int8** type values.

Syntax

```
int ifx_int8mul(n1, n2, product)  
    ifx_int8_t *n1;  
    ifx_int8_t *n2;  
    ifx_int8_t *product;
```

- n1* is a pointer to the **int8** structure that contains the first operand.
n2 is a pointer to the **int8** structure that contains the second operand.
product is 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. ♦

ifx_int8sub()

The **ifx_int8sub()** function subtracts two **int8** type values.

Syntax

```
int ifx_int8sub(n1, n2, difference)  
    ifx_int8_t *n1;  
    ifx_int8_t *n2;  
    ifx_int8_t *difference;
```

- n1* is a pointer to the **int8** structure that contains the first operand.
- n2* is a pointer to the **int8** structure that contains the second operand.
- difference* is 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. ♦

IUS

ifx_int8toasc()

The **ifx_int8toasc()** function converts an **int8** type number to a C **char** type value.

Syntax

```
int ifx_int8toasc(int8_val, strng_val, len)
    ifx_int8_t *int8_val;
    char *strng_val;
    int len;
```

- int8_val* is a pointer to the **int8** structure whose value **ifx_int8toasc()** converts to a text string.
- strng_val* is a pointer to the first byte of the character buffer where the **ifx_int8toasc()** function places the text string.
- len* is 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 [Informix Guide to GLS Functionality](#). ♦

Return Codes

- 0 The conversion was successful.
- 1207 The converted value does not fit into the allocated space. ♦

GLS

ifx_int8todbl()

The **ifx_int8todbl()** function converts an **int8** type number into a C **double** type number.

Syntax

```
int ifx_int8todbl(int8_val, dbl_val)
    ifx_int8_t *int8_val;
    double *dbl_val;
```

int8_val is a pointer to the **int8** structure whose value **ifx_int8todbl()** converts to a **double** type value.

dbl_val is 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. ♦

ifx_int8todec()

The **ifx_int8todec()** function converts an **int8** type number into a **decimal** type number.

Syntax

```
int ifx_int8todec(int8_val, dec_val)
    ifx_int8_t *int8_val;
    dec_t *dec_val;
```

int8_val is a pointer to an **int8** structure whose value **ifx_int8todec()** converts to a **decimal** type value.

dec_val is 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. ♦

ifx_int8toint()

The **ifx_int8toint()** function converts an **int8** type number into a C **int** type number.

Syntax

```
int ifx_int8toint(int8_val, int_val)
    ifx_int8_t *int8_val;
    int *int_val;
```

int8_val is a pointer to an **int8** structure whose value **ifx_int8toint()** converts to an **int** type value.

int_val is a pointer to an **int** 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. ♦

ifx_int8tolong()

The **ifx_int8tolong()** function converts an **int8** type number into a C **long** type number.

Syntax

```
int ifx_int8tolong(int8_val, lng_val)  
    ifx_int8_t *int8_val;  
    long *lng_val;
```

int8_val is a pointer to an **int8** structure whose value **ifx_int8tolong()** converts to a **long** integer type value.

lng_val is a pointer to a **long** 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. ♦

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.

GLS

When you use **rfmtdec()**, **rfmtdouble()**, or **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, the numeric-formatting functions use 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 [Informix Guide to SQL: Reference](#)). For more information on locales, see the [Informix Guide to GLS Functionality](#). ♦

[Figure 5-2 on page 5-30](#) shows sample format strings for numeric expressions. The character **b** represents a blank or space.

Figure 5-2
Sample Format Patterns and Their Results

Formatting Mask	Numeric Value	Formatted Result
"#####"	0	bbbbbb
"&&&&&"	0	00000
"\$\$\$\$\$"	0	bbbb\$
"*****"	0	*****
"<<<<<"	0	(null string)
"##,###"	12345	12,345
"##,###"	1234	b1,234
"##,###"	123	bbb123
"##,###"	12	bbbb12
"##,###"	1	bbbbbb1
"##,###"	-1	bbbbbb1
"##,###"	0	bbbbbb
"&&, &&&"	12345	12,345
"&&, &&&"	1234	01,234
"&&, &&&"	123	000123
"&&, &&&"	12	000012
"&&, &&&"	1	000001
"&&, &&&"	-1	000001
"&&, &&&"	0	000000
"\$\$\$\$\$"	12345	***** (overflow)
"\$\$\$\$\$"	1234	\$1,234
"\$\$\$\$\$"	123	bb\$123
"\$\$\$\$\$"	12	bbb\$12
"\$\$\$\$\$"	1	bbbb\$1
"\$\$\$\$\$"	-1	bbbb\$1
"\$\$\$\$\$"	0	bbbbbb\$
"\$\$\$\$\$" (DBMONEY set to DM)	1234	DM1,234
"** * **"	12345	12,345
"** * **"	1234	*1,234
"** * **"	123	***123
"** * **"	12	*****12
"** * **"	1	*****1
"** * **"	0	*****

Formatting Mask	Numeric Value	Formatted Result
"##,###.##"	12345.67	12,345.67
"##,###.##"	1234.56	b1,234.56
"##,###.##"	123.45	bbb123.45
"##,###.##"	12.34	bbbb12.34
"##,###.##"	1.23	bbbbb1.23
"##,###.##"	0.12	bbbbbb.12
"##,###.##"	0.01	bbbbbb.01
"##,###.##"	-0.01	bbbbbb.01
"##,###.##"	-1	bbbbbb1.00
"&&, &&&. &&"	.67	000000.67
"&&, &&&. &&"	1234.56	01,234.56
"&&, &&&. &&"	123.45	000123.45
"&&, &&&. &&"	0.01	000000.01
"\$\$\$\$. \$"	12345.67	***** (overflow)
"\$\$\$\$. \$"	1234.56	\$1,234.56
"\$\$\$\$. ##"	0.00	bbbbbb\$.00
"\$\$\$\$. ##"	1234.00	\$1,234.00
"\$\$\$\$. &&"	0.00	bbbbbb\$.00
"\$\$\$\$. &&"	1234.00	\$1,234.00
"-##,###.##"	-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	bbbbbb-1.00
"-##,###.##"	12345.67	b12,345.67
"-##,###.##"	1234.56	bb1,234.56
"-##,###.##"	123.45	bbbb123.45
"-##,###.##"	12.34	bbbbbb12.34
"-##,###.##"	12.34	bbbbbb12.34
"---,###.##"	12.34	bbbbbb12.34
"---,###.##"	12.34	bbbbbb12.34

(2 of 4)

Formatting Numeric Strings

Formatting Mask	Numeric Value	Formatted Result
"---,---,###"	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	bbbbbb-\$1.12
"\$***,***,&&"	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(bbb\$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)

(3 of 4)

Formatting Mask	Numeric Value	Formatted Result
"(((,((\$.&&)"	-1.23	bbbbb(\$1.23)
"(((,((\$.&&)"	-.12	bbbbbb(\$.12)
"(\$\$\$,\$\$\$.&&)"	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	bbbbbb\$12.34
"((\$\$,\$\$\$.&&)"	1.23	bbbbbbb\$1.23
"(((,((\$.&&)"	12345.67	b\$12,345.67
"(((,((\$.&&)"	1234.56	bb\$1,234.56
"(((,((\$.&&)"	123.45	bbbb\$123.45
"(((,((\$.&&)"	12.34	bbbbbb\$12.34
"(((,((\$.&&)"	1.23	bbbbbbb\$1.23
"(((,((\$.&&)"	.12	bbbbbbbb\$.12
"<<<, <<<"	12345	12,345
"<<<, <<<"	1234	1,234
"<<<, <<<"	123	123
"<<<, <<<"	12	12

(4 of 4)

Numeric-Formatting Functions

INFORMIX-ESQL/C provides special functions that allow you to format numeric expressions. 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 ESQL/C library includes the following functions that support formatting masks for numeric values.

Function Name	Description
rfmtdec()	Converts a decimal value to a string
rfmtdouble()	Converts a double value to a string
rfmtlong()	Converts a long integer value to a string

When you compile your ESQL/C program with the **esql** command, the preprocessor automatically links these functions to your program.

This chapter describes the characters that you can use to create a formatting mask. It also provides extensive examples that show the results of applying these masks to numeric values.

rfmtdec()

The **rfmtdec()** function uses a formatting mask to convert a **decimal** value to a character string.

Syntax

```
int rfmtdec(dec_val, fmtstring, outbuf)
    dec_t *dec_val;
    char *fmtstring;
    char *outbuf;
```

- dec_val* is a pointer to the **decimal** value to format.
- fmtstring* is a pointer to a character buffer that contains the formatting mask to use for the *dec_val* value.
- outbuf* is 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 on these formatting characters, see [“Formatting Numeric Strings” on page 5-28](#).

GLS

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 [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 [Informix Guide to GLS Functionality](#). ♦

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()
{
    int x;
    int 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");
}

```

Example Output

RFMTDEC Sample ESQL Program running.

```

String = 210203.204
Format String = '**#####'
Format String = '$$$$$$.##'
Format String = '(&&&&&&.)'
Format String = '<<<<<<<<<'
Format String = '$*****.**'

Result = '**      210203'
Result = '      $210203.20'
Result = ' 000210,203.'
Result = '210,203'
Result = '$***210203.20'

String = 4894
Format String = '**#####'
Format String = '$$$$$$.##'
Format String = '(&&&&&&.)'
Format String = '<<<<<<<<<'
Format String = '$*****.**'

Result = ' **      4894'
Result = '      $4894.00'
Result = ' 000004,894.'
Result = '4,894'
Result = '$****4894.00'

String = 443.334899312
Format String = '**#####'
Format String = '$$$$$$.##'
Format String = '(&&&&&&.)'
Format String = '<<<<<<<<<'
Format String = '$*****.**'

Result = ' **      443'
Result = '      $443.33'
Result = ' 0000000443.'
Result = ' 443'
Result = '$*****443.33'

String = -12344455
Format String = '**#####'
Format String = '$$$$$$.##'
Format String = '(&&&&&&.)'
Format String = '<<<<<<<<<'
Format String = '$*****.**'

Result = ' **      12344455'
Result = '      $12344455.00'
Result = '(12,344,455.)'
Result = '12,344,455'
Result = '$*12344455.00'

RFMTDEC Sample Program over.

```

rfmtdouble()

The **rfmtdouble()** function uses a formatting mask to convert a **double** value to a character string.

Syntax

```
int rfmtdouble(dbl_val, fmtstring, outbuf)
    double dbl_val;
    char *fmtstring;
    char *outbuf;
```

dbl_val is the **double** number to format.

fmtstring is a pointer to a character buffer that contains the formatting mask for the value in *dbl_val*.

outbuf is 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 on these formatting characters, see [“Formatting Numeric Strings” on page 5-28](#).

GLS

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 [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 [Informix Guide to GLS Functionality](#). ♦

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**.

```

/* * rfmdtdouble.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()
{
 int x;
 int 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 = rfmdtdouble(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");
}

```

### Example Output

RFMTDOUBLE Sample ESQL Program running.

```
Double Number = 210203
Format String = '#####'
Format String = '<,<<,<<,<<'
Format String = '$$$$$$$.##'
Format String = '(&&,&&&,&&&.)'
Format String = '$*****.**'

Double Number = 4894
Format String = '#####'
Format String = '<,<<,<<,<<'
Format String = '$$$$$$$.##'
Format String = '(&&,&&&,&&&.)'
Format String = '$*****.**'

Double Number = 443.335
Format String = '#####'
Format String = '<,<<,<<,<<'
Format String = '$$$$$$$.##'
Format String = '(&&,&&&,&&&.)'
Format String = '$*****.**'

Double Number = -1.23445e+07
Format String = '#####'
Format String = '<,<<,<<,<<'
Format String = '$$$$$$$.##'
Format String = '(&&,&&&,&&&.)'
Format String = '$*****.**'
```

RFMTDOUBLE Sample Program over.



## rfmtlong()

The **rfmtlong()** function uses a formatting mask to convert a C **long** value to a character string.

### Syntax

```
int rfmtlong(lng_val, fmtstring, outbuf)
 long lng_val;
 char *fmtstring;
 char *outbuf;
```

*lng\_val* is the **long** integer that **rmtlong()** converts to character value.

*fmtstring* is a pointer to a character buffer that contains the formatting mask for the value in *lng\_val*.

*outbuf* is 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 on these formatting characters, see [“Formatting Numeric Strings” on page 5-28](#).

#### GLS

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 [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 [Informix Guide to GLS Functionality](#). ♦

## 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[] =
{
    21020304L,
    334899312L,
    -334899312L,
    -12344455L,
    0
};

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

char result[41];

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

    printf("RFMTLONG Sample ESQL Program running.\n\n");

    while(lngs[s])                /* for each long in lngs[] */
    {
        printf("Long Number = %ld\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 %ld 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");
}

```

Example Output

RFMTLONG ESQL Sample Program running.

Long Number = 21020304

```

Format String = '#####'
Format String = '$$$$$$$.##'
Format String = '(&,&&&,&&&,&&&.)'
Format String = '<<<<,<<,<<,<<'
Format String = '$*****.***'

```

```

Result = '          21020304'
Result = '      $21020304.00'
Result = '  00021,020,304. '
Result = '21,020,304'
Result = '$***21020304.00'

```

Long Number = 334899312

```

Format String = '#####'
Format String = '$$$$$$$.##'
Format String = '(&,&&&,&&&,&&&.)'
Format String = '<<<<,<<,<<,<<'
Format String = '$*****.***'

```

```

Result = '          334899312'
Result = '      $334899312.00'
Result = '  00334,899,312. '
Result = '334,899,312'
Result = '$***334899312.00'

```

Long Number = -334899312

```

Format String = '#####'
Format String = '$$$$$$$.##'
Format String = '(&,&&&,&&&,&&&.)'
Format String = '<<<<,<<,<<,<<'
Format String = '$*****.***'

```

```

Result = '          334899312'
Result = '      $334899312.00'
Result = ' (00334,899,312.)'
Result = '334,899,312'
Result = '$***334899312.00'

```

Long Number = -12344455

```

Format String = '#####'
Format String = '$$$$$$$.##'
Format String = '(&,&&&,&&&,&&&.)'
Format String = '<<<<,<<,<<,<<'
Format String = '$*****.***'

```

```

Result = '          12344455'
Result = '      $12344455.00'
Result = ' (00012,344,455.)'
Result = '12,344,455'
Result = '$***12344455.00'

```

RFMTLONG Sample Program over.

Working with Time Data Types

The SQL DATE Data Type	6-3
Formatting Date Strings	6-4
DATE Library Functions	6-5
rdatestr()	6-6
Return Codes	6-6
rdayofweek()	6-8
rdefmtdate().	6-10
rfmtdate().	6-14
rjulmdy()	6-17
rleapyear()	6-19
rmdyjul()	6-21
rstrdate()	6-23
rtoday()	6-26
The SQL DATETIME and INTERVAL Data Types	6-28
The datetime Data Type	6-29
The interval Data Type	6-30
Macros for datetime and interval Data Types	6-31
Fetching and Inserting DATETIME and INTERVAL Values.	6-33
Fetching and Inserting into datetime Host Variables.	6-33
Fetching and Inserting into interval Host Variables	6-34
Implicit Data Conversion	6-35
ANSI SQL Standards for DATETIME and INTERVAL Values	6-36
Operations on datetime and interval Values	6-36
Data Conversion for datetime and interval Values.	6-37

DATETIME and INTERVAL Library Functions	6-38
dtaddinv()	6-40
dtcurrent()	6-42
dtcvasc()	6-44
dtcvfmtasc()	6-47
dtextend()	6-51
dtsub()	6-54
dtsubinv()	6-57
dttoasc()	6-59
dttofmtasc()	6-62
incvasc()	6-65
incvfmtasc()	6-68
intoasc()	6-71
intofmtasc()	6-74
invdivdbl()	6-77
invdivinv()	6-80
invextend()	6-82
invmuldbl()	6-85

T

his chapter explains how to use **date**, **datetime**, and **interval** data types in an INFORMIX-ESQL/C program. It contains the following information:

- An overview of the ESQL/C **date** data type
- The syntax of the ESQL/C library functions that you can use to manipulate the **date** data type
- An overview of the ESQL/C **datetime** and **interval** data types and how to use them
- The syntax of 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 ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#) For information about SQL data types, see the *[Informix Guide to SQL: Reference](#)*.

The SQL DATE Data Type

ESQL/C supports the SQL DATE data type with the ESQL/C **date** data type for host variables. The **date** data type stores internal DATE values. It is implemented as a **long** integer, a four-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 *[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.

Format	Meaning
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 in the 1900s (00 through 99)
yyyy	Year as a four-digit number (0001 through 9999)

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 [Informix Guide to GLS Functionality](#).

When you use **rfmtdate** or **rdefmtdate()** to format DATE values, the function uses the date end-user formats that the **GL_DATE** or **DBDATE** 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 **DBDATE**, see the [Informix Guide to GLS Functionality](#). ♦

GLS

DATE Library Functions

The following date-manipulation functions are in the ESQL/C library. They convert dates between a string format and the internal DATE format.

Function Name	Description
rdatestr()	Converts an internal DATE to a character string format
rdayofweek()	Returns the day of the week of a date in internal format
rdefmtdate()	Converts a specified string format to an internal DATE
rfmtdate()	Converts an internal DATE to a specified string format
rjulmdy()	Returns month, day, and year from an internal DATE
rleapyear()	Determines whether specified year is a leap year
rmdyjul()	Returns an internal DATE from month, day, and year
rstrdate()	Converts a character string format to an internal DATE
rtoday()	Returns a system date as an internal DATE

When you compile your ESQL/C program with the **esql** command, **esql** automatically links these functions into your program. The following pages describe each of these functions.

rdatestr()

The **rdatestr()** function converts an internal DATE to a character string.

Syntax

```
int rdatestr(jdate, outbuf)
           long jdate;
           char *outbuf;
```

jdate is the internal representation of the date to format.

outbuf is 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 on **DBDATE**, refer to the [Informix Guide to SQL: Reference](#).
2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set). For more information on **GL_DATE**, refer to the [Informix Guide to GLS Functionality](#). ♦
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 [Informix Guide to GLS Functionality](#). ♦

Return Codes

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

GLS

GLS

- 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()
{
    int errnum;
    char today_date[20];
    long 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");
}

```

Example Output

```

RTODAY Sample ESQL Program running.

    Today's date is 10/26/1996.

RTODAY Sample Program over.

```

rdayofweek()

The **rdayofweek()** function returns the day of the week as an integer value for an internal DATE.

Syntax

```
int rdayofweek(jdate)  
    long jdate;
```

jdate is 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 obtains today's date from the system  
and determines what day of the week it is.  
*/  
  
#include <stdio.h>  
  
main()  
{  
    int errnum;  
    long 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");
}

```

Example Output

```

RDAYOFWEEK Sample ESQL Program running.

Enter a date as a single string, month.day.year
10.12.96

The date string is 10.12.96.
This date is a Saturday.

RDAYOFWEEK Sample Program over.

```

rdefmtdate()

The **rdefmtdate()** function uses a formatting mask to convert a character string to an internal DATE format.

Syntax

```
int rdefmtdate(jdate, fmtstring, inbuf)
    long *jdate;
    char *fmtstring;
    char *inbuf;
```

- jdate* is a pointer to a **long** integer value that receives the internal DATE value for the *inbuf* string.
- fmtstring* is a pointer to the buffer that contains the formatting mask to use the *inbuf* string.
- inbuf* is 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 on these date formats, see [“Formatting Date Strings” on page 6-4](#).

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. The following combinations of *fmtstring* and *input* are valid.

Formatting Mask	Input
mmddy	Dec. 25th, 1995
mmm. dd. yyyy	dec 25 1995
mmm. dd. yyyy	DEC-25-1995
mmm. dd. yyyy	122595

Formatting Mask	Input
mmm. dd. yyyy	12 / 25 / 95
yy / mm / dd	95 / 12 / 25
yy / mm / dd	1995, December 25th
yy / mm / dd	In the year 1995, the month of December, it is the 25th day
dd-mm-yy	This 25th day of December, 1995

(2 of 2)

When you use a two-digit year (yy) in a format, the **rdefmtdate()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, ESQL / C uses the 20th century. For information on how to set **DBCENTURY**, see the [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 [Informix Guide to GLS Functionality](#). ♦

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()
{
    int x;
    char date[20];
    long i_date;
    char *day_name;

    static char fmtstr[9] = "mmdyyyy";

    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");
}

```


Example 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.
```

rfmtdate()

The **rfmtdate()** function uses a formatting mask to convert an internal DATE format to a character string.

Syntax

```
int rfmtdate(jdate, fmtstring, outbuf)
    long jdate;
    char *fmtstring;
    char *outbuf;
```

- jdate* is the internal representation of a date to convert.
- fmtstring* is a pointer to the buffer that contains the formatting mask to use the *jdate* value.
- outbuf* is 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 on these date formats, see [“Formatting Date Strings” on page 6-4](#).

The examples in the following table use the formatting mask in *fmtstring* to convert the integer *jdate*, whose value corresponds to December 25, 1995, to a formatted string *outbuf*.

Formatting Mask	Formatted Result
"mmddy"	122595
"ddmmy"	251295
"ymmdd"	951225
"yy/mm/dd"	95/12/25

(1 of 2)

Formatting Mask	Formatted Result
"yy mm dd"	95 12 25
"yy-mm-dd"	95-12-25
"mmm. dd, yyyy"	Dec. 25, 1995
"mmm dd yyyy"	Dec 25 1995
"yyyy dd mm"	1995 25 12
"mmm dd yyyy"	Dec 25 1995
"ddd, mmm. dd, yyyy"	Mon, Dec. 25, 1995
"(ddd) mmm. dd, yyyy"	(Mon) Dec. 25, 1995

(2 of 2)

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 [Informix Guide to GLS Functionality](#). ♦

Return Codes

- 0 The conversion was successful.
- 1210 The internal date cannot be converted to month-day-year format.
- 1211 The program ran out of memory (memory-allocation error).

Example

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];
```

```

long i_date;
int x;
int errnum;
static short mdy_array[3] = { 12, 21, 1994 };

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-yy): %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");
}

```

Example Output

RFMTDATE Sample ESQL Program running.

Converted date (mm-dd-yy): 12-10-1994.

Converted date (mm.dd.yy): 12.10.94.

Converted date (mmm ddth, yyyy): Dec 10th, 1994

RFMTDATE Sample Program over.

rjulmdy()

The **rjulmdy()** function creates an array of three **short** integer values that represent the month, day, and year from an internal DATE value.

Syntax

```
int rjulmdy(jdate, mdy)
    long jdate;
    short mdy[3];
```

jdate is the internal representation of the date.

mdy is an array of **short** integers, where *mdy[0]* is the month (1 to 12), *mdy[1]* is the day (1 to 31), and *mdy[2]* is the year (1 to 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 converts today's date, represented as an
integer in internal format, to an array of three short integers that
contain the month, day and year.
*/

#include <stdio.h>

main()
{
    long i_date;
    short mdy_array[3];
    int errnum;
    char date[20];
```

```

int 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");
}

```

Example Output

```

RJULMDY Sample ESQL Program running.

Enter a date as a single string, month.day.year
10.12.94

The date string is 10.12.94.
The month component is: 10
The day component is: 12
The year component is: 1994

RJULMDY Sample Program over.

```

rleapyear()

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

```
int rleapyear(year)
int year;
```

year is an integer.

Usage

The argument *year* must be the year component of a date and not the date itself. You must express the *year* in full form (1996) and not abbreviated form (96).

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 obtains the system date into a long integer,
which stores this date in the internal format.
It then converts the internal format into an array of three short integers
that contain the month, day, and year portions of the date.
It then tests the year value to see if the year is a leap year.
*/

#include <stdio.h>

main()
{
    long i_date;
    int errnum;
```

```

short mdy_array[3];
char date[20];
int 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");
}

```

Example Output

```

RLEAPYEAR Sample ESQl Program running.

Enter a date as a single string, month.day.year
10.12.94

The date string is 10.12.94.
1994 is not a leap year

RLEAPYEAR Sample Program over.

```

rmdyjul()

The **rmdyjul()** function creates an internal DATE from an array of three **short** integer values that represent month, day, and year.

Syntax

```
int rmdyjul(mdy, jdate)
    short mdy[3];
    long *jdate;
```

mdy is an array of **short** integer values, where *mdy*[0] is the month (1 to 12), *mdy*[1] is the day (1 to 31), and *mdy*[2] is the year (1 to 9999).

jdate is 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 (1996) or abbreviated form (96).

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()
{
    long i_date;
    int errnum;
    static short mdy_array[3] = { 12, 21, 1994 };
    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");
}

```

Example Output

```

RMDYJUL Sample ESQL Program running.

Date 'Dec 21 1994' converted to internal format

RMDYJUL Sample Program over.

```

rstrdate()

The **rstrdate()** function converts a character string to an internal DATE.

Syntax

```
int rstrdate(inbuf, jdate)
    char *inbuf;
    long *jdate;
```

inbuf is a pointer to the string that contains the date to convert.

jdate is a pointer to a **long** 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 on **DBDATE**, refer to the [Informix Guide to SQL: Reference](#).
2. The format that the **GL_DATE** environment variable specifies (if **GL_DATE** is set). For more information on **GL_DATE**, refer to the [Informix Guide to GLS Functionality](#). ♦
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 (1995) or as two digits (95).

GLS

GLS

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 [Informix Guide to GLS Functionality](#). ♦

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 on how to set **DBCENTURY**, see the [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 "mmdyyyy" format to an internal date format.
 */

#include <stdio.h>

main()
{
    long i_date;
    int errnum;
    char str_date[15];

    printf("RSTRDATE Sample ESQL Program running.\n\n");

    /* Convert Sept. 6th, 1994 into i_date */
    if ((errnum = rstrdate("9.6.1994", &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");
}

```

Example Output

```
RSTRDATE Sample ESQL Program running.  
Date 'Sep 06 1994' converted to internal format  
RSTRDATE Sample Program over.
```

rtoday()

The **rtoday()** function returns the system date as an internal DATE value.

Syntax

```
void rtoday(today)
          long *today;
```

today is a pointer to a **long** integer 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()
{
    int errnum;
    char today_date[20];
    long 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");
}
```

Example Output

```
RTODAY Sample ESQL Program running.
```

```
    Today's date is 09/16/1996.
```

```
RTODAY Sample Program over.
```

The SQL DATETIME and INTERVAL Data Types

ESQL/C supports two data types that can hold information about time values:

- The **datetime** data type encodes an instant in time that is expressed as a calendar date and a time of day.
- The **interval** data type encodes a span of time.

Figure 6-1 summarizes these two time data types.

Figure 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 these 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 on the DATETIME and INTERVAL data types, refer to the [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 **dttime** structure and **dttime_t** typedef have two parts. Figure 6-2 lists these parts.

Figure 6-2
Fields in the dttime Structure

Field	Description
dt_qual	The qualifier of the datetime value.
dt_dec	The 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 on [page 6-31](#).

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. Figure 6-3 lists these parts.

Figure 6-3
Fields in the intrvl Structure

Field	Description
in_qual	The qualifier of the interval value.
in_dec	The 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 shown on [page 6-31](#).

Macros for datetime and interval Data Types

In addition to these data structures, the **datetime.h** file defines the names and macro functions that Figure 6-4 shows.

Figure 6-4
Qualifier Macros for datetime and interval Data Types

Name of Macro	Description
TU_YEAR	The time unit for the YEAR qualifier field
TU_MONTH	The time unit for the MONTH qualifier field
TU_DAY	The time unit for the DAY qualifier field
TU_HOUR	The time unit for the HOUR qualifier field
TU_MINUTE	The time unit for the MINUTE qualifier field

(1 of 2)

Name of Macro	Description
TU_SECOND	The time unit for the SECOND qualifier field
TU_FRAC	The time unit for the leading qualifier field of FRACTION
TU_Fn	The names for datetime ending fields of FRACTION(<i>n</i>), for <i>n</i> from 1 to 5
TU_START(<i>q</i>)	Returns the leading field number from qualifier <i>q</i>
TU_END(<i>q</i>)	Returns the trailing field number from qualifier <i>q</i>
TU_LEN(<i>q</i>)	Returns the length in digits of the qualifier <i>q</i>
TU_FLEN(<i>f</i>)	Returns the length in digits of the first field, <i>f</i> , of an interval qualifier
TU_ENCODE(<i>p,f,t</i>)	Creates a qualifier from the first field number <i>f</i> with precision <i>p</i> and trailing field number <i>t</i>
TU_DTENCODE(<i>f,t</i>)	Creates a datetime qualifier from the first field number <i>f</i> and trailing field number <i>t</i>
TU_IENCODE(<i>p,f,t</i>)	Creates an interval qualifier from the first field number <i>f</i> with precision <i>p</i> and trailing field number <i>t</i>

(2 of 2)

You need the macros in [Figure 6-4 on page 6-31](#) only when you work directly with qualifiers in binary form. 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. This is shown in the following example:

```

/* 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);

```

In the previous example, the **interval** variable gets a **day to second** qualifier. The precision of the largest field in the qualifier, **day**, is set to 2.

Fetching and Inserting DATETIME and INTERVAL Values

When an application fetches or inserts a DATETIME or INTERVAL value, 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 **dttime_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, ESQL/C must find a valid qualifier in the **datetime** host variable. ESQL/C takes one of the following actions, based on the value of the **dt_qual** field in the **dttime_t** structure that is associated with the host variable:

- When the **dt_qual** field contains a valid qualifier, 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 ESQL/C **dtextend()** function.
- When the **dt_qual** field does *not* contain a valid qualifier, ESQL/C takes different actions for a fetch and an insert:
 - For a fetch, ESQL/C uses the DATETIME column value and its qualifier to initialize the **datetime** host variable.
Zero is an invalid qualifier. Therefore, if you set the **dt_qual** field to zero, you can ensure that ESQL/C uses the qualifier of the DATETIME column.
 - For an insert, ESQL/C cannot perform the insert or update operation.
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, ESQL/C must find a valid qualifier in the **interval** host variable. 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, 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, 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, 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 ESQL/C **invextend()** function.

- When the **in_qual** field does *not* contain a valid qualifier, 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, 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, ESQL/C cannot perform the insert or update operation.

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. 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. For more information, see [“ANSI SQL Standards for DATETIME and INTERVAL Values” on page 6-36](#).

If the host variable is too short, 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.



Important: Informix products do not support automatic data conversion from DATETIME and INTERVAL column values to numeric (**double**, **int**, and so on) host variables.

You can also insert a DATETIME or INTERVAL column value from a character (**char**, **string**, **fixchar**, or **vchar**) host variable. 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. For more information, see [“ANSI SQL Standards for DATETIME and INTERVAL Values” on page 6-36](#).

If the conversion fails, 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.



Important: Informix products do not support automatic data conversion from numeric (**double**, **int**, and so on) and **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 following format:

YYYY-MM

A subset of this format is also valid: for example, just a month interval.

- The DAY TO FRACTION class has the following format:

DD HH:MM:SS.F

Any subset of contiguous fields is also valid: for example, MINUTE TO FRACTION.

Operations on datetime and interval Values

The ESQL/C **datetime** and **interval** data types use their own internal structure to store DATETIME and INTERVAL values. (For more information, see [“The datetime Data Type” on page 6-29](#) and [“The interval Data Type” on page 6-30](#).) Use the following ESQL/C library functions to perform all operations on **datetime** and **interval** host variables.

dtaddinv()	dtsub()	invdivdbl()	invmuldbl()
dtcurrent()	dtsubinv()	invdivinv()	

For more information on how to use these functions, refer to [“DATETIME and INTERVAL Library Functions” on page 6-38](#). For more information on operations on DATETIME and INTERVAL values, see the [Informix Guide to SQL: Reference](#).

Data Conversion for datetime and interval Values

You can use the 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 ESQL/C library functions **incvasc()**, **incvfmtasc()**, **intoasc()**, and **intofmtasc()**.

For example, you can perform conversions between the DATETIME and DATE data types with ESQL/C library functions and intermediate strings.

To convert a DATETIME value to a DATE value

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

To convert a DATE value into a DATETIME value

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 **rfmtdate()** with a pattern of *yyyy-mm-dd* to convert the DATE value to a character string.
3. Use **dtcvasc()** to convert the character string to a value in the prepared DATETIME variable.
4. If necessary, use **dtextend()** to adjust the DATETIME qualifier.

DATETIME and INTERVAL Library Functions

The following C functions are available in ESQL/C to handle **datetime** and **interval** host variables.

Function Name	Description
dtaddinv()	Adds an interval value to a datetime value
dtcurrent()	Gets the current date and time
dtcvasc()	Converts an ANSI-compliant character string to a datetime value
dtcvfmtasc()	Converts a character string with a specified format to a datetime value
dtextend()	Changes the qualifier of a datetime value
dtsub()	Subtracts one datetime value from another
dtsubinv()	Subtracts an interval value from a datetime value
dttoasc()	Converts a datetime value to an ANSI-compliant character string
dttofmtasc()	Converts a datetime value to a character string with a specified format
incvasc()	Converts an ANSI-compliant character string to an interval value
incvfmtasc()	Converts a character string with a specified format to an interval value
intoasc()	Converts an interval value to an ANSI-compliant character string
intofmtasc()	Converts an interval value to a character string with a specified format

(1 of 2)

Function Name	Description
invdivdbl()	Divides an interval value by a numeric value
invdivinv()	Divides an interval value by another interval value
invextend()	Extends an interval value to a different interval qualifier
invmuldbl()	Multiplies an interval value by a numeric value

(2 of 2)

When you compile your ESQL/C program with the **esql** command, **esql** automatically links these functions into your program. The following pages describe each of these functions.

dtaddinv()

The **dtaddinv()** function adds an **interval** value to a **datetime** value. The result is a **datetime** value.

Syntax

```
int dtaddinv(dt, inv, res)
    dttime_t *dt;
    intrvl_t *inv;
    dttime_t *res;
```

dt is a pointer to the initialized **datetime** host variable.

inv is a pointer to the initialized **interval** host variable.

res is 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.

Example

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=1994-11-28 11:40\n");
    dtcvasc("1994-11-28 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");
}

```

Example Output

```

DTADDINV Sample ESQL Program running.

datetime year to minute value=1994-11-28 11:40
interval day to minute value =                50 10:20
-----
                                Sum=1995-01-17 22:00

DTADDINV Sample Program over.

```

dtcurrent()

The **dtcurrent()** function assigns the current date and time to a **datetime** variable.

Syntax

```
void dtcurrent(d)
           dttime_t *d;
```

d is 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);
```

Example

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()
{
    int 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");
}
```

Example Output

```
DTCURRENT Sample ESQL Program running.

Today's datetime (year to minute) value is 1996-09-16 14:49

DTCURRENT Sample Program over.
```

dtcvasc()

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 [page 6-36](#).

Syntax

```
int dtcvasc(inbuf, dtvalue)
    char *inbuf;
    dttime_t *dtvalue;
```

inbuf is a pointer to the buffer that contains an ANSI-standard DATETIME string.

dtvalue is 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 20th century (it adds 1900 to the year value). 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.

Example Call

The following example initializes the host variable **leap_day** to the extra day of leap year 1996:

```
EXEC SQL BEGIN DECLARE SECTION;  
        datetime year to day leap_day;  
EXEC SQL END DECLARE SECTION;  
  
dctvasc("96-02-29",&leap_day);
```

Example

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 (dttime_t) structure.
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
    int 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 = 1994-02-11 3:10:35\n");
    if (x = dtcvasc("1994-02-11 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 = 1994-02-04 26:10:35\n");
    if (x = dtcvasc("1994-02-04 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");
}

```

Example Output

```

DTCVASC Sample ESQL Program running.

Datetime string #1 = 1994-02-11 3:10:35
Result = successful conversion

Datetime string #2 = 1994-02-04 26:10:35
Result = failed with conversion error:-1263

DTCVASC Sample Program over.

```

dtrcvfmtasc()

The **dtrcvfmtasc()** function uses a formatting mask to convert a character string to a **datetime** value.

Syntax

```
int dtrcvfmtasc(inbuf, fmtstring, dtvalue)
    char *inbuf;
    char *fmtstring;
    dttime_t *dtvalue;
```

inbuf is a pointer to the buffer that contains the string to convert.

fmtstring is 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, refer to the description of **DBTIME** in the [Informix Guide to SQL: Reference](#)).

dtvalue is 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, **dtrcvfmtasc()** 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 **dtrcvfmtasc()** 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 on acceptable digits and delimiters for a DATETIME value, see the “ANSI SQL Standards for DATETIME and INTERVAL Values” on page 6-36.

The **dttvfmtasc()** function returns an error if the formatting mask, *fmtstring*, is an empty string. If *fmtstring* is a null pointer, the **dttvfmtasc()** 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 on **DBTIME**, refer to the [Informix Guide to SQL: Reference](#).
2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set). For more information on **GL_DATETIME**, refer to the [Informix Guide to GLS Functionality](#).
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 (1995) or as two digits (95). When you use a two-digit year (**%y**) in a formatting mask, the **dttvfmtasc()** function uses the value of the **DBCENTURY** environment variable to determine which century to use. If you do not set **DBCENTURY**, **dttvfmtasc()** assumes the 20th century for two-digit years. For information on how to set **DBCENTURY**, see the [Informix Guide to SQL: Reference](#).

GLS

When you use a nondefault locale (one other than U.S. English) and do not set the **DBTIME** or **GL_DATETIME** environment variables, **dttvfmtasc()** uses the default DATETIME format that the locale defines. For more information, see the [Informix Guide to GLS Functionality](#). ♦

When the character string and the formatting mask are acceptable, the **dttvfmtasc()** 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.

Example

The **demo** directory contains this sample program in the file **dtdcvfmtasc.ec**. The code initializes the variable **birthday** to a fictitious birthday.

```

/* *dtdcvfmtasc.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];
    int 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("DTCVFMASC Sample ESQL Program running.\n\n");

    /* Initialize birthday to "09-06 13:30" */
    printf("Birthday #1 = September 6 at 01:30 pm\n");
    x = dtdcvfmtasc("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 = dtdcvfmtasc("July 14, 1988. Time: 9:15am",
        "%B %d, %Y. Time: %I:%M%p", &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. dtdcvfmtasc provides current information
    * for the missing year.
    */
    printf("Birthday #3 = July 14. Time: 9:15 am\n");
    x = dtdcvfmtasc("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("\nDTCVFM TASC Sample Program over.\n\n");
    }
```

Example Output

DTCVFM TASC 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 = 1988-07-14 09:15

Birthday #3 = July 14. Time: 9:15 am
Datetime (year to minute) value with current year = 1995-07-14 09:15

DTCVFM TASC Sample Program over.

dtextend()

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

```
int dtextend(in_dt, out_dt)  
          dttime_t *in_dt, *out_dt;
```

in_dt is a pointer to the **datetime** variable to extend.

out_dt is 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.

Return Codes

0	The operation was successful.
-1268	A parameter contains an invalid datetime qualifier.

Example Call

In these statements, 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);
```

Example

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()
{
    int 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");
}
```

Example Output

DTEXTEND Sample ESQL Program running.

Datetime (month to day) value = 12-07

Datetime (year to minute) extended value = 1995-12-07 00:00

DTEXTEND Sample Program over.

dtsub()

The **dtsub()** function subtracts one **datetime** value from another. The result is an **interval** value.

Syntax

```
int dtsub(d1, d2, inv)
        dttime_t *d1, *d2;
        intrvl_t *inv;
```

d1 is a pointer to an initialized **datetime** host variable.

d2 is a pointer to an initialized **datetime** host variable.

inv is 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.

Example

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()
{
    int 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 = 1994-10\n");
    dtcvasc("1994-10", &dt_var1);
    printf("Datetime (year to month) value #2 = 1991-08\n");
    dtcvasc("1991-08", &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 */
        dttoasc(&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 */
        dttoasc(&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 */
        dttoasc(&i_dth, out_str);
        printf("%s\n", out_str);
    }

    printf("\nDTSUB Sample Program over.\n\n");
}

```

Example Output

DTSUB Sample ESQL Program running.

Datetime (year to month) value #1 = 1994-10

Datetime (year to month) value #2 = 1991-08

Difference (year to month) = 0003-02

Difference (month to month) = 38

Difference (day to hour) = Error from dtsub(): -1266

DTSUB Sample Program over.

dtsubinv()

The **dtsubinv()** function subtracts an **interval** value from a **datetime** value. The result is a **datetime** value.

Syntax

```
int dtsubinv(dt, inv, res)
    dtm_t *dt;
    intrvl_t *inv;
    dtm_t *res;
```

dt is a pointer to an initialized **datetime** host variable.

inv is a pointer to an initialized **interval** host variable.

res is 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.

Example

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 = 1994-11-28\n");
    dtcvasc("1994-11-28 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");
}

```

Example Output

```

DTSUBINV Sample ESQL Program running.

Datetime (year to month) value = 1994-11-28
Interval (day to minute) value =                50 10:20
-----
Difference (year to hour)      = 1994-10-09  01:20

DTSUBINV Sample Program over.

```

dttoasc()

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 [page 6-36](#).

Syntax

```
int dttoasc(dtvalue, outbuf)
    dttime_t *dtvalue;
    char *outbuf;
```

- dtvalue* is a pointer to the initialized **datetime** variable to convert.
- outbuf* is 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.

Example

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");
}
```


Example Output

```
DTTOASC Sample ESQL Program running.
```

```
    Today's datetime (year to hour) value is 1995-09-19 08
```

```
DTTOASC Sample Program over.
```

dttofmtasc()

The **dttofmtasc()** function uses a formatting mask to convert a **datetime** variable to a character string.

Syntax

```
int dttofmtasc(dtvalue, outbuf, buflen, fmtstring)
    dttime_t *dtvalue;
    char *outbuf;
    int buflen;
    char *fmtstring;
```

dtvalue is a pointer to the initialized **datetime** variable to convert.

outbuf is a pointer to the buffer that receives the string for the value in *dtvalue*.

buflen is a length of the *outbuf* buffer.

fmtstring is 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, refer to the description of **DBTIME** in the [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 on **DBTIME**, refer to the [Informix Guide to SQL: Reference](#).
2. The format that the **GL_DATETIME** environment variable specifies (if **GL_DATETIME** is set). For more information on **GL_DATETIME**, refer to the [Informix Guide to GLS Functionality](#).
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 20th century for two-digit years. For information on how to set **DBCENTURY**, see the [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 [Informix Guide to GLS Functionality](#). ♦

Return Codes

- 0 The conversion was successful.
- <0 The conversion failed. Check the text of the error message.

Example

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];
    int x;

    EXEC SQL BEGIN DECLARE SECTION;
        datetime month to minute birthday;
    EXEC SQL END DECLARE SECTION;

    printf("DTTOFMTASC Sample ESQL Program runnin.\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");
}

```

Example 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, 1994 at 13:30

```

DTTOFMTASC Sample Program over.

incvasc()

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 [page 6-36](#).

Syntax

```
int incvasc(inbuf, invvalue)
    char *inbuf;
    intrvl_t *invvalue;
```

inbuf is a pointer to a buffer that contains an ANSI-standard INTERVAL string.

invvalue is 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()
{
    int 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 = 1994-02-11 3:10:35\n");
    in1.in_qual = TU_IENCODE(4, TU_YEAR, TU_SECOND);
    if(x = incvasc("1994-02-11 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");
}

```

Example 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 = 1994-02-11 3:10:35
Result = failed with conversion error:-1268

INCVASC Sample Program over.

```

incvfmtasc()

The **incvfmtasc()** function uses a formatting mask to convert a character string to an **interval** value.

Syntax

```
int incvfmtasc(inbuf, fmtstring, invvalue)  
    char *inbuf;  
    char *fmtstring;  
    intrvl_t *invvalue;
```

inbuf is a pointer to a buffer that contains the string to convert.

fmtstring is 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, refer to the description of **DBTIME** in the [Informix Guide to SQL: Reference](#)).

invvalue is 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 on acceptable digits and delimiters for an INTERVAL value, see [“ANSI SQL Standards for DATETIME and INTERVAL Values” on page 6-36](#).

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];
    int 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");
}

```

Example Output

```

INCVFMTASC 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

INCVFMTASC Sample Program over.

```

intoasc()

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 DATETIME standard, see [page 6-36](#).

Syntax

```
int intoasc(invvalue, outbuf)
          intrvl_t *invvalue;
          char *outbuf;
```

invvalue is a pointer to an initialized **interval** variable to convert.

outbuf is 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()
{
    int 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");
}

```

Example Output

```

INTOASC Sample ESQL Program running.

Interval (day(3) to day) string is '3'
Interval value after conversion is '    3'

INTOASC Sample Program over.

```

intofmtasc()

The **intofmtasc()** function uses a formatting mask to convert an **interval** variable to a character string.

Syntax

```
int intofmtasc(invvalue, outbuf, buflen, fmtstring)
    intrvl_t *invvalue;
    char *outbuf;
    int buflen;
    char *fmtstring;
```

invvalue is a pointer to an initialized **interval** variable to convert.

outbuf is a pointer to the buffer that receives the string for the value in *invvalue*.

buflen is the length of the *outbuf* buffer.

fmtstring is 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, refer to the description of **DBTIME** in the [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];
    int 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 ESQ Program running.\n\n");

/* Initialize short_time (day to minute) interval value */
    printf("\nInterval 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");
}

```

Example Output

```

INTOFMTASC Sample ESQ 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.

```


invdivdbl()

The **invdivdbl()** function divides an **interval** value by a numeric value.

Syntax

```
int invdivdbl(iv, num, ov)
    intrvl_t *iv;
    double num;
    intrvl_t *ov;
```

iv is a pointer to an **interval** variable to be divided.

num is a numeric divisor value.

ov is 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**.

```

/*
 * invdivdbl.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");
}

```

Example 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.
```

invdivinv()

The **invdivinv()** function divides an **interval** value by another **interval** value.

Syntax

```
int invdivinv(i1, i2, num)
    intrvl_t *i1, *i2;
    double *num;
```

i1 is a pointer to an **interval** variable that is the dividend.

i2 is a pointer to an **interval** variable that is the divisor.

num is 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()
{
    int 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");
}

```

Example 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.

```

invextend()

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

```
int invextend(in_inv, out_inv)
            intrvl_t *in_inv, *out_inv;
```

in_inv is a pointer to the **interval** variable to extend.

out_inv is 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()
{
    int 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()

```
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");
}
```

Example 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.

invmuldbl()

The **invmuldbl()** function multiplies an **interval** value by a numeric value.

Syntax

```
int invmuldbl(iv, num, ov)
    intrvl_t *iv;
    double num;
    intrvl_t *ov;
```

iv is a pointer to the **interval** variable to multiply.

num is the numeric **double** value.

ov is 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");
}

```

Example 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.
```


Working with INFORMIX- Universal Server Smart Large Objects

Data Structures for Smart Large Objects	7-6
Declaring a Host Variable	7-7
The LO-Specification Structure	7-7
The ifx_lo_create_spec_t Structure	7-8
ESQL/C Functions That Use the LO-Specification Structure	7-11
Obtaining Storage Characteristics	7-12
Deallocating the LO-Specification Structure	7-16
The LO-Pointer Structure	7-16
Storing a Smart Large Object	7-17
The ifx_lo_t Structure	7-18
ESQL/C Functions That Use the LO-Pointer Structure	7-19
The LO File Descriptor	7-20
ESQL/C Library Functions That Use an LO File Descriptor	7-20
Creating a Smart Large Object	7-21
Accessing a Smart Large Object	7-22
Selecting a Smart Large Object	7-22
Opening a Smart Large Object	7-23
Access Modes	7-23
Smart-Large-Object Locks	7-26
Duration of an Open on a Smart Large Object	7-27
Deletion of a Smart Large Object	7-28
Modifying a Smart Large Object	7-28
Reading Data From a Smart Large Object	7-28
Writing Data to a Smart Large Object	7-29
Closing a Smart Large Object	7-29

Obtaining Status of a Smart Large Object	7-29
Obtaining a Valid LO-Pointer Structure	7-30
Allocating and Accessing an LO-Status Structure	7-30
Allocating an LO-Status Structure	7-30
Accessing the LO-Status Structure	7-30
Deallocating the LO-Status Structure	7-32
Reading and Writing Smart Large Objects on an Optical Disc	7-32
The ESQL/C API for Smart Large Objects	7-33
ifx_lo_alter()	7-36
ifx_lo_close()	7-38
ifx_lo_col_info()	7-39
ifx_lo_copy_to_file()	7-41
ifx_lo_copy_to_lo()	7-44
ifx_lo_create()	7-46
ifx_lo_def_create_spec()	7-49
ifx_lo_filename()	7-50
ifx_lo_from_buffer()	7-52
ifx_lo_open()	7-53
ifx_lo_read()	7-55
ifx_lo_readwithseek()	7-57
ifx_lo_release()	7-59
ifx_lo_seek()	7-61
ifx_lo_spec_free()	7-63
ifx_lo_specget_estbytes()	7-64
ifx_lo_specget_extsz()	7-66
ifx_lo_specget_flags()	7-67
ifx_lo_specget_maxbytes()	7-69
ifx_lo_specget_sbospace()	7-71
ifx_lo_specset_estbytes()	7-73
ifx_lo_specset_extsz()	7-75
ifx_lo_specset_flags()	7-77
ifx_lo_specset_maxbytes()	7-79
ifx_lo_specset_sbospace()	7-80
ifx_lo_stat()	7-81
ifx_lo_stat_atime()	7-83
ifx_lo_stat_cspect()	7-84

ifx_lo_stat_ctime()	7-86
ifx_lo_stat_free()	7-87
ifx_lo_stat_mtime_sec()	7-88
ifx_lo_stat_refcnt()	7-89
ifx_lo_stat_size()	7-91
ifx_lo_tell()	7-92
ifx_lo_to_buffer()	7-93
ifx_lo_truncate()	7-95
ifx_lo_write()	7-96
ifx_lo_writewithseek()	7-98
The create_clob Program	7-100
Storage Characteristics for the Example	7-100
Source for the create_clob.ec File	7-101

T

he information in this chapter applies only if you are using Universal Server as your database server.

A smart large object is a large object that is stored in an sbpace on disk, and is recoverable. You can access a smart large object with read, write, and seek operations similar to an operating-system file. Smart large objects include the data types CLOB or BLOB. The CLOB data type stores any kind of text data. The BLOB data type can store any kind of binary data in an undifferentiated byte stream.

This chapter discusses 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
- Reading and writing smart large objects on an optical disc
- The 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 within an 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 [Informix Guide to SQL: Reference](#).

Data Structures for Smart Large Objects

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 ESQL/C program does not store the data directly in a host variable. Instead, the data is accessed by the client application as a file-like structure. To use smart-large-object variables in an 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 7-7](#).
- 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 7-7](#) and [“Obtaining Storage Characteristics” on page 7-12](#).
 - The LO-pointer structure, **ifx_lo_t**
For more information, see [“Deallocating the LO-Specification Structure” on page 7-16](#).
 - An integer LO file descriptor
For more information, see [“Opening a Smart Large Object” on page 7-23](#).



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, Informix 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 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 tabl;
```

The **ifx_lo_t** data type stores an *LO-pointer structure*. For more information, see [“The ifx_lo_t Structure” on page 7-18](#).



Tip: For more information on the **fixed binary** ESQL/C data type, see [“Accessing a Fixed-Length Opaque Type” on page 10-18](#).

To access smart large objects, you must include the **locator.h** header file in your 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 on the **ifx_lo_def_create_spec()** function, see [page 7-49](#).

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 ESQL/C program. The **locator.h** header file defines the LO-specification structure, so you must include the **locator.h** file in your ESQL/C programs that access this structure.



Important: The LO-specification structure, **ifx_lo_create_spec_t**, is an opaque structure to ESQL/C programs. Do not access its internal structure directly. Informix does not guarantee that the internal structure of **ifx_lo_create_spec_t** will not change in future releases. Therefore, to create portable code, always use the 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 [Figure 7-1 on page 7-9](#) and [Figure 7-2 on page 7-10](#).

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 Universal Server determine how to store the smart large object most efficiently on disk. [Figure 7-1](#) shows the disk-storage information along with the corresponding ESQL/C access functions.

Figure 7-1
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. Universal 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. Universal 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. Universal 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_specget_extsz(), ifx_lo_specset_extsz()
Name of the sbpace	The name of the sbpace that contains the smart large object. The sbpace name can be at most 18 characters long. This name must be null terminated.	ifx_lo_specget_sbpace(), ifx_lo_specset_sbpace()

For most applications, Informix recommends that you use the values for the disk-storage information that the database server determines. For more information on the disk-storage information, see the description of the appropriate ESQL/C access function in [“The ESQL/C API for Smart Large Objects” on page 7-33](#).

Create-Time Flags

The LO-specification structure stores create-time flags, which tell Universal Server what options to assign to the smart large object. Figure 7-2 shows the create-time flags along with the corresponding ESQL/C access functions.

Figure 7-2
Create-Time Flags in the LO-Specification Structure

Type of Indicator	Create-Time Flag	Description
Logging	LO_LOG	Tells Universal Server to log changes to the smart large object in the system log file. <i>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 the “Administrator’s Guide” for your database server.</i>
	LO_NOLOG	Tells Universal Server to turn off logging for all operations that involve the associated smart large object.
Last access-time	LO_KEEP_LASTACCESS_TIME	Tells Universal 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. <i>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.</i>
	LO_NOKEEP_LASTACCESS_TIME	Tells Universal 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 on the inheritance hierarchy, see [“Obtaining Storage Characteristics” on page 7-12](#).

For more information on the create-time flags, see the description of the **ifx_lo_specset_flags()** and **ifx_lo_specget_flags()** ESQL/C access function in [“The ESQL/C API for Smart Large Objects” on page 7-33](#).

ESQL/C Functions That Use the LO-Specification Structure

The following table shows the ESQL/C library functions that access the LO-specification structure.

ESQL/C Library Function	Purpose	For More Information
ifx_lo_col_info()	Updates the LO-specification structure with the column-level storage characteristics	page 7-39
ifx_lo_create()	Reads an LO-specification structure to obtain storage characteristics for a new smart large object that it creates	page 7-46
ifx_lo_def_create_spec()	Allocates and initializes an LO-specification structure	page 7-49
ifx_lo_spec_free()	Frees the resources of the LO-specification structure	page 7-63
ifx_lo_specget_estbytes()	Gets the estimated number of bytes from the LO-specification structure	page 7-64
ifx_lo_specget_extsz()	Gets the allocation extent size from the LO-specification structure	page 7-66
ifx_lo_specget_flags()	Gets the create-time flags from the LO-specification structure	page 7-67

(1 of 2)

ESQL/C Library Function	Purpose	For More Information
ifx_lo_specget_maxbytes()	Gets the maximum number of bytes from the LO-specification structure	page 7-69
ifx_lo_specget_sbospace()	Gets the name of the sbospace from the LO-specification structure	page 7-71
ifx_lo_specset_estbytes()	Sets the estimated number of bytes from the LO-specification structure	page 7-73
ifx_lo_specset_extsz()	Sets the allocation extent size in the LO-specification structure	page 7-75
ifx_lo_specset_flags()	Sets the create-time flags in the LO-specification structure	page 7-77
ifx_lo_specset_maxbytes()	Sets the maximum number of bytes in the LO-specification structure	page 7-79
ifx_lo_specset_sbospace()	Sets the name of the sbospace in the LO-specification structure	page 7-80
ifx_lo_stat_cspec()	Returns the storage characteristics into the LO-specification structure for a specified smart large object	page 7-84

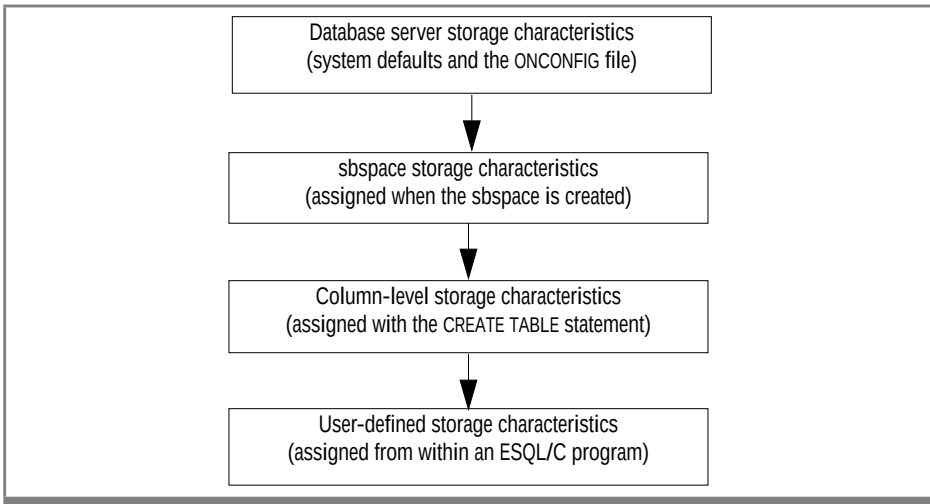
(2 of 2)

Obtaining 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.

Universal Server uses an inheritance hierarchy to obtain storage characteristics. [Figure 7-3 on page 7-13](#) shows the inheritance hierarchy for smart-large-object storage characteristics.

Figure 7-3
Inheritance
Hierarchy for
Storage
Characteristics



The System-Specified Storage Characteristics

Universal Server 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.

Universal 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.

Informix recommends 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 on database server and sbspace storage characteristics, see the description of the **onspaces** utility in the [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 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 [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 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 on how to obtain column-level storage characteristics, see [“The Column-Level Storage Characteristics” on page 7-14](#).

- 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 7-13](#).

To specify user-defined storage characteristics, use the appropriate ESQL/C accessor functions for the LO-specification structure. For more information on these accessor functions, see [“The LO-Specification Structure” on page 7-7](#).

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 on the **ifx_lo_spec_free()** function, see [page 7-63](#).

The LO-Pointer Structure

To open a smart large object for read and write operations, an 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 on [page 7-41](#).

To create an LO-pointer structure for a new smart large object, use the **ifx_lo_create()** function. The **ifx_lo_create()** 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 on the LO-specification structure, see [page 7-8](#).
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 on an LO file descriptor, see [page 7-27](#).

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 on the **ifx_lo_def_create_spec()** function, see [page 7-49](#).

Storing a Smart Large Object

An ESQL/C program accesses a smart large object through an LO-pointer structure. The ESQL/C library functions in the table on [page 7-19](#) 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 7-7.](#))

Figure 7-4 shows how the ESQL/C client application transfers the data of a smart large object to the database server.

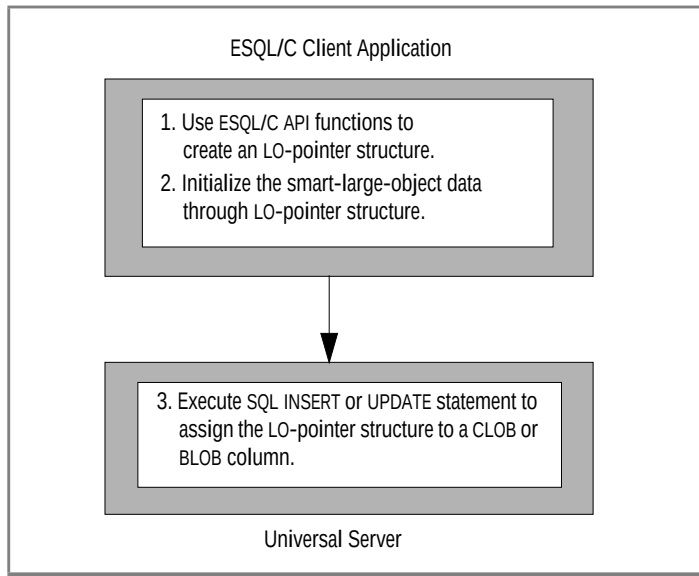


Figure 7-4
Transferring Smart-Large-Object Data from Client Application to Database Server

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 a sample of code that stores a new smart large object into a database column, see [“The create_clob Program” on page 7-100](#). For information on how to select a smart large object from the database, see [page 7-22](#).

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 ESQL/C programs that access this structure.



Important: The LO-pointer structure, **ifx_lo_t**, is an opaque structure to ESQL/C programs. That is, you should not access its internal structure directly. Informix does not guarantee that the internal structure of **ifx_lo_t** will not change. Therefore, to create portable code, use the appropriate ESQL/C library function (see [page 7-19](#)) 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 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 [page 7-7](#).

ESQL/C Functions That Use the LO-Pointer Structure

The following table shows the ESQL/C library functions that access the LO-pointer structure and how they access it.

ESQL/C Library Function	Purpose	For More Information
ifx_lo_copy_to_file()	Copies the smart large object that the LO-pointer structure identifies to an operating-system file.	page 7-41
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.	page 7-46
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.	page 7-50
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.	page 7-52
ifx_lo_release()	Tells the database server to release the resources associated with the temporary smart large object that the LO-pointer structure references.	page 7-59
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.	page 7-93

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 ESQL/C library functions that accepts LO file descriptors.

ESQL/C Library Functions That Use an LO File Descriptor

The following table shows the ESQL/C library functions that access the LO file descriptor.

ESQL/C Library Function	Purpose	For More Information
ifx_lo_close()	Closes the smart large object that the LO file descriptor identifies and deallocates the LO file descriptor	page 7-38
ifx_lo_copy_to_lo()	Copies an operating-system file to an open smart large object that the LO file descriptor identifies	page 7-44
ifx_lo_create()	Creates and opens a new smart large object and returns an LO file descriptor	page 7-46
ifx_lo_open()	Opens a smart large object and returns an LO file descriptor	page 7-52
ifx_lo_read()	Reads data from the open smart large object that the LO file descriptor identifies	page 7-55
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	page 7-57
ifx_lo_seek()	Moves the file position in the open smart large object that the LO file descriptor identifies	page 7-61

(1 of 2)

ESQL/C Library Function	Purpose	For More Information
ifx_lo_stat()	Obtains status information for the open smart large object that the LO file descriptor identifies	page 7-61
ifx_lo_tell()	Determines the current file position in the open smart large object that the LO file descriptor identifies	page 7-92
ifx_lo_truncate()	Truncates at a specified offset the open smart large object that the LO file descriptor identifies	page 7-93
ifx_lo_write()	Writes data to the open smart large object that the LO file descriptor identifies	page 7-96
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	page 7-98

(2 of 2)

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 7-29](#).

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 on creating a smart large object, see [“The LO-Specification Structure” on page 7-7](#).

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 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 7-7](#).)

Figure 7-5 shows how the database server transfers the data of a smart large object to the ESQL/C client application.

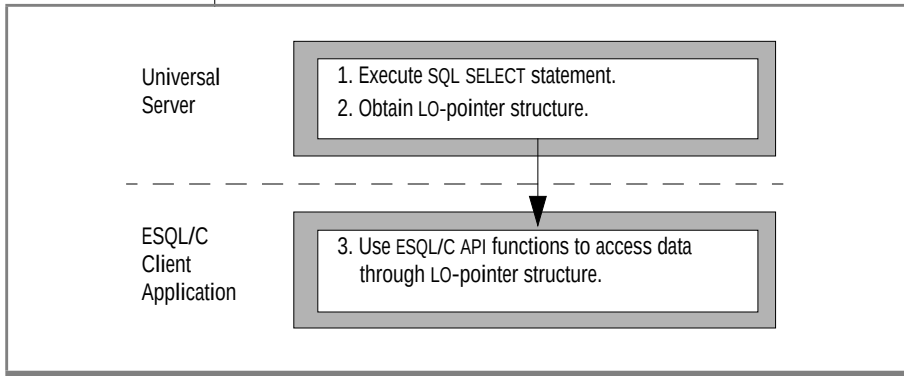


Figure 7-5
Transferring Smart-Large-Object Data from Database Server to Client Application

For a sample of code that selects a smart large object from a database column, see [“The create_clob Program” on page 7-100](#). For information on how to store a smart large object in the database, see [page 7-17](#).

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.

Figure 7-6 shows the access modes and their corresponding defined constants that the **ifx_lo_open()** and **ifx_lo_create()** functions support.

Figure 7-6
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
Write-only mode	Only write operations are valid on the data.	LO_WRONLY
Write / append mode	Appends any data you write to the end of the smart large object. By itself, it is equivalent to write-only mode followed by a seek to the end of the smart large object. Read operations fail.	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



Tip: These access-mode flags for a smart large object are patterned after the UNIX System V access modes.

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 has been 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 has been 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 the *INFORMIX-Universal Server 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 has been 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 on the **onspaces** utility, refer to the [Administrator's Guide](#) for your database server.

Smart-Large-Object Locks

To prevent simultaneous access to smart-large-object data, the database server locks a smart large object when you open it. 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 objects because many columns can contain the same smart-large-object data.

You use the access-mode flags, `LO_RDONLY`, `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 data. 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.*

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.

Deletion of 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 on how to save the smart large object, see [“Storing a Smart Large Object” on page 7-17](#).

Reading Data From a Smart Large Object

The **ifx_lo_read()** and **ifx_lo_readwithseek()** 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 on [page 7-55](#) and the **ifx_lo_readwithseek()** function on [page 7-57](#).

Writing Data to a Smart Large Object

The **ifx_lo_write()** and **ifx_lo_writewithseek()** 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 on [page 7-96](#) and the **ifx_lo_writewithseek()** function on [page 7-98](#).

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 on the **ifx_lo_close()** function, see [page 7-38](#).

Obtaining 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 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 7-22](#).
- Create a new smart large object.
For more information, see [“Deallocating the LO-Specification Structure” on page 7-16](#).

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 on the **ifx_lo_stat()** function, see [page 7-81](#).

Accessing the LO-Status Structure

The LO-status structure, **ifx_lo_stat_t**, stores the status information for a smart large object in an ESQL/C program. The **locator.h** header file defines the LO-status structure so you must include the **locator.h** file in your ESQL/C programs that access this structure.



Important: The LO-status structure, `ifx_lo_stat_t`, is an opaque structure to ESQL/C programs. Do not access its internal structure directly. Informix does not guarantee that the internal structure of `ifx_lo_stat_t` will not change in future releases. Therefore, to create portable code, always use the ESQL/C accessor functions for this structure (which Figure 7-7 shows) to obtain and store values in the LO-status structure.

Figure 7-7 shows the status information along with the corresponding ESQL/C accessor functions.

Figure 7-7
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 <code>LO_KEEP_LASTACCESS_TIME</code> flag is set for this smart large object.	<code>ifx_lo_stat_atime()</code>
Storage characteristics	The storage characteristics for the smart large object. These characteristics are stored in an LO-specification structure (see page 7-7). Use the ESQL/C accessor functions for an LO-specification structure (see Figure 7-1 on page 7-9 and Figure 7-2 on page 7-10) to obtain this information.	<code>ifx_lo_stat_cspec()</code>
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.	<code>ifx_lo_stat_ctime()</code>
Last modification time (seconds)	The time, in seconds, that the smart large object was last modified.	<code>ifx_lo_stat_mtime_sec()</code>
Reference count	A count of the number of references to the smart large object.	<code>ifx_lo_stat_refcnt()</code>
Size	The size, in bytes, of the smart large object.	<code>ifx_lo_stat_size()</code>

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.

UNIX

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 on the **ifx_lo_stat_free()** function, see [page 7-87](#).

Reading and Writing Smart Large Objects on an Optical Disc

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 [page 7-16](#)). The LO-pointer structure can point to an sbpage in an sbpace 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* (WORM). The optical disc must contain the sbspaces for the smart large objects. Your application can use the ESQL/C API for smart large objects (which this chapter describes) to access the smart large objects on the mounted optical disc.

Universal 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” on page 8-40](#).

For details about the optical subsystem of Universal Server, see the *INFORMIX-Universal Server Administrator's Guide* and the [INFORMIX-OnLine/Optical User Manual](#). ♦

The ESQL/C API for Smart Large Objects

The 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 ESQL/C application program accesses smart large objects through the ESQL/C library functions in the following table.

ESQL/C Function	Description
ifx_lo_alter()	Alters the storage characteristics of an existing smart large object
ifx_lo_close()	Closes an open smart large object
ifx_lo_col_info()	Retrieves column-level storage characteristics in an LO-specification structure
ifx_lo_copy_to_file()	Copies a smart large object into an operating-system file
ifx_lo_copy_to_lo()	Copies an operating-system file into an open smart large object
ifx_lo_create()	Creates an LO-pointer structure for a smart large object
ifx_lo_def_create_spec()	Allocates an LO-specification structure and initializes its fields to null values
ifx_lo_filename()	Returns the generated filename, given an LO-pointer structure and a file specification
ifx_lo_from_buffer()	Copies a specified number of bytes from a user-defined buffer into a smart large object
ifx_lo_open()	Opens an existing smart large object
ifx_lo_read()	Reads a specified number of bytes from an open smart large object
ifx_lo_readwithseek()	Seeks to a specified position in an open smart large object and reads a specified number of bytes

(1 of 3)

ESQ/C Function	Description
ifx_lo_release()	Releases resources committed to a temporary smart large object
ifx_lo_seek()	Sets the seek position for the next read or write on an open smart large object
ifx_lo_spec_free()	Frees the resources allocated to an LO-specification structure
ifx_lo_specget_estbytes()	Gets the estimated size, in bytes, of the smart large object
ifx_lo_specget_extsz()	Gets the allocation extent size for the smart large object
ifx_lo_specget_flags()	Gets the create-time flags for the smart large object
ifx_lo_specget_maxbytes()	Gets the maximum size for the smart large object
ifx_lo_specset_sbspace()	Gets the sbspace name for the smart large object
ifx_lo_specset_estbytes()	Sets the estimated size, in bytes, of the smart large object
ifx_lo_specset_extsz()	Sets the allocation extent size for the smart large object
ifx_lo_specset_flags()	Sets the create-time flags for the smart large object
ifx_lo_specset_maxbytes()	Sets the maximum size for the smart large object
ifx_lo_specset_sbspace()	Sets the sbspace name for the smart large object
ifx_lo_stat()	Obtains status information for an open smart large object
ifx_lo_stat_atime()	Returns the last access time for a smart large object
ifx_lo_stat_cspec()	Returns the storage characteristics for a smart large object
ifx_lo_stat_ctime()	Returns the last change-in-status time for the smart large object
ifx_lo_stat_free()	Frees the resources allocated to an LO-status structure

(2 of 3)

ESQL/C Function	Description
ifx_lo_stat_mtime_sec()	Returns the last modification time, in seconds, for the smart large object
ifx_lo_stat_refcnt()	Returns the reference count for the smart large object
ifx_lo_stat_size()	Returns the size of the smart large object
ifx_lo_tell()	Returns the current seek position of an open smart large object
ifx_lo_to_buffer()	Copies a specified number of bytes from a smart large object into a user-defined buffer
ifx_lo_truncate()	Truncates a smart large object to a specific offset
ifx_lo_write()	Writes a specified number of bytes to an open smart large object
ifx_lo_writewithseek()	Seeks to a specified position in an open smart large object and writes a specified number of bytes

(3 of 3)

The following pages describe each of these functions in more detail.

ifx_lo_alter()

The **ifx_lo_alter()** function alters the storage characteristics of an existing smart large object.

Syntax

```
int ifx_lo_alter(LO_ptr, LO_spec)
    ifx_lo_t *LO_ptr;
    ifx_lo_create_spec_t *LO_spec;
```

LO_ptr is a pointer to an LO-pointer structure that identifies the smart large object whose storage characteristics are altered. For more information on LO-pointer structures, see [page 7-16](#).

LO_spec is 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 on the LO-specification structure, see [page 7-7](#).

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_specset_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 Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

Related Functions

**`ifx_lo_col_info()`, `ifx_lo_create()`, `ifx_lo_def_create_spec()`,
`ifx_lo_specset_extsz()`, `ifx_lo_specset_flags()`**

ifx_lo_close()

The **ifx_lo_close()** function closes an open smart large object.

Syntax

```
int ifx_lo_close(LO_fd)  
    int LO_fd;
```

LO_fd is the LO file descriptor of the smart large object to close. For more information on an LO file descriptor, see [page 7-23](#).

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()** or **ifx_lo_create()** function returns an LO file descriptor for a smart large object when it successfully opens it.

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 Values

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

Related Functions

ifx_lo_create(), ifx_lo_open()

ifx_lo_col_info()

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

```
int ifx_lo_col_info(column_name, LO_spec)
char *column_name;
ifx_lo_create_spec_t *LO_spec;
```

column_name is a pointer to a buffer that contains the name of the database column whose column-level storage characteristics you wish to use.

LO_spec is a pointer to the LO-specification structure in which to store the column-level storage characteristics for *column_name*. For more information on the LO-specification structure, see [page 7-7](#).

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 on the inheritance hierarchy, see [“Obtaining Storage Characteristics” on page 7-12](#).

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()**.

ANSI



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

ifx_lo_create()*, *ifx_lo_def_create_spec()

ifx_lo_copy_to_file()

The **ifx_lo_copy_to_file()** function copies the contents of a smart large object into an operating-system file.

Syntax

```
int ifx_lo_copy_to_file(LO_ptr, fname, flags, result)
    ifx_lo_t *LO_ptr;
    char *fname;
    int flags;
    char *result;
```

LO_ptr is a pointer to the LO-pointer structure that you provide to identify the smart large object to copy. For more information on LO-pointer structures, see [page 7-16](#).

fname is the full pathname of the target file to hold the data.

flags is an integer that specifies the location of the *fname* file.

result is a pointer to a buffer that contains the filename 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 <i>fna5me</i> file is on the client computer.
LO_SERVER_FILE	The <i>fname</i> file is on the server computer.

If you do not specify a value for *flags*, the default location is the client computer (LO_CLIENT_FILE).

By default, the **ifx_lo_copy_to_file()** function generates a filename of the form:

```
fname.hex_id
```

In this format, *fname* is the filename 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 follows:

```
'/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 filename, you can specify the following wildcards in the filename portion of *fname*:

- One or more contiguous question mark (?) characters in the filename can generate a unique filename.

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 follows:

```
'/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/resumb2.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 filename indicates that the filename does not need to be unique.

For example, suppose you specify a *pathname* value as follows:

```
'/tmp/resume.txt!'
```

The **ifx_lo_copy_to_file()** function does not use the smart-large-object identifier in the filename so it generates the following file:

```
/tmp/resume.txt
```

The exclamation point overrides the question marks in the filename specification.



Tip: These wildcards are also valid in the “fname” argument of the **ifx_lo_filename()** function. For more information on **ifx_lo_filename()**, see [page 7-50](#).

Your application must ensure that there is sufficient space to hold the generated file.

Return Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

Related Functions

ifx_lo_copy_to_lo(), ifx_lo_filename()

ifx_lo_copy_to_lo()

The **ifx_lo_copy_to_lo()** function copies the contents of a file into an open smart large object.

Syntax

```
int ifx_lo_copy_to_lo(LO_fd, fname, flags)
    int LO_fd;
    char *fname;
    int flags;
```

- LO_fd* is the LO file descriptor for the open smart large object in which to write the file contents. For more information on an LO file descriptor, see [page 7-23](#).
- fname* is the full pathname of the source file that contains the data to copy.
- flags* is 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 <i>fname</i> file is on the client computer.
LO_SERVER_FILE	The <i>fname</i> file is on the server computer.
LO_APPEND	Append the data in <i>fname</i> to the end of the specified smart large object. This flag can be masked with one of the preceding flags.

Return Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

Related Functions

`ifx_lo_copy_to_file()`

ifx_lo_create()

The **ifx_lo_create()** function creates a new smart large object and opens it for access within an ESQL/C program.

Syntax

```
int ifx_lo_create(LO_spec, flags, LO_ptr, error)
    ifx_lo_create_spec_t *LO_spec;
    int flags;
    ifx_lo_t *LO_ptr;
    int *error;
```

- LO_spec* is a pointer to the LO-specification structure that contains the storage characteristics for new smart large objects. For information on the LO-specification structure, see [page 7-7](#).
- flags* is an integer that specifies the mode in which to open the new smart large object. For more information, see “[Access Modes](#)” on [page 7-23](#).
- LO_ptr* is a pointer to the LO-pointer structure for the new smart large object. For more information on LO-pointer structures, see [page 7-16](#).
- error* is 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 on the inheritance hierarchy, see [“Obtaining Storage Characteristics” on page 7-12](#).

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 [Figure 7-6 on page 7-24](#) shows. For more information on access modes, see [“Opening a Smart Large Object” on page 7-23](#).

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.

Universal Server 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 7-25](#).

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 pointer to a valid LO file descriptor. You can then use this file descriptor, *LO_fd*, to identify which smart large object to access in subsequent function calls such as **ifx_lo_read()** and **ifx_lo_write()**. However, this *LO_fd* file descriptor is only valid within the current database connection.

Return Values

A valid LO file descriptor	The function successfully created and opened the new smart large object and returned a valid LO file descriptor.
-1	The function was not successful; examine the <i>error</i> for a detailed error code.

Related Functions

ifx_lo_close(), **ifx_lo_col_info()**, **ifx_lo_def_create_spec()**

ifx_lo_def_create_spec()

The **ifx_lo_def_create_spec()** function allocates and initializes an LO-specification structure.

Syntax

```
int ifx_lo_def_create_spec(LO_spec)
    ifx_lo_create_spec_t **LO_spec;
```

LO_spec is a pointer that points to a pointer to a new LO-specification structure that contains initialized fields. For information on the LO-specification structure, see [page 7-7](#).

Usage

The **ifx_lo_def_create_spec()** function initializes the fields in the LO-specification structure, to which *LO_spec* points, to the appropriate null values. The database server later interprets these null values to mean that system-specified defaults should be used for the storage characteristics of the smart large object. For more information, see “[The System-Specified Storage Characteristics](#)” on [page 7-13](#).



Important: You must call the **ifx_lo_def_create_spec()** function before you use an LO-specification structure in your ESQL/C program. You can use the **ifx_lo_col_info()** function to obtain storage characteristics that are associated with a particular column.

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

ifx_lo_col_info(), **ifx_lo_create()**, **ifx_lo_spec_free()**

ifx_lo_filename()

The **ifx_lo_filename()** function returns the pathname that the database server would use if you copied a smart large object to a file.

Syntax

```
int ifx_lo_filename(LO_ptr, fname, result, result_buf_nbytes)
    ifx_lo_t *LO_ptr;
    char *fname;
    char *result;
    int result_buf_nbytes;
```

LO_ptr is a pointer to the LO-pointer structure that identifies the smart large object to copy. For more information on LO-pointer structures, see [page 7-16](#).

fname is the full pathname of the target file to hold the data.

result is a pointer to a buffer that contains the filename that **ifx_lo_copy_to_file()** would generate.

result_len is the size, in bytes, of the *result* character buffer.

Usage

The **ifx_lo_filename()** function generates a filename from the *fname* argument that you provide. Use the **ifx_lo_filename()** function to determine the filename 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 filename of the form:

```
fname.hex_id
```

However, you can specify wildcards in the *fname* argument that can change this default filename. You can use these wildcards in the *fname* argument of **ifx_lo_filename()** to see what filename these wildcards generate. For more information on the wildcards that are valid in the *fname* argument, see the description of the **ifx_lo_copy_to_file()** function on [page 7-41](#).

Return Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

Related Functions

`ifx_lo_copy_to_file()`

ifx_lo_from_buffer()

The **ifx_lo_from_buffer()** function copies a specified number of bytes from a user-defined buffer into a smart large object.

Syntax

```
int ifx_lo_from_buffer(LO_ptr, size, buffer, error)
    ifx_lo_t *LO_ptr;
    int size;
    char *buffer;
    int *error;
```

<i>LO_ptr</i>	is the LO-pointer structure for the smart large object into which you want to copy the data.
<i>size</i>	is an integer that identifies the number of bytes to copy to the smart large object.
<i>buffer</i>	is a pointer to a user-defined buffer from which you want to copy the data.
<i>error</i>	contains the address of the integer 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 Values

0	The function was successful.
-1	The function was not successful.

ifx_lo_open()

The **ifx_lo_open()** function opens an existing smart large object for access.

Syntax

```
int ifx_lo_open(LO_ptr, flags, error)
    ifx_lo_t *LO_ptr;
    int flags;
    int *error;
```

- LO_ptr* is a pointer to the LO-pointer structure that identifies the smart large object to open. For more information on LO-pointer structures, see [page 7-16](#).
- flags* is an integer that specifies the mode in which to open the smart large object that *LO_ptr* identifies. For more information, see [“Access Modes” on page 7-23](#).
- error* is a pointer to an integer that contains the error code that **ifx_lo_open()** sets.

Usage

Your ESQL/C program must call the **ifx_lo_open()** function for each instance of a smart large object that it needs to access.

The flag values for the *flags* argument indicate the mode of the smart large object after **ifx_lo_open()** successfully completes. Valid values include all access-mode constants, which [Figure 7-6 on page 7-24](#) shows. For more information on access modes, see [“Opening a Smart Large Object” on page 7-23](#).

Universal Server uses the default parameters that the call to **ifx_lo_open()** (or **ifx_lo_create()**) has established 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 7-25](#).

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 pointer to a valid LO file descriptor. You can then use this file descriptor, *LO_fd*, to identify which smart large object to access in subsequent function calls such as **ifx_lo_read()** and **ifx_lo_write()**. However, this *LO_fd* file descriptor is only valid 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). It also obtains a lock on the entire smart large object. The type of lock depends on the access mode in which the smart large object is opened, as follows:

- 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 Values

-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

ifx_lo_close(), **ifx_lo_create()**, **ifx_lo_read()**, **ifx_lo_write**

ifx_lo_read()

The **ifx_lo_read()** function reads a specified number of bytes of data from an open smart large object.

Syntax

```
int ifx_lo_read(LO_fd, buf, nbytes, error)
    int LO_fd;
    char *buf;
    int nbytes;
    int *error;
```

LO_fd is the LO file descriptor for the smart large object from which to read. For more information on an LO file descriptor, see [page 7-23](#).

buf is a pointer to a character buffer that contains the data that **ifx_lo_read()** reads from the smart large object.

nbytes is the size, in bytes, of the *buf* character buffer. This value cannot exceed 2 gigabytes.

error is a pointer to an integer 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 gigabytes in size. To read smart large objects that are larger than 2 gigabytes, read them in 2-gigabyte chunks.

Return Values

- >=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

ifx_lo_open(), ifx_lo_readwithseek(), ifx_lo_seek(), ifx_lo_tell(), ifx_lo_write()

ifx_lo_readwithseek()

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

```
int ifx_lo_readwithseek(LO_fd, buf, nbytes, offset, whence,  
error)  
    char *buf;  
    int nbytes;  
    ifx_int8_t *offset;  
    int whence;  
    int *error;
```

<i>LO_fd</i>	is the LO file descriptor for the smart large object from which to read. For more information on an LO file descriptor, see page 7-23 .
<i>buf</i>	is a pointer to a character buffer that contains the data that ifx_lo_readwithseek() reads from the smart large object.
<i>nbytes</i>	is the size, in bytes, of the <i>buf</i> character buffer. This value cannot exceed 2 gigabytes.
<i>offset</i>	is a pointer to the 8-byte integer (INT8) offset from the starting seek position.
<i>whence</i>	is an integer value that identifies the starting seek position.
<i>error</i>	is a pointer to an integer 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 on how to access eight-bit (INT8) integers, see [“The int8 Data Type” on page 5-7](#).

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 gigabytes. To read smart large objects that are larger than 2 gigabytes, read them in 2-gigabyte chunks.

Return Values

- >=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

ifx_lo_open(), ifx_lo_read(), ifx_lo_seek(), ifx_lo_write()

ifx_lo_release()

The **ifx_lo_release()** function tells the database server to release the resources associated with a temporary smart large object.

Syntax

```
int ifx_lo_release(LO_ptr)
    ifx_lo_t *LO_ptr;
```

LO_ptr is 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.

ifx_lo_release()

Return Values

- 0 The function was successful.
- < 0 The function was *not* successful.

ifx_lo_seek()

The **ifx_lo_seek()** function sets the file position for the next read or write operation on the open smart large object.

Syntax

```
int ifx_lo_seek(LO_fd, offset, whence, seek_pos)
    int LO_fd;
    ifx_int8_t *offset;
    int whence;
    ifx_int8_t *seek_pos;
```

- LO_fd* is the LO file descriptor for the smart large object whose seek position you wish to change. For more information on an LO file descriptor, see [page 7-23](#).
- offset* is a pointer to the 8-byte integer offset from the starting seek position.
- whence* is an integer value that identifies the starting seek position.
- seek_pos* is a pointer to the 8-byte integer that identifies the new seek position.

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 Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

Related Functions

ifx_lo_read(), ifx_lo_tell(), ifx_lo_write()

ifx_lo_spec_free()

The **ifx_lo_spec_free()** function frees the resources of an LO-specification structure.

Syntax

```
int ifx_lo_spec_free(LO_spec)
    ifx_lo_create_spec_t *LO_spec;
```

LO_spec is a pointer to the LO-specification structure to free. For information on the LO-specification structure, see [page 7-7](#).

Usage

The **ifx_lo_spec_free()** function frees the LO-specification structure to which *LO_spec* points. This structure was allocated by a call to **ifx_lo_def_create_spec()**. ESQL/C does not perform memory management for LO-specification structures. You must call **ifx_lo_spec_free()** for each LO-specification structure that you have allocated with a call to the **ifx_lo_def_create_spec()** function.

Return Values

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

Related Functions

ifx_lo_def_create_spec()

ifx_lo_specget_estbytes()

The **ifx_lo_specget_estbytes()** function obtains from an LO-specification structure the estimated size of a smart large object.

Syntax

```
int ifx_lo_specget_estbytes(LO_spec, estbytes)
    ifx_lo_create_spec_t *LO_spec;
    ifx_int8_t *estbytes;
```

LO_spec is a pointer to the LO-specification structure from which to obtain the estimated size. For information on the LO-specification structure, see [page 7-7](#).

estbytes is 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 on the estimated size, see [Figure 7-1 on page 7-9](#).

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 Values

0	The function was successful.
-1	The function was not successful.

Related Functions

**ifx_lo_def_create_spec(), ifx_lo_spec_free(), ifx_lo_specget_extsz(),
ifx_lo_specget_flags(), ifx_lo_specget_maxbytes(),
ifx_lo_specget_sbspace(), ifx_lo_specset_estbytes()**

ifx_lo_specget_extsz()

The **ifx_lo_specget_extsz()** function obtains from an LO-specification structure the allocation extent size of a smart large object.

Syntax

```
int ifx_lo_specget_extsz(LO_spec)
    ifx_lo_create_spec_t *LO_spec;
```

LO_spec is a pointer to the LO-specification structure from which to obtain the extent size. For information on the LO-specification structure, see [page 7-7](#).

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 Universal Server generates for how large an extent should be. For more information on the allocation extent, see [Figure 7-1 on page 7-9](#).



Important: You must call the **ifx_lo_def_create_spec()** function to initialize an LO-specification structure before you call **ifx_lo_specget_extbytes()**. 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 Values

- >=0 The function was successful and the return value indicates the extent size.
- 1 The function was not successful.

Related Functions

ifx_lo_def_create_spec(), **ifx_lo_spec_free()**, **ifx_lo_specget_extbytes()**, **ifx_lo_specget_flags()**, **ifx_lo_specget_maxbytes()**, **ifx_lo_specget_sbspace()**, **ifx_lo_specset_extsz()**

ifx_lo_specget_flags()

The **ifx_lo_specget_flags()** function obtains from an LO-specification structure the create-time flags of a smart large object.

Syntax

```
int ifx_lo_specget_flags(LO_spec)
    ifx_lo_create_spec_t *LO_spec;
```

LO_spec is a pointer to the LO-specification structure from which to obtain the flag value. For information on the LO-specification structure, see [page 7-7](#).

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 on the create-time flags, see [Figure 7-2 on page 7-10](#).

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 Values

- >=0 The function was successful and the return value is the value of the create-time flags.
- 1 The function was not successful.



ifx_lo_specget_flags()

Related Functions

**ifx_lo_def_create_spec(), ifx_lo_spec_free(), ifx_lo_specget_estbytes(),
ifx_lo_specget_extsz(), ifx_lo_specget_maxbytes(),
ifx_lo_specget_sbspace(), ifx_lo_specset_flags()**

ifx_lo_specget_maxbytes()

The **ifx_lo_specget_maxbytes()** function obtains from an LO-specification structure the maximum size of a smart large object.

Syntax

```
int ifx_lo_specget_maxbytes(LO_spec, maxbytes)
    ifx_lo_create_spec_t *LO_spec;
    ifx_int8_t *maxbytes;
```

LO_spec is a pointer to the LO-specification structure from which to obtain the maximum size. For information on the LO-specification structure, see [page 7-7](#).

maxbytes is a pointer to an **int8** value into which **ifx_lo_specget_maxbytes()** puts the maximum size, in bytes, of the smart large object.

Usage

Universal Server does not allow the size of a smart large object to exceed the *maxbytes* value. For more information on the maximum size, see [Figure 7-1 on page 7-9](#).



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 Values

- 0 The function was successful.
- 1 The function was not successful.

ifx_lo_specget_maxbytes()

Related Functions

**ifx_lo_def_create_spec(), ifx_lo_spec_free(), ifx_lo_specget_estbytes(),
ifx_lo_specget_extsz(), ifx_lo_specget_flags(), ifx_lo_specget_sbspace(),
ifx_lo_specset_maxbytes()**

ifx_lo_specget_sbospace()

The **ifx_lo_specget_sbospace()** function obtains from an LO-specification structure the name of an sbospace where a smart large object is stored.

Syntax

```
int ifx_lo_specget_sbospace(LO_spec, sbospace_name, length)
    ifx_lo_create_spec_t *LO_spec;
    char *sboospace_name;
    int length;
```

- LO_spec* is a pointer to the LO-specification structure from which to obtain the sbospace name. For information on the LO-specification structure, see [page 7-7](#).
- sboospace_name* is a character buffer into which **ifx_lo_specget_sbospace()** puts the name of the sbospace for the smart large object.
- length* is an integer value that specifies the size, in bytes, of the *sboospace_name* buffer.

Usage

The **ifx_lo_specget_sbospace()** function returns the current setting for the name of the sbospace in which to store the smart large object. The function copies up to *(length-1)* bytes into the *sboospace_name* buffer and ensures that it is null terminated. For more information on an sbospace name, see [Figure 7-1 on page 7-9](#).



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.

An sbospace name can be up to 18 characters long. However, you might want to allocate at least 129 bytes for the *sboospace_name* buffer to accommodate future increases in the length of an sbospace name.

Return Values

- 0 The function was successful.
- 1 The function was not successful.

Related Functions

ifx_lo_def_create_spec(), ifx_lo_spec_free(), ifx_lo_specget_estbytes(), ifx_lo_specget_extsz(), ifx_lo_specget_flags(), ifx_lo_specget_maxbytes(), ifx_lo_specset_sbospace()

ifx_lo_specset_estbytes()

The **ifx_lo_specset_estbytes()** function sets the estimated size of a smart large object.

Syntax

```
int ifx_lo_specset_estbytes(LO_spec, estbytes)
    ifx_lo_create_spec_t *LO_spec;
    ifx_int8_t *estbytes;
```

LO_spec is a pointer to the LO-specification structure in which to save the estimated size. For information on the LO-specification structure, see [page 7-7](#).

estbytes is 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 on the estimated byte size, see [Figure 7-1 on page 7-9](#).

If you do not specify an *estbytes* value when you create a new smart large object, Universal Server obtains the value from the inheritance hierarchy of storage characteristics. For more information on the inheritance hierarchy, see [“Obtaining Storage Characteristics” on page 7-12](#).

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 allocated unused storage.

ifx_lo_specset_estbytes()

Return Values

- 0 The function was successful.
- 1 The function was not successful.

Related Functions

**ifx_lo_def_create_spec(), ifx_lo_specget_estbytes(), ifx_lo_specset_extsz(),
ifx_lo_specset_flags(), ifx_lo_specset_maxbytes(),
ifx_lo_specset_sbspace(), ifx_lo_spec_free()**

ifx_lo_specset_extsz()

The **ifx_lo_specset_extsz()** function sets the allocation extent size for a smart large object.

Syntax

```
int ifx_lo_specset_extsz(LO_spec, extsz)
    ifx_lo_create_spec_t *LO_spec;
    int extsz;
```

LO_spec is a pointer to the LO-specification structure in which to save the extent size. For information on the LO-specification structure, see [page 7-7](#).

extsz is 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 Universal Server generates for how large an extent should be. For more information on the allocation extent, see [Figure 7-1 on page 7-9](#).

If you do not specify an *extsz* value when you create a new smart large object, Universal Server 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 on the inheritance hierarchy, see [“Obtaining Storage Characteristics” on page 7-12](#).

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.

ifx_lo_specset_extsz()

Return Values

- 0 The function was successful.
- 1 The function was not successful.

ifx_lo_specset_flags()

The **ifx_lo_specset_flags()** function sets the create-time flags of a smart large object.

Syntax

```
int ifx_lo_specset_flags(LO_spec, flags)
ifx_lo_create_spec_t *LO_spec;
int flags;
```

LO_spec is a pointer to the LO-specification structure in which to save the flags value. For information on the LO-specification structure, see [page 7-7](#).

flags is 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 on the create-time flags, see [Figure 7-2 on page 7-10](#).

If you do not specify a *flags* value when you create a new smart large object, Universal Server obtains the value from the inheritance hierarchy of storage characteristics. For more information on the inheritance hierarchy, see [“Obtaining Storage Characteristics” on page 7-12](#).

Return Values

0	The function was successful.
-1	The function was not successful.

ifx_lo_specset_flags()

Related Functions

**ifx_lo_def_create_spec(), ifx_lo_spec_free(), ifx_lo_specget_flags(),
ifx_lo_specset_extsz(), ifx_lo_specset_flags(), ifx_lo_specset_maxbytes(),
ifx_lo_specset_sbspace()**

ifx_lo_specset_maxbytes()

The **ifx_lo_specset_maxbytes()** function sets the maximum size for a smart large object.

Syntax

```
int ifx_lo_specset_maxbytes(LO_spec, maxbytes)
    ifx_lo_create_spec_t *LO_spec;
    ifx_int8_t *maxbytes;
```

LO_spec is a pointer to the LO-specification structure in which to save the maximum size. For information on the LO-specification structure, see [page 7-7](#).

maxbytes is a pointer to an **ifx_int8_t** structure that contains the maximum number of bytes for the smart large object.

Usage

Universal Server 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 on the maximum size, see [Figure 7-1 on page 7-9](#).

Return Values

0	The function was successful.
-1	The function was not successful.

Related Functions

ifx_lo_def_create_spec(), **ifx_lo_specget_maxbytes()**,
ifx_lo_specset_estbytes(), **ifx_lo_specset_extsz()**, **ifx_lo_specset_flags()**,
ifx_lo_specset_sbspace(), **ifx_lo_spec_free()**

ifx_lo_specset_sbospace()

The **ifx_lo_specset_sbospace()** function sets the name of the sbospace for a smart large object.

Syntax

```
int ifx_lo_specset_sbospace(LO_spec, sboospace_name)
    ifx_lo_create_spec_t *LO_spec;
    char *sboospace_name;
```

sboospace_name is a pointer to a buffer that contains the name of the sbospace in which to store the smart large object.

LO_spec is a pointer to the LO-specification structure in which to save the sbospace name. For information on the LO-specification structure, see [page 7-7](#).

Usage

The name of the sbospace can be at most 18 characters long. It must also be null terminated.

If you do not specify an *sboospace_name* when you create a new smart large object, Universal Server obtains the sbospace name from either the column information or from the SBSPACENAME parameter of the ONCONFIG file. For more information, see [“Obtaining Storage Characteristics” on page 7-12](#).

Return Values

0	The function was successful.
-1	The function was not successful.

Related Functions

ifx_lo_def_create_spec(), **ifx_lo_specget_sbospace()**,
ifx_lo_specset_estbytes(), **ifx_lo_specset_extsz()**, **ifx_lo_specset_flags()**,
ifx_lo_specset_maxbytes(), **ifx_lo_spec_free()**

ifx_lo_stat()

The **ifx_lo_stat()** function returns information about the current status of an open smart large object.

Syntax

```
int ifx_lo_stat(LO_fd, LO_stat)
           int LO_fd;
           ifx_lo_stat_t **LO_stat;
```

LO_fd is the LO file descriptor for the open smart large object whose status information you wish to obtain. For more information on an LO file descriptor, see [page 7-23](#).

LO_stat is 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 on the LO-status structure, see [page 7-30](#).

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 ESQL/C accessor functions for the LO-status structure. For more information on the status information and the corresponding accessor functions, see [Figure 7-7 on page 7-31](#).

Use the **ifx_lo_stat_free()** function to deallocate an LO-status structure.

Return Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

ifx_lo_stat()

Related Functions

**ifx_lo_create(), ifx_lo_open(), ifx_lo_stat_atime(), ifx_lo_stat_cspec(),
ifx_lo_stat_ctime(), ifx_lo_stat_free(), ifx_lo_stat_mtime_sec(),
ifx_lo_stat_refcnt(), ifx_lo_stat_size()**

ifx_lo_stat_atime()

The **ifx_lo_stat_atime()** function returns the time of last access for a smart large object.

Syntax

```
int ifx_lo_stat_atime(LO_stat)
    ifx_lo_stat_t *LO_stat;
```

LO_stat is a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information on the LO-status structure, see [page 7-30](#).

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 Values

- >=0 The last-access time for the smart large object that *LO_stat* identifies.
- 1 The function was not successful.

Related Functions

ifx_lo_stat(), **ifx_lo_stat_cspec()**, **ifx_lo_stat_ctime()**, **ifx_lo_stat_free()**, **ifx_lo_stat_mtime_sec()**, **ifx_lo_stat_refcnt()**, **ifx_lo_stat_size()**

ifx_lo_stat_cspect()

The **ifx_lo_stat_cspect()** function returns the LO-specification structure for a smart large object.

Syntax

```
ifx_lo_create_spec_t *ifx_lo_stat_cspect(LO_stat)
ifx_lo_stat_t *LO_stat;
```

LO_stat is a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information on the LO-status structure, see [page 7-30](#).

Usage

The **ifx_lo_stat_cspect()** function returns a pointer to an LO-specification structure that 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. To access storage characteristics from this structure, use the ESQL/C accessor functions for the LO-specification structure. For more information, see [page 7-7](#).

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_cspect()** with a call to **ifx_lo_stat()**.

Return Values

A valid pointer to an LO-specification (ifx_lo_create_spec_t) structure	The function was successful.

-1	The function was not successful.

Related Functions

**ifx_lo_stat(), ifx_lo_stat_atime(), ifx_lo_stat_ctime(), ifx_lo_stat_free(),
ifx_lo_stat_mtime_sec(), ifx_lo_stat_refcnt(), ifx_lo_stat_size()**

ifx_lo_stat_ctime()

The **ifx_lo_stat_ctime()** function returns the time of last change in status for a smart large object.

Syntax

```
int ifx_lo_stat_ctime(LO_stat)
    ifx_lo_stat_t *LO_stat;
```

LO_stat is a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information on the LO-status structure, see [page 7-30](#).

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 Values

>=0	The last-change time for the smart large object that <i>LO_stat</i> identifies.
-1	The function was not successful.

Related Functions

ifx_lo_stat(), **ifx_lo_stat_atime()**, **ifx_lo_stat_cspec()**, **ifx_lo_stat_free()**, **ifx_lo_stat_mtime_sec()**, **ifx_lo_stat_refcnt()**, **ifx_lo_stat_size()**

ifx_lo_stat_free()

The **ifx_lo_stat_free()** function frees an LO-status structure.

Syntax

```
int ifx_lo_stat_free(LO_stat)
    ifx_lo_stat_t *LO_stat;
```

LO_stat is a pointer to an LO-status structure that the **ifx_lo_stat()** function has allocated. For more information on the LO-status structure, see [page 7-30](#).

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 Values

0	The function was successful.
-1	The function was not successful.

Related Functions

ifx_lo_stat()

ifx_lo_stat_mtime_sec()

The **ifx_lo_stat_mtime_sec()** function returns the time of last modification for a smart large object.

Syntax

```
int ifx_lo_stat_mtime_sec(LO_stat)
    ifx_lo_stat_t *LO_stat;
```

LO_stat is a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information on the LO-status structure, see [page 7-30](#).

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 Values

- >=0 The last-modification time for the smart large object that *LO_stat* identifies.
- 1 The function was not successful.

Related Functions

ifx_lo_stat(), **ifx_lo_stat_atime()**, **ifx_lo_stat_cspec()**, **ifx_lo_stat_ctime()**, **ifx_lo_stat_free()**, **ifx_lo_stat_refcnt()**, **ifx_lo_stat_size()**

ifx_lo_stat_refcnt()

The **ifx_lo_stat_refcnt()** function returns the number of references to a smart large object.

Syntax

```
int ifx_lo_stat_refcnt(LO_stat)
    ifx_lo_stat_t *LO_stat;
```

LO_stat is a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information on the LO-status structure, see [page 7-30](#).

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 Values

- `>=0` The reference count for the smart large object that *LO_stat* identifies.
- `-1` The function was not successful.

Related Functions

**`ifx_lo_stat()`, `ifx_lo_stat_atime()`, `ifx_lo_stat_cspec()`, `ifx_lo_stat_ctime()`,
`ifx_lo_stat_free()`, `ifx_lo_stat_mtime_sec()`, `ifx_lo_stat_size()`**

ifx_lo_stat_size()

The **ifx_lo_stat_size()** function returns the size, in bytes, of a smart large object.

Syntax

```
int ifx_lo_stat_size(LO_stat, size)
    ifx_lo_stat_t *LO_stat;
    ifx_int8_t *size;
```

LO_stat is a pointer to an LO-status structure that **ifx_lo_stat()** allocates and fills in with status information. For more information on the LO-status structure, see [page 7-30](#).

size is 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 Values

- 0 The function was successful.
- 1 The function was not successful.

Related Functions

ifx_lo_stat(), **ifx_lo_stat_atime()**, **ifx_lo_stat_cspec()**, **ifx_lo_stat_ctime()**, **ifx_lo_stat_free()**, **ifx_lo_stat_mtime_sec()**, **ifx_lo_stat_refcnt()**

ifx_lo_tell()

The **ifx_lo_tell()** function returns the current file or seek position for an open smart large object.

Syntax

```
int ifx_lo_tell(LO_fd, seek_pos)  
    int LO_fd;  
    ifx_int8_t *seek_pos;
```

LO_fd is the LO file descriptor for the open smart large object whose seek position you wish to determine. For more information on an LO file descriptor, see [page 7-23](#).

seek_pos is 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 Values

0	The function was successful.
<0	The function was not successful and the return value indicates the cause of the failure.

Related Functions

ifx_lo_readwithseek(), ifx_lo_seek(), ifx_lo_writewithseek()

ifx_lo_to_buffer()

The **ifx_lo_to_buffer()** function copies a specified number of bytes from a smart large object into a user-defined buffer.

Syntax

```
int ifx_lo_to_buffer(LO_ptr, size, buf_ptr)
    fx_lo_t *LO_ptr;
    char **buf_ptr;
```

<i>LO_ptr</i>	is the LO-pointer structure for the smart large object from which you want to copy the data.
<i>size</i>	is an integer that identifies the number of bytes to copy from the smart large object
<i>buf_ptr</i>	is a doubly indirect pointer to a user-defined buffer to which you want to copy the data.
<i>error</i>	contains the address of the integer 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.

ifx_lo_to_buffer()

Return Values

- 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.

ifx_lo_truncate()

The **ifx_lo_truncate()** function truncates a smart large object at a specified byte position.

Syntax

```
int ifx_lo_truncate(LO_fd, offset)
    int LO_fd;
    ifx_int8_t *offset;
```

LO_fd is the LO file descriptor for the open smart large object whose value you wish to truncate. For more information on an LO file descriptor, see [page 7-23](#).

offset is 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 Values

- 0 The function was successful.
- <0 The function was not successful and the return value indicates the cause of the failure.

ifx_lo_write()

The **ifx_lo_write()** function writes a specified number of bytes of data to an open smart large object.

Syntax

```
int ifx_lo_write(LO_fd, buf, nbytes, error)
    int LO_fd;
    char *buf;
    int nbytes;
    int *error;
```

LO_fd is the LO file descriptor for the smart large object to which to write. For more information on an LO file descriptor, see [page 7-23](#).

buf is a pointer to a buffer that contains the data that the function writes to the smart large object.

nbytes is the number of bytes to write to the smart large object. This value cannot exceed 2 gigabytes.

error is a pointer to an integer 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 Values

- >=0 The number of bytes that the function has written from the *buf* character buffer to open the smart large object.
- 1 The function was not successful; examine the *error* for a detailed error code.

Related Functions

**ifx_lo_open(), ifx_lo_read(), ifx_lo_seek(), ifx_lo_tell(),
ifx_lo_writewithseek()**

ifx_lo_writewithseek()

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

```
int ifx_lo_writewithseek(LO_fd, buf, nbytes, offset, whence, error)
    int LO_fd;
    char *buf;
    int nbytes;
    ifx_int8_t *offset;
    int whence;
    int *error;
```

<i>LO_fd</i>	is the LO file descriptor for the smart large object to which to write. For more information on an LO file descriptor, see page 7-23 .
<i>buf</i>	is a pointer to a buffer that contains the data that the function writes to the smart large object.
<i>nbytes</i>	is the number of bytes to write to the smart large object. This value cannot exceed 2 gigabytes.
<i>offset</i>	is a pointer to the 8-byte integer (INT8) offset from the starting seek position.
<i>whence</i>	is an integer value that identifies the starting seek position.
<i>error</i>	is a pointer to an integer 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 on how to access eight-bit (INT8) integers, see [“The int8 Data Type” on page 5-7](#).

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 Values

- >=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

ifx_lo_open(), ifx_lo_readwithseek(), ifx_lo_seek(), ifx_lo_write()

The create_clob 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

In this example, assume that the **sb1** sbspace has the following storage characteristics defined:

- Logging is on
- Do not keep last access time
- Integrity is high
- Extent size is 1000 kilobytes

Assume that the **cat_descr** column has the following storage characteristics defined:

- Logging is on (default from **sb1** sbspace)
- Keep last access time
- Integrity is high (default from **sb1** sbspace)
- Extent size is 100 kilobytes

The **create_clob** program creates the **cat_descr** smart large object that has the following user-defined storage characteristics:

- Logging is on: LO_LOG
- Keep last access time (default from **cat_descr** column):
LO_KEEP_ACCESSTIME
- Integrity is high (default from **sb1** sbspace)
- Allocation extent size is 10 kilobytes

Source for the create_clob.ec File

```
EXEC SQL include int8;
EXEC SQL include locator;

EXEC SQL define BUFSZ 10;

extern char statement[30];

main()
{
EXEC SQL BEGIN DECLARE SECTION;
    long catalog_number;
    int error;
    int numbytes;
    char buf[BUFSZ];
    char svr_name[256];
    char col_name[300];
    int buflen = BUFSZ;
    int8 estbytes, offset;
    ifx_lo_create_spec_t *create_spec;
    fixed binary 'blob' ifx_lo_t picture;
    int lofd;
EXEC SQL END DECLARE SECTION;

    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
        :svr_name = server_name;

/* Set up a test table called catalog2 in the demonstration
 * database.
 */
    strcpy(statement, "CREATE DISTINCT TYPE stmt");
    EXEC SQL create distinct type fpicture as clob;
    strcpy(statement, "CREATE CAST to CLOB stmt");
    EXEC SQL create cast (fpicture as clob);
    strcpy(statement, "CREATE CAST from CLOB stmt");
    EXEC SQL create cast (clob as fpicture);
```

Source for the create_clob.ec File

```
/* Assign the following column-level storage characteristics
 * for the cat_desc column:
 *     sbspace name = sb1 (this sbspace must already exist)
 *     extent size = 100 kilobytes
 *     keep last access time is on
 */
strcpy(statement, "CREATE TABLE stmt");
EXEC SQL create table catalog2
(
    catalog_num INTEGER,
    cat_descr fpicture
) put cat_descr in (sb1)
(extent size = 100,
keep access time);

/* Allocate and initialize the L0-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, cat_descr.
 */
sprintf(col_name, "stores7@s:catalog2.cat_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
 * cat_desc with the following user-defined storage
 * characteristics:
 *     no logging
 *     extent size = 10 kilobytes
 */
ifx_lo_specset_flags(create_spec, L0_LOG);
ifx_int8cvint(BUFSZ, &estbytes);
ifx_lo_specset_estbytes(create_spec, &estbytes);
```

```
/* Create an LO-pointer structure for the smart large object
*/
    if (lofd = ifx_lo_create(create_spec, LO_RDWR,
        &picture, &error) == -1)
    {
        strcpy(statement, "ifx_lo_create()");
        handle_lo_error(error);
    }

/* Copy data into the character buffer 'buf' */
    sprintf(buf, "%s %s %s"
        "The rain in Spain stays mainly in the plain.",
        "In Hartford, Hereford, and Hampshire, hurricanes",
        "hardly happen.");

/* 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 catalog2");
    EXEC SQL insert into catalog2 values (5, :picture);

/* 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.
*/
    strcpy(statement, "SELECT FROM catalog2");
    EXEC SQL select cat_descr into :picture from catalog2;

/* 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);
    }
```

Source for the create_clob.ec File

```
/* 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);
    }

    printf("Contents of CLOB column:\n %s", 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);
}

/* Include source code for whenexp_chk() exception-checking
 * routine
 */
EXEC SQL include exp_chk.ec;
```

Working with Simple Large Objects

Choosing a Large-Object Data Type	8-4
Programming with Simple Large Objects	8-5
Declaring a Host Variable for a Simple Large Object	8-5
Accessing the Locator Structure	8-6
The Fields of the Locator Structure.	8-8
Locations for Simple-Large-Object Data	8-10
Locating Simple Large Objects in Memory.	8-11
Allocating the Memory Buffer.	8-12
A Memory Buffer That the ESQL/C Libraries Allocate	8-13
A Memory Buffer That the Program Allocates	8-14
Selecting a Simple Large Object into Memory	8-15
Inserting a Simple Large Object from Memory	8-17
Locating Simple Large Objects in Files	8-19
Locating Simple Large Objects in Open Files.	8-21
Selecting a Simple Large Object into an Open File	8-22
Inserting a Simple Large Object from an Open File	8-23
Locating Simple Large Objects in Named Files	8-26
Selecting a Simple Large Object into a Named File	8-27
Inserting a Simple Large Object from a Named File	8-28
Using User-Defined Simple-Large-Object Locations	8-31
Selecting a Simple Large Object into a User-Defined Location.	8-33
Inserting a Simple Large Object into a User-Defined Location.	8-34

Creating the User-Defined Simple-Large-Object Functions	8-35
The User-Defined Open Function	8-35
The User-Defined Read Function	8-36
The User-Defined Write Function	8-37
The User-Defined Close Function	8-39
Reading and Writing Simple Large Objects to an Optical Disc	8-40
The dispcat_pic Program	8-44
Loading the Simple-Large-Object Images	8-45
Choosing the Image Files	8-45
Using the blobload Utility	8-46
Compiling the dispcat_pic Program	8-48
Avoiding Display of Images	8-48
Displaying Sun Raster Images	8-49
Guide to the dispcat_pic.ec File	8-50
Guide to the prdesc.c File	8-66
Guide to the inpfuns.c File.	8-68

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.

This chapter covers the following topics for simple large objects:

- Choosing whether to use a simple large object or a smart large object in your 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** database.

For information about the data types available in an INFORMIX-ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#) For information about the TEXT and BYTE data types, as well as other SQL data types, see the *[Informix Guide to SQL: Reference.](#)*

Choosing a Large-Object Data Type

If you use INFORMIX-Universal Server as your database server, you can choose between using simple large objects or smart large objects.

INFORMIX-Universal Server supports simple large objects primarily for backward compatibility with existing 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 <i>all or nothing</i> basis.	Have an interface that provides access like an operating-system file; you can access only specified portions of the smart large object.
Writing the large object	The database server updates a simple large object on an <i>all or nothing</i> basis.	Update to a portion of a smart large object does not cause the database server to rewrite the entire smart large object.
Data logging	Data logging always on	Data logging can be turned off and on

For more information on how to use smart large objects, see [Chapter 7, “Working with INFORMIX-Universal Server Smart Large Objects.”](#) The remainder of this chapter discusses how to use simple large objects. ♦



Programming with Simple Large Objects

ESQL/C supports the SQL simple large objects, the data types TEXT and BYTE, with the **loc_t** data type.

Tip: You cannot use literal values within an INSERT or UPDATE statement to put simple-large-object data within 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 within an ESQL/C client application.

Because of the potentially huge size of simple-large-object data, the 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 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 on the fields of the locator structure, see [“The Fields of the Locator Structure” on page 8-8](#).



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 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 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.
- 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 8-1 shows the definition of the **loc_t** locator structure from the **locator.h** file.

```
typedef struct tag_loc_t
{
    short loc_loctype;          /* USER: type of locator - see below */
    union                      /* variant on 'loc' */
    {
        struct                /* case LOCMEMORY */
        {
            long lc_bufsize;   /* USER: buffer size */
            char *lc_buffer;    /* USER: memory buffer to use */
            char *lc_currddata_p; /* INTERNAL: current memory buffer */
            int lc_mflags;      /* USER/INTERNAL: memory flags */
                                /* (see below) */
        } lc_mem;

        struct                /* cases LOCFNAME & LOCFILE */
        {
            char *lc_fname;     /* USER: file name */
            int lc_mode;        /* USER: perm. bits used if creating */
            int lc_fd;          /* USER: os file descriptor */
            long lc_position;    /* INTERNAL: seek position */
        } lc_file;
    } lc_union;

    long loc_indicator;        /* USER/SYSTEM: indicator */
    long loc_type;             /* SYSTEM: type of blob */
    long loc_size;             /* USER/SYSTEM: num bytes in blob or -1 */
    int loc_status;            /* SYSTEM: status return of locator ops */
    char *loc_user_env;        /* USER: for the user's PRIVATE use */
    long loc_xfercount;        /* INTERNAL/SYSTEM: Transfer count */

    int (*loc_open)();         /* USER: open function */
    int (*loc_close)();        /* USER: close function */
    int (*loc_read)();         /* USER: read function */
    int (*loc_write)();        /* USER: write function */
    int loc_oflags;            /* USER/INTERNAL: see flag definitions below */
} loc_t;
```

Figure 8-1
*Declaration of loc_t
in the locator.h
Header File*

In [Figure 8-1 on page 8-7](#), the following comments in the **locator.h** file indicate how the fields are used in the locator structure.

Comment	Description
USER	The ESQL/C program sets the field, and the ESQL/C libraries inspect the field.
SYSTEM	The ESQL/C libraries set the field, and the ESQL/C program inspects the field.
INTERNAL	The field is a work area for the ESQL/C libraries, and the ESQL/C program does <i>not</i> need to examine the field.

ESQL/C does *not* automatically include the **locator.h** header file in an ESQL/C program. You must include the **locator.h** header file in any 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 on **loc_loctype**, see [“Locations for Simple-Large-Object Data” on page 8-10](#).
- The **lc_union** structure is a **union** (overlapping variant structures) structure.
The variant in use depends on where ESQL/C can expect to find the simple large object at runtime. For more information on this structure, see [“Locating Simple Large Objects in Memory” on page 8-10](#) and [“Locating Simple Large Objects in Files” on page 8-19](#).
- Several fields are common to all types of simple-large-object variables.

Figure 8-2 lists the fields in the locator structure common to all simple-large-object locations.

Figure 8-2
Fields in Locator Structure Common to all Simple-Large-Object Data Locations

Field	Data Type	Description
loc_indicator	long	A value of -1 in the loc_indicator field indicates a null simple-large-object value. The ESQL/C program can set the field to indicate insertion of a null value; ESQL/C libraries set it on a select or fetch. 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	long	Contains the size of the simple-large-object data in bytes. This field indicates the amount of simple-large-object data that the ESQL/C libraries read or write. The ESQL/C program sets loc_size when it inserts a simple large object in the database; the ESQL/C libraries set loc_size after it selects or fetches a simple large object.
loc_status	int	Indicates the status of the last locator operation. The 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	long	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 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. Figure 8-3 shows the possible locations for simple-large-object data.

Figure 8-3
Possible Locations for Simple-Large-Object Data

Value of loc_loctype Field	Location of Simple-Large-Object Data	Reference
LOCMEMORY	In memory	page 8-10
LOCFILE	In an open file	page 8-21
LOCFNAME	In a named file	page 8-26
LOCUSER	At a user-defined location	page 8-31

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 ESQL / C program, use these constant names rather than their constant values when you assign values to **loc_loctype**.

In a client-server environment, 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 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. Figure 8-4 summarizes the **lc_union.lc_mem** fields.

Figure 8-4

Fields in lc_union.lc_mem Structure Used for Simple Large Objects Located in Memory

Field	Data Type	Description
lc_bufsize	long	The size, in bytes, of the buffer to which the lc_buffer field points. For more information, see “Allocating the Memory Buffer” on page 8-12 .
lc_buffer	char *	The address of the buffer to hold the simple large-object value. Your 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 8-12 .
lc_currdata_p	char *	The address of the system buffer. <i>This is an internal field and must not be modified by the ESQL/C program.</i>
lc_mflags	int	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: Informix recommends that you use these shortcut names when you access the locator structure. The shortcut names improve code readability and reduce coding errors. This manual 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 ESQL/C programs that demonstrate how to handle simple-large-object data located in an open file:

- The **getcd_me.ec** program selects a simple large object into memory.
- The **updc_d_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.ec** and **updc_d_me.ec** programs are briefly explained on [page 8-15](#) and [page 8-17](#), respectively.

Allocating the Memory Buffer

When your program selects simple-large-object data into memory, 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 ESQL/C allocates this memory buffer:

- If you set the **loc_bufsize** to -1, 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, 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, ESQL/C allocates the memory buffer on a fetch or select. 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, ESQL/C sets the **loc_status** field to -465 to indicate an error.) When the select (or the *first* fetch) completes, 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 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, 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 with the **free()** system call.

When you set **loc_mflags** to LOC_ALLOC, ESQL/C handles memory allocation as follows:

- If the size of the simple-large-object data *increases*, ESQL/C frees the existing buffer and allocates the necessary memory.
If this reallocation occurs, 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. 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*, 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.

ESQL/C frees the allocated memory when it fetches the next simple-large-object value. Therefore, 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 8-15.](#)

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 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, 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. 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 8-5 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 8-5
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 ESQL/C returns the **cat_descr** text in a memory buffer.
- The **loc_bufsize** field is set to -1 to have ESQL/C allocate the memory for the buffer. For more information, see [“A Memory Buffer That the ESQL/C Libraries Allocate” on page 8-12](#).
- 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 8-12](#).

The program in [Figure 8-5 on page 8-15](#) 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 8-66](#). 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 program in [Figure 8-5 on page 8-15](#) displays the **cat_descr** column for a catalog number that the user enters. Figure 8-6 shows the user input and the output that results from the **cat_descr** column of the **stores7** 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 8-6
*Sample Output from
the getcd_me
Sample Program*

Inserting a Simple Large Object from Memory

The **updc_d_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 8-7 shows sample output as the user updates the **cat_descr** column of the **stores7** database.

Figure 8-7

*Sample Output from
the updc_d_me
Sample Program*

```
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 8-8 shows a code excerpt that illustrates how the **updc_d_me** program uses the locator structure to update the **cat_descr** column from the text that is stored in memory.

Figure 8-8

*Code Excerpt from
the updc_d_me
Sample Program*

```
/* 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) + 1;
    /* Update */
    EXEC SQL update catalog
        set cat_descr =:cat_descr
        where catalog_num =:cat_num;

    :
}
}
```

The program sets the **cat_descr** locator structure fields as follows:

- The **loc_loctype** field is set to **LOCMEMORY** so that 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 8-9 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 8-9
*Sample INSERT
Operation from
Primary Memory*

After the UPDATE or INSERT statement, 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 8-12](#).

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 filename 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. Figure 8-10 summarizes the **lc_union.lc_file** fields.

Figure 8-10
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 pathname 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 , refer to 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	long	The current seek position within the opened file. <i>This is an internal field and must not be modified by the ESQL/C program.</i>



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: Informix recommends that you use these shortcut names when you access the locator structure. The shortcut names improve code readability and reduce coding errors. This manual uses these shortcut names when it refers to the **lc_fname**, **lc_fd**, and **lc_position** fields of the **lc_union.lc_file** structure.

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_RDONLY** 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.

One of these flags is passed to the **loc_open()** function when ESQL/C opens the file. ESQL/C reads the data and writes it to the current location (which the **loc_position** field indicates) in the file. If 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, respectively. If ESQL/C is unable to close a file, it sets **loc_status** to -462. ESQL/C updates the **SQLCODE** variable with this same value.

Locating Simple Large Objects in Open Files

To have 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 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 ESQL/C how to access the file when it opens it. For a list of file-open mode flags, see [page 8-20](#).

The **demo** directory contains the following two sample 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 **updc_d_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 8-11 shows a code excerpt that selects the **cat_descr** column into a file that the user specifies.

```
EXEC SQL BEGIN DECLARE SECTION;
    long 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 8-11
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 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 within 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 within the **locator.h** file.

After 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 [page 8-20](#).

Inserting a Simple Large Object from an Open File

The **upcdcd_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 8-12 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;
    long cat_num;
    loc_t cat_descr;
EXEC SQL END DECLARE SECTION;
:
:
if ((fd = open(descf1, 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_RDONLY;       /* 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 8-12
*Code Excerpt from
the updc_d_of
Sample Program*

The **upddcd_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 **upddcd_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 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_RDONLY**, 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 **upddcd_of** program is then able to perform the database update. After ESQL/C reads data from the open file and sends it to the database server, ESQL/C updates the **loc_size** field with the number of bytes read from the open file and sent to the database server. 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 [page 8-20](#).

Locating Simple Large Objects in Named Files

To have 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 ESQL/C program must specify a pointer to the filename 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 ESQL/C how to access the file when it opens it. For a list of file-open mode flags, see [page 8-20](#).

To open a named file, 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, ESQL/C creates it. 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 ESQL/C cannot open this file, it sets the **loc_status** field (and SQLCODE) to -461. When the transfer is complete, ESQL/C closes the file, which releases the file descriptor in the **loc_fd** field.

The **demo** directory contains the following two sample 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 **updc_d_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 8-13 shows a code excerpt from the **getcd_nf** sample program.

Figure 8-13

*Code Excerpt from
the getcd_nf Sample
Program*

```
EXEC SQL BEGIN DECLARE SECTION;
    long 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");
}
```

The program sets the **cat_descr** locator structure fields as follows:

- The **loc_loctype** field contains LOCFNAME to tell 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 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 ESQL/C to append selected data to the existing file.

The **getcd_nf** program then executes the SELECT statement to retrieve the row. After 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 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 [page 8-20](#).

Inserting a Simple Large Object from a Named File

The **upcd_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 8-14 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;
    long 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_RDONLY;      /* 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 8-14
Code Excerpt from
the *upcd_nf*
Sample Program

The **updcn_nf** program in [Figure 8-14 on page 8-29](#) 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 ([page 8-66](#)) 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 **updcn_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_RDONLY** to tell **ESQL/C** to open the file in read-only mode.
- The **cat_descr.loc_size** field is set to **-1** to tell **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 **ESQL/C** reads data from the named file and sends it to the database server, **ESQL/C** updates the **loc_size** field with the number of bytes read from the named file and sent to the database server. **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. See [page 8-20](#) for possible causes of the error.

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 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;
      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 Figure 8-15 summarizes.

Figure 8-15
Fields in the Locator Structure Used to Create User-Defined Location Functions

Field	Data Type	Description
loc_open	int (*)()	A pointer to a user-defined open function that returns an integer value. For more information, see “The User-Defined Open Function” on page 8-35 .
loc_read	int (*)()	A pointer to a user-defined read function that returns an integer value. For more information, see “The User-Defined Read Function” on page 8-36 .
loc_write	int (*)()	A pointer to a user-defined write function that returns an integer value. For more information, see “The User-Defined Write Function” on page 8-37 .

(1 of 2)

Field	Data Type	Description
loc_close	int (*)()	A pointer to a user-defined close function that returns an integer value. For more information, see “The User-Defined Close Function” on page 8-39 .
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	long	The number of bytes that the last transfer operation for the simple large object transferred.

(2 of 2)

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. [Figure 8-4 on page 8-11](#) and [Figure 8-10 on page 8-19](#) summarize fields of the **lc_union.lc_mem** structure and **lc_union.lc_file** structure, respectively.

Selecting a Simple Large Object into a User-Defined Location

When your program *selects* a simple-large-object value, the ESQL/C libraries must receive the data from the database server and transfer it to the ESQL/C program. To do this, ESQL/C performs the following steps:

1. Before the transfer, 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. ESQL/C receives the simple-large-object value from the database server and puts it into a program buffer.
3. ESQL/C calls the user-defined write function to transfer the simple-large-object data from the program buffer to the user-defined location.

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, ESQL/C performs the clean-up operations specified in the user-defined close function.

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 ESQL/C libraries must transfer the data from the ESQL/C program to the database server. To do this, ESQL/C performs the following steps:

1. Before the transfer, 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_RDONLY.
2. 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. ESQL/C sends the value in the program buffer to the database server. 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, 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 ESQL/C to send the entire user-defined simple-large-object location 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

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 **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, 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, **loc_t *loc_struct**, where *loc_struct* 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_ONLY** if ESQL/C calls the open function to send the simple large object to the database, or **LOC_WONLY** if 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 8-16 shows a skeleton function of a user-defined open function.

```
open_simple_lo(adloc, oflags)
loc_t *adloc;
int oflags;
{
    adloc->loc_status = 0;
    adloc->loc_xfercount = 0L;
    if (0 == (oflags & adloc->loc_oflags))
        return(-1);
    if (oflags & LOC_RDONLY)
        /** prepare for store to db */
    else
        /** prepare for fetch to program */
    return(0);
}
```

Figure 8-16
*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 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, ESQL/C calls the user-defined read function. Your 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, **loc_t *loc_struct**, where *loc_struct* 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. 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 has been 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 8-17 shows a skeleton function of a user-defined read function.

```
read_simple_lo(adloc, bufp, ntoread)
loc_t *adloc;
char *bufp;
int ntoread;
{
    int ntoxfer;

    ntoxfer = ntoread;
    if (adloc->loc_size != -1)
        ntoxfer = min(ntoread,
            adloc->loc_size - adloc->loc_xfercount);

    /** transfer "ntoread" bytes to *bufp ***/

    adloc->loc_xfercount += ntoxfer;
    return(ntoxfer);
}
```

Figure 8-17
*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 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, ESQL/C calls the user-defined write function. Your 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, **loc_t** **loc_struct*, where *loc_struct* 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. 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 has been 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 8-18 shows a skeleton function of a user-defined write function.

```
write_simple_lo(adloc, bufp, ntowrite)
loc_t *adloc;
char *bufp;
int ntowrite;
{
    int xtofer;

    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 8-18
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, 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, **loc_t** **loc_struct*, where *loc_struct* 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 8-19 shows a skeleton function of a user-defined close function.

```
close_simple_lo (adloc)
loc_t *adloc;
{
    adloc->loc_status = 0;
    if (adloc->loc_oflags & LOC_WONLY) /* if fetching */
    {
        adloc->loc_indicator = 0; /* clear indicator */
        adloc->loc_size = adloc->loc_xfercount;
    }
    return(0);
}
```

Figure 8-19
*A Sample
User-Defined Close
Function*

UNIX

IUS

ODS

Reading and Writing Simple Large Objects to an Optical Disc

Only Universal Server and INFORMIX-OnLine Dynamic Server, on the UNIX operating system, can use optical discs.

Within a table, rows that include simple-large-object data do not include the simple-large-object data in the row 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 blobpage blobpage, or a platter in an optical storage subsystem. For details, see the [Administrator's Guide](#) for your database server and the [INFORMIX-OnLine/Optical User Manual](#).

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 8-20 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.

Figure 8-20

Code Fragment to Retrieve the Simple-Large-Object Descriptor

```
#include <stdio.h>
EXEC SQL include locator;

char errmsg[400];

EXEC SQL BEGIN DECLARE SECTION;
    long cat_num;
    short stock_num;
    char manu_code[4];
    loc_t cat_descr;
    loc_t cat_picture;
EXEC SQL END DECLARE SECTION;

main(argc, argv)
int 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);

    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);
}
/* 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);
}
```

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 8-21 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 8-21
*Using DESCR() to
Access a Simple-
Large-Object
Descriptor*



The *dispcat_pic* Program

The **dispcat_pic** demo program is only implemented on UNIX platforms.

The **dispcat_pic** program, annotated on the following pages, uses the ESQL/C **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 on how to create the demonstration database, see “[Demonstration Database](#)” on page 5 in 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 temporary file.

If you compile the program with the -DSUNVIEW option, the program forks a second process and executes the Sun **screenload** system utility to display the raster image from the temporary file. In all cases, 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 ESQL/C platform. Finally, the program deletes the temporary file it created for **cat_picture** and allows the user to enter another **catalog_num** value or terminate the program.

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. If you display the simple large objects on a Sun workstation, you specify a command-line definition to perform conditional compilation.

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 ESQL/C demonstration directory provides five graphic images and the **blobload** program to load five rows of the **catalog** table with graphic images that can be displayed.

To display these simple-large-object images from the **dispcat_pic** program, you must load the images to the **catalog** table.

To load the images in the catalog table

1. Choose the appropriate simple-large-object images for your graphics environment.
2. Use the **blobload** utility to load the simple-large-object images into the **cat_picture** column of the **catalog** table.

Choosing the Image Files

Informix provides the five **cat_picture** images in two formats:

- Sun raster image format files, which have the **.rs** file extension.
If you compile **dispcat_pic.ec** with the conditional display logic, you must use the Sun raster image files to load the five **cat_picture** columns. The second column of [Figure 8-22 on page 8-46](#) shows the names of the raster image files for the simple-large-object images.
- Graphics Interchange Format files, which have the **.gif** file extension.
ESQL/C also provides the images in **.gif** files to provide them in a standard format that can be displayed on other platforms or translated into other formats with filter programs that other vendors supply. The third column of [Figure 8-22](#) shows the names of the **.gif** files for the simple-large-object images.

Figure 8-22
Image Files for Simple-Large-Object Demo

Image	Sun Raster Image Format (.rs Files)	Graphics Interchange Format (.gif Files)
Baseball glove	cn_10001.rs	cn_10001.gif
Bicycle crankset	cn_10027.rs	cn_10027.gif
Bicycle helmet	cn_10031.rs	cn_10031.gif
Golf balls	cn_10046.rs	cn_10046.gif
Running shoe	cn_10049.rs	cn_10049.gif

The numeric portion of the image filename is the **catalog_num** value for the row of the **catalog** table to which the image is to be updated. For example, **cn_10027.rs** should be updated to the **cat_picture** column of the row where 10027 is the value of **catalog_num**.



Important: The **dispcat_pic** program, as delivered, only displays Sun raster images. To display the Graphics Interchange Format, or any other format, you must modify the **display_picture()** function of **dispcat_pic**, substituting display logic for logic that can use the format of your image files. For more information, see [“Compiling the dispcat_pic Program” on page 8-48](#).

Using the blobload Utility

The **blobload** utility is an ESQL/C program that is provided as part of the 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 8-23 shows the output of this command that describes the command-line arguments that **blobload** expects.

Figure 8-23

*Sample Output from
the blobload Utility*

```
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.
```

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.rs** 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.rs -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 filename (**-f** option) and corresponding **catalog_num** value (**-k** option) of the image file that you want to load.

You can use the **blobload** program to load the **.gif** files to the **catalog** table in the same manner that you use it to load the **.rs** files.

Compiling the `dispcat_pic` Program

The **`dispcat_pic`** program is provided as part of the ESQL/C demonstration files so that you can compile and run it yourself. The program includes conditional display logic to illustrate how to display a graphic image from a BYTE type column. As provided, the program can handle the display of a BYTE column in one of the following ways:

- A single **`printf()`** call says that the program cannot display a BYTE value. To use this method to handle BYTE display, compile **`dispcat_pic`** with *no* conditional flag.
- The Sun **`screenload`** utility displays a raster image of the **`cat_picture`** column on a Sun workstation. To use this method to handle BYTE display, compile **`dispcat_pic`** with the `-DSUNVIEW` conditional flag.

The **`display_picture()`** function of **`dispcat_pic`** contains both sets of conditional compilations. The set that the ESQL/C preprocessor uses depends on whether the `SUNVIEW` definition flag has been defined.

Avoiding Display of Images

To display a simple-large-object image, the **`dispcat_pic`** program must execute the appropriate image-display utility for the graphics environment. If your applications run under some graphics environment other than a Sun workstation, **`dispcat_pic`** does not call the correct image-display utility.

Use the following command to compile the **`dispcat_pic`** program so that it does not display the BYTE images:

```
esql -o dispcat_pic dispcat_pic.ec
```

The **`-o dispcat_pic`** option causes the executable program to be named **`dispcat_pic`**. Without the **`-o`** option, the name of the executable program defaults to **`a.out`**. For more information on the **`esql`** command, see [“Using the esql Command” on page 1-47](#).

When the program is compiled *without* the `SUNVIEW` flag, the program just displays the following message when it encounters a `BYTE` value:

```
** Cannot print catalog picture.
```

To display the **cat_picture** column on a Sun workstation, you can set the `SUNVIEW` conditional flag when you compile. (For more information, see [“Displaying Sun Raster Images.”](#)) To display this `BYTE` image on another `ESQL/C` platform, you must substitute display logic that runs on that platform.

Displaying Sun Raster Images

To display the Sun raster simple-large-object images, the **display_picture()** function uses the Sun **screenload** utility program.

To compile **dispcat_pic** *with* the conditional **cat_picture** display logic for a Sun workstation, use the following command:

```
esql -o dispcat_pic -DSUNVIEW dispcat_pic.ec
```

The `-DSUNVIEW` option causes the conditional display logic to be compiled.



Tip: *For simplicity, the program is not designed to display the **cat_picture** raster image under Sun View or any other windowing environment. If you display the **cat_picture** column, for best results, run the program directly from the SunOS command line, not through the window manager.*

Guide to the `dispcat_pic.ec` File

```
1 /*
2  * dispcat_pic.ec *

3  The following program prompts the user for a catalog number and
4  displays the cat_picture column, if it is not null, for that row of
5  the catalog table.

6  WARNING: This program displays a Sun standard raster file using the
7  screenload program provided by Sun. It will display properly only
8  on a Sun display.
9 */

10 #include <stdio.h>
11 EXEC SQL include locator;

12 #define WARNNOTIFY      1
13 #define NOWARNNOTIFY    0

14 #define BUFFSZ 256

15 extern int errno;

16 EXEC SQL BEGIN DECLARE SECTION;
17   long cat_num;
18   loc_t cat_descr;
19   loc_t cat_picture;
20 EXEC SQL END DECLARE SECTION;

21 char cpfl[18];

22 main(argc, argv)
23 int argc;
24 char *argv[];
25 {
```

Continued on [page 8-52](#)

Lines 10 and 11

The **#include <stdio.h>** statement includes the **stdio.h** UNIX header file from the **/usr/include** directory. The **stdio.h** file enables **dispcat_pic** to use the standard C I/O library. The program also includes an ESQL/C header file (line 11). The **locator.h** file contains the definition of the locator structure and the constants that are needed to work with this structure.

Lines 12 to 15

The `WARNNOTIFY` and `NOWARNNOTIFY` constants (lines 12 and 13) are used 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 185 to 192 for more information about the **exp_chk2()** function.

The program uses `BUFFSZ` (line 14) to specify the size of arrays that store input from the user. Line 15 defines **errno**, an external integer where system calls store an error number.

Lines 16 to 20

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 18 and 19 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 **loc_t** typedef. The program uses the locator structure to specify simple-large-object size and location.

Line 21

Line 21 defines a single *global C variable*. The **cpfl** character array stores the name of a temporary file. This named file is the location for the simple-large-object raster image of **cat_picture** that the database server writes.

Lines 22 to 24

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.

```
26 char ans[BUFFSZ];
27 long ret, exp_chk2();
28 char db_msg[ BUFFSZ + 1 ];

29 EXEC SQL BEGIN DECLARE SECTION;
30     char db_name[20];
31     char description[16];
32 EXEC SQL END DECLARE SECTION;

33 printf("DISPCAT_PIC Sample ESQL Program running.\n\n");
34 if (argc > 2)/* correct no. of args? */
35 {
36     printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
37         argv[0]);
38     printf("DISPCAT_PIC Sample Program over.\n\n");
39     exit(1);
40 }
41 strcpy(db_name, "stores7");
42 if(argc == 2)
43     strcpy(db_name, argv[1]);
44 EXEC SQL connect to :db_name;
45 sprintf(db_msg, "CONNECT TO %s",db_name);
46 if(exp_chk2(db_msg, NOWARNNOTIFY) < 0)
47 {
48     printf("DISPCAT_PIC Sample Program over.\n\n");
49     exit(1);
50 }
```

Continued on page 8-54

Lines 26 to 32

Lines 26 to 28 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 27 defines a **long** 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 29 to 32 define the 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.

Lines 34 to 50

These lines interpret the command-line arguments and open the database. Line 34 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 41 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 43 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 44) 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 45). It does so for the sake of the **exp_chk2()** call on line 46, which takes as its argument the name of a statement. Line 46 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.

```

51 if(sqlca.sqlwarn.sqlwarn3 != 'W')
52 {
53     printf("\nThis program does not work with the INFORMIX-SE");
54     printf("Server\n");
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;

61 while(1)
62 {
63     strcpy(cpfl, "./cpfl.XXXXXX");
64     if(!mktemp(cpfl))
65     {
66         printf("** Cannot create temporary file for catalog picture.\n");
67         EXEC SQL disconnect current;
68         printf("\nDISPCAT_PIC Sample Program over.\n\n");
69         exit(1);
70     }
71     printf("\nEnter catalog number: ");/* prompt for cat. number */
72     if(!getans(ans, 6))
73         continue;
74     printf("\n");
75     if(rstol(ans, &cat_num)) /* cat_num string to long */
76     {
77         printf("** Cannot convert catalog number '%s' to integer\n",
78             ans);
79         EXEC SQL disconnect current;
80         printf("\nDISPCAT_PIC Sample Program over.\n\n");
81         exit(1);
82     }

```

Continued on [page 8-56](#)

Lines 51 to 60

IUS

ODS

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 *only* the OnLine Dynamic Server and Universal Server database servers support, line 51 checks the type of database server. If the **sqlwarn3** element of **sqlca.sqlwarn** is set to W, the established connection is to an OnLine Dynamic Server or Universal Server database server. 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). ♦

Lines 61 to 70

The **while(1)** on line 61 begins the main processing loop within **dispcat_pic**. First the loop creates a uniquely named file to receive **cat_picture**. Line 63 copies the name of the temporary file to the **cpfil[]** array. The UNIX **mktemp()** function (line 64) creates a unique filename, with **cpfil[]** as the argument. If **mktemp()** cannot create a unique filename, it returns 0; lines 66 to 69 display a message to the user and exit.

Lines 71 to 74

Line 71 prompts the user to enter a catalog number for the **cat_picture** column that the user wants to see. Line 72 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 73 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 8-68](#).

Lines 75 to 82

Line 75 calls the 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 77 to 82 display a message to the user, close the connection, and exit.

```
83      /*
84      * Prepare locator structure for select of cat_descr
85      */
86      cat_descr.loc_loctype = LOCMEMORY; /* set for 'in memory' */
87      cat_descr.loc_bufsize = -1; /* let db get buffer */
88      cat_descr.loc_mflags = 0; /* clear memory-deallocation feature */
89      cat_descr.loc_oflags = 0; /* clear loc_oflags */
90      /*
91      * Prepare locator structure for select of cat_picture
92      */
93      cat_picture.loc_loctype = LOCFNAME; /* type = named file */
94      cat_picture.loc_fname = cpfl; /* supply file name */
95      cat_picture.loc_oflags = LOC_WONLY; /* file-open mode = write */
96      cat_picture.loc_size = -1; /* size = size of file */

97      /* Look up catalog number */
98      EXEC SQL select description, catalog_num, cat_descr, cat_picture
99      into :description, :cat_num, :cat_descr, :cat_picture
100      from stock, catalog
101      where catalog_num = :cat_num
102      and catalog.stock_num = stock.stock_num
103      and catalog.manu_code = stock.manu_code;
104      if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100) /* if not found */
105      {
106          printf("** Catalog number %ld not found in ", cat_num);
107          printf("catalog table.\n");
108          printf("\t OR item not found in stock table.\n");
109          if(!more_to_do())
110              break;
111          continue;
112      }
```

Continued on page 8-58

Lines 83 to 89

These lines define the simple-large-object location for the TEXT **cat_descr** column of the **catalog** table, as follows:

- Line 86 sets **loc_loctype** in the **cat_descr** locator structure to LOCMEMORY to tell ESQL/C to select the data for **cat_descr** into memory.
- Line 87 sets **loc_bufsize** to -1 so that ESQL/C allocates a memory buffer to receive the data for **cat_descr**.
- Line 88 sets **loc_mflags** to 0 to disable the memory-deallocation feature (see Line 138) of ESQL/C.

If the select is successful, ESQL/C returns the address of the allocated buffer in **loc_buffer**. Line 89 sets the **loc_oflags** file-open mode flags to 0 because the program retrieves the simple large object into memory rather than a file.

Lines 90 to 96

These lines prepare the locator structure to retrieve the BYTE **cat_picture** of the **catalog** table. Line 93 moves LOCFNAME to **loc_loctype** to tell ESQL/C to locate the data for **cat_descr** in a named file. Line 94 moves the address of the **cpfl** filename into **loc_fname**. Line 95 moves the LOC_WONLY value into the **loc_oflags** file-open mode flags to tell ESQL/C to open the file in write-only mode. Finally, line 96 sets **loc_size** to -1 to tell ESQL/C to send the BYTE data in a single transfer rather than break the value into smaller pieces and use multiple transfers.

Lines 97 to 103

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 **loc_t** host variables, **cat_descr** and **cat_picture**, are listed in this clause for the TEXT and BYTE values.

Lines 104 to 112

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 106 to 112 handle a NOT FOUND condition. If the **exp_chk2()** function returns 100, the row was not found; lines 106 to 108 display a message to that effect. The **more_to_do()** function (line 109) 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 141 to close the database before the program terminates.

```
113     if(ret < 0)
114     {
115         EXEC SQL disconnect current;
116         printf("\nDISPCAT_PIC Sample Program over.\n\n");
117         exit(1);
118     }
119     printf("Displaying catalog picture for %ld...\n", cat_num);
120     if(cat_picture.loc_indicator == -1)
121         printf("\tNo picture available for catalog number %ld\n\n",
122             cat_num);
123     else
124         display_picture();
125     /*
126     * Display catalog_num and description from catalog table
127     */
128     printf("Stock Item for %ld: %s\n", cat_num, description);
129     prdesc(); /* display catalog.cat_descr */

130     unlink(cpf1); /* remove temp file for cat_picture */
131     if(!more_to_do()) /* More to do? */
132         break; /* no, terminate loop */

133     /* If user chooses to display more catalog rows, enable the
134     * memory-deallocation feature so that ESQL/C deallocates old
135     * catalog.cat_cesc buffer before it allocates a new one.
136     */
137     cat_descr.loc_mflags = LOC_ALLOC; /* enable memory-deallocation feature */
138 }
```

Continued on [page 8-60](#)

Lines 113 to 118

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 119 to 123

Line 119 notifies the user that the catalog picture is about to be displayed. If **cat_picture.loc_indicator** contains -1 (line 120), the **cat_picture** column contains a null and the program informs the user (lines 121 and 122). Execution then continues to line 128 to display the other returned column values.

Lines 124 and 125

If **cat_picture** is *not* null, the **display_picture()** function handles the display of the **cat_picture** data that the database server wrote to a temporary file. The format for the display depends on the value that you gave to the SUNVIEW conditional flag when you compiled. For more information, see [“Compiling the dispcat_pic Program” on page 8-48](#). For a detailed description of **display_picture()**, see [“Lines 142 to 169” on page 8-61](#).

Lines 128 and 129

These lines display the other columns that the SELECT statement returned. Line 128 displays the catalog number that is being processed and the **description** column from the **stock** table. Line 129 calls **prdesc()** to display the **cat_descr** column. For a detailed description of **prdesc()**, see [“Guide to the prdesc.c File” on page 8-66](#).

Lines 130 to 138

Line 130 deletes the file named in **cpfl[]**, the temporary file that contains the simple-large-object image for **cat_picture**. 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.

If execution continues to line 137, the user has chosen to enter another catalog number. In this case, the program sets **loc_mflags** to the LOC_ALLOC constant to enable the memory-deallocation feature for the ESQL/C buffers (see line 88). If ESQL/C needs to allocate a larger buffer, it first deallocates the existing buffer. If **loc_mflags** had remained 0, ESQL/C would *not* deallocate the existing buffer.

The closing brace on line 138 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.

```
139EXEC SQL disconnect current;
140    printf("\nDISPCAT_PIC Sample Program over.\n\n");

141} /* end main */

142/*
143 * Display the sunview raster file. Note that this function works only
144 * on SUN platforms.
145 */
146
147display_picture()
148{
149
150#ifdef SUNVIEW
151    int child, childstat, w;
152    static char path[] = "/bin/screenload";
153    static char *slargs[] = /* arguments for screenload */
154        {
155        "-w",
156        "-x260",
157        "-y300",
158        "-X400",
159        "-Y350",
160        cpfl,
161        (char *) 0,
162        };
163
164    if((child = fork()) == 0) /* child displays cat_picture */
165        {
166        execv(path, slargs); /* execute screenload */
167        fprintf(stderr, "\tCouldn't execute %s, errno %d\n", path, errno);
168        exit(1);
169        }
```

Continued on [page 8-62](#)

Line 139 to 141

When a **break** statement (line 132) terminates the main processing loop that the **while(1)** on line 61 began, control transfers to line 139, which closes the database and the connection to the default database server. The closing brace on line 141 terminates the **main()** function on line 22 and terminates the program.

Lines 142 to 169

Line 147 begins the declaration of the `display_picture()` function that displays the BYTE `cat_picture` value. This function can display the `cat_picture` column *only* if the program is compiled with the `-DSUNVIEW` option. For more information, see [“Compiling the `dispcat\_pic` Program” on page 8-48](#). At compile time, the `esql` command executes line 150 to check whether `SUNVIEW` is defined. It includes lines 151 to 177 in `display_picture()` *only* if `SUNVIEW` is defined. Otherwise, it includes only line 180 to print a message and return to `main()`.

Lines 151 to 162 define variables that participate in the storage and display of `cat_picture`, as follows.

Variable	Description
<code>child</code>	Receives the process ID of the child process that <code>display_picture()</code> creates with <code>fork()</code> .
<code>childstat</code>	Stores the status of the child process that <code>wait()</code> returns.
<code>w</code>	The value that <code>wait()</code> returns, the process ID of the terminated child process.
<code>path</code>	Stores the location and name of <code>screenload</code> , the program that displays the <code>cat_picture</code> image from <code>cpfl</code> .
<code>slargs</code>	Stores the command-line arguments for <code>screenload</code> , the program that displays the <code>cat_picture</code> image, as follows: <code>-w</code> Specifies the background color. <code>-x260, y300</code> Specifies the location of the picture in pixels. <code>-X400, -Y350</code> Specifies the size of the picture in pixels. <code>cpfl</code> The name of the file that contains the picture. <code>(char *) 0</code> The null terminator for the <code>slargs</code> array.

The database server writes the image from the **cat_picture** column to the file named in **cpfl**. To display the image, **display_picture()** calls the **fork()** system function to create a second process. The **display_picture()** program checks the value that **fork()** returns to distinguish the parent from the child process. The **fork()** returns 0 to the child process and the child process ID to the parent, so that only the child process executes lines 166 to 168. Line 166 executes the program named in **path**, the Sun **screenload** utility program. The **screenload** utility program overlays **dispcat_pic** in the child process and executes with the command-line arguments that are passed in **slargs[]**. Lines 167 and 168, which display an error message and exit, only execute if the system is unable to launch **screenload**.

```
170 /*
171  * parent waits for child to finish
172  */
173 if((w = wait(&childstat)) != child && w != -1)
174 {
175     printf("Error or orphaned child %d", w);
176     exit(-1);
177 }
178#endif /* SUNVIEW */

179#ifndef SUNVIEW
180 printf("*** Cannot print catalog picture.\n");
181#endif /* not SUNVIEW */
182}

183/* prdesc() prints cat_desc for a row in the catalog table */
184#include "prdesc.c"

185/*
186 * The inpfuns.c file contains the following functions used in this
187 * program:
188 *     more_to_do() - asks the user to enter 'y' or 'n' to indicate
189 *                   whether to run the main program loop again.
190 *
191 *     getans(ans, len) - accepts user input, up to 'len' number of
192 *                       characters and puts it in 'ans'
193 */
194#include "inpfuns.c"
```

Continued on page 8-64

Line 170 to 178

The parent process calls the **wait()** system function (line 173) and waits for the child process to terminate. The **wait()** system call returns the process ID of the process that terminates. Line 173 checks that this value, **w**, is the same as **child**, the value that **fork()** returns. It also checks that the value is not -1, which indicates that an error or an interrupt occurred in the child process. If either error occurs, lines 175 and 176 display a message and exit the program.

Lines 179 to 182

The **esql** command executes lines 179 and 181 at compile time. The **endif** directive marks the end of the text to include in the **display_picture()** function if SUNVIEW is defined. Line 179 checks if SUNVIEW is *not* defined.

Lines 183 and 184

Several of the 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 ESQL/C statements, the program can include it with the C **#include** preprocessor statement (instead of the ESQL/C **include** directive). For a description of this function, see [“Guide to the prdesc.c File” on page 8-66](#).

Lines 185 to 194

Several of the 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 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 infuncs.c File” on page 8-68](#).

```
195 /*
196 * The exp_chk.ec file contains the exception handling functions to
197 * check the SQLSTATE status variable to see if an error has occurred
198 * following an SQL statement. If a warning or an error has
199 * occurred, exp_chk2() executes the GET DIAGNOSTICS statement and
200 * displays the detail for each exception that is returned.
201 */
202 EXEC SQL include exp_chk.ec;
```

Lines 195 to 202

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 have been broken out into a separate **`exp_chk.ec`** source file. The **`dispcat_pic`** program must use the ESQL/C **`include`** directive to include this file because the exception-handling functions use ESQL/C statements. For a description of the **`exp_chk.ec`** source file, see [“Guide to the `exp\_chk.ec` File” on page 11-59](#).



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 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
3 prdesc()
4 {
5     long size;
6     char shdesc[81], *p;
7
8     size = cat_descr.loc_size; /* get size of data */
9     printf("Description for %ld:\n", cat_num);
10    p = cat_descr.loc_buffer; /* set p to buffer addr */
11
12    /* print buffer 80 characters at a time */
13    while(size >= 80)
14    {
15        ldchar(p, 80, shdesc); /* mv from buffer to shdesc */
16        printf("\n%80s", shdesc); /* display it */
17        size -= 80; /* decrement length */
18        p += 80; /* bump p by 80 */
19    }
20    strncpy(shdesc, p, size);
21    shdesc[size] = '\0';
22    printf("%-s\n", shdesc); /* display last segment */
23 }
```

Lines 3 to 22

Lines 3 to 22 make up the **prdesc()** function, which displays the **cat_descr** column of the **catalog** table. Line 5 defines **size**, a long integer that **prdesc()** initializes with the value in **cat_descr.loc_size**. Line 6 defines **shdesc[81]**, an array into which **prdesc()** temporarily moves 80-byte chunks of the **cat_descr** text for output. Line 6 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 8 moves **cat_descr.loc_size** to **size**. Line 9 displays the string "Description for:" as a header for the **cat_descr** text. Line 10 sets the **p** pointer to the buffer address that the database server returned in **cat_descr.loc_size**.

Line 13 begins the loop that displays the **cat_descr** text to the user. The **while()** repeats the loop until **size** is less than 80. Line 14 begins the body of the loop. The 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 16 prints the contents of **shdesc[]**. Line 17 subtracts 80 from **size** to account for the portion of the buffer that was just printed. Line 18, 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 20 copies the remainder of the buffer into **shdesc[]** for the length of **size**. Line 21 appends a null to **shdesc[size]** to mark the end of the array and line 22 displays **shdesc[]**.

Guide to the *inpfuns.c* File

The **inpfuns.c** file contains the following two functions:

- The **getans()** function
- The **more_to_do()** function

Because these functions are used in several ESQL/C demonstration programs, they are in a separate file and included in the appropriate demonstration source files.

```
1 /* The inpfuns.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 int len;
16 {
17     char buf[BUFSIZE + 1];
18     int c, n = 0;

19     while((c = getchar()) != '\n' && 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 }
```

Continued on page 8-70

Lines 4 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 12 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 **BUFSZ**. 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. ♦

Working with INFORMIX- Universal Server Complex Data Types

Accessing a Collection.	9-4
Declaring Collection Variables	9-5
Typed and Untyped Collection Variables	9-6
Client Collections.	9-11
Managing Memory for Collections	9-12
Operating on a Collection Variable	9-13
Using the Collection-Derived Table Clause on Collections.	9-14
Initializing a Collection Variable	9-16
Inserting into a Collection Variable.	9-18
Selecting from a Collection Variable	9-24
Updating a Collection Variable	9-29
Specifying Element Values	9-32
Deleting Elements from a Collection Variable	9-34
Accessing a Nested Collection	9-37
Saving into a Collection Column	9-40
Operating on a Collection Column	9-41
Selecting from a Collection Column	9-41
Inserting into and Updating a Collection Column	9-41
Deleting an Entire Collection.	9-44
ifx_cl_card()	9-45
Accessing Row Types.	9-48
Declaring Row Variables	9-49
Typed and Untyped Row Variables	9-50
Using Named Row Types	9-53
Client-Side Rows	9-56
Managing Memory for Rows	9-57

Operating on a Row Variable	9-58
Using the Collection-Derived-Table Clause on	
Row Types	9-59
Initializing a Row Variable.	9-62
Inserting into a Row Variable.	9-64
Selecting from a Row Variable	9-64
Updating a Row Variable	9-66
Deleting from a Row Variable	9-68
Specifying Field Names.	9-68
Specifying Field Values.	9-69
Saving into a Row-Type Column	9-71
Accessing a Typed Table.	9-72
Operating on a Row-Type Column	9-73
Selecting from a Row-Type Column	9-73
Inserting into and Updating Row-Type Columns	9-74
Deleting an Entire Row Type	9-76

T

he information in this chapter applies only if you are using INFORMIX-Universal Server as your database server.

This chapter explains how to use **collection** and **row** data types in an INFORMIX-ESQL/C program. These ESQL/C data types access the complex data types that Universal Server supports, as the following table shows.

Universal Server Data Type	ESQL/C Host Variable
Collection types: LIST, MULTiset, SET	Typed collection host variable
	Untyped collection 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 ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#) For information about SQL complex data types, see the [Informix Guide to SQL: Reference](#).

Accessing a Collection

ESQL/C supports the SQL collection data types with the ESQL/C **collection** data type for host variables. A *collection data type* is a complex data type that contains one or more members called elements. All *elements* have the same data type. Universal Server 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.

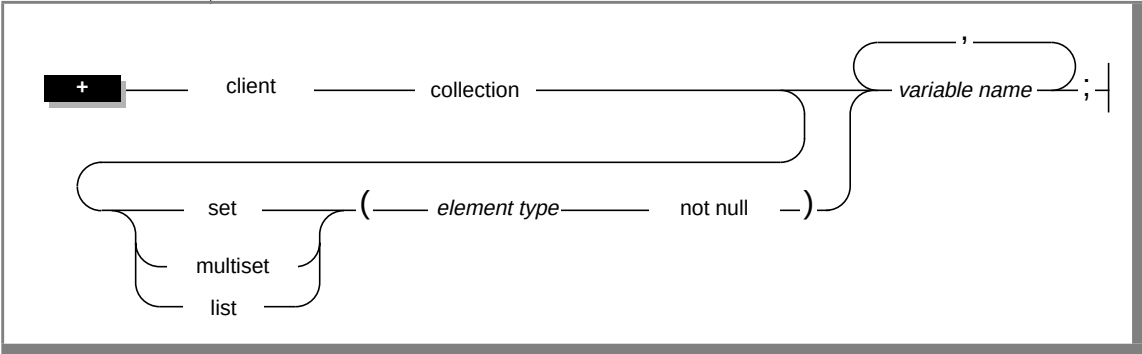
To access the elements of a column in a Universal Server table that has a collection type (LIST, MULTiset, or SET) as its data type, perform the following steps:

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.

The following sections describe each of these steps in more detail.

Declaring Collection Variables

Use the **collection** data type to declare host variables for database columns of collection data types (SET, MULTISSET, or LIST). To declare a **collection** host variable, use the **collection** keyword as the variable data type, as the following syntax shows.



Element	Purpose	Restrictions	SQL Syntax
<i>element type</i>	Data type of the elements in the collection variable.	Can be any data type except SERIAL, SERIAL8, TEXT, or BYTE.	Data Type segment in the Informix Guide to SQL: Syntax
<i>variable name</i>	Name of the 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, MULTISSET, or SET.

Important: You must specify the **client** keyword when you declare **collection** variables.

Typed and Untyped Collection Variables

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, MULTISSET, or LIST) and the element type for the **collection** variable.

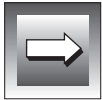
Figure 9-1 shows declarations for three typed **collection** variables.

```
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 ESQL/C data types. For example, as the declaration for the **list1** variable in [Figure 9-1 on page 9-6](#) illustrates, use the SQL SMALLINT data type, not the 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 cur into:set_float;  
EXEC SQL fetch cur1 into :set_int;
```

When it executes the first **FETCH** statement, the **ESQL/C** client program automatically converts the integer elements in the column to the float values in the **set_float** host variable. The **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 (**ESQL/C** needs to know what the type is)
- When you update an element in a derived table (**ESQL/C** needs to know what the type is)
- When you want the server to perform a cast. (The **ESQL/C** client sends the type information to the database server, which attempts to perform the requested cast operation. If it's 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 on the collection-derived table clause, see [“Using the Collection-Derived Table Clause on Collections” on page 9-14.](#))



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 9-9.](#)

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 [page 9-6](#).

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. 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 9-16](#) 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;

an element at the end of the LIST in the untyped
 * collection variable */
EXEC SQL insert into table(:a_coll) values (7);
```



Tip: For more information about the **ALLOCATE COLLECTION** statement, see [“Managing Memory for Collections” on page 9-12](#). For more information about how to insert into a collection variable, see [“Inserting into a Collection Variable” on page 9-18](#).

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 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 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. ESQL/C will perform the select, insert, update, or delete operation. 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, 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 9-18](#). For more information on a collection-derived table, see [“Operating on a Collection Variable” on page 9-13](#).)

Managing Memory for Collections

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. For syntax information for the **ALLOCATE COLLECTION** and **DEALLOCATE COLLECTION** statements, refer to their descriptions in the [Informix Guide to SQL: Syntax](#).

Operating on a Collection Variable

Universal Server 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 9-41.](#))



Tip: Universal Server 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 ESQL/C application constructs a subtable, called a *collection-derived table*, in the collection host variable. From collection-derived table, the 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 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 ESQL/C application before it appears in a collection-derived table clause.

For more information on the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the [Informix Guide to SQL: Syntax](#).

Accessing a Collection Variable

In SQL statements, the 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 9-18](#).

- 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 9-24](#).

- 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 9-29](#).

- 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 9-34](#).



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 9-41](#).

Once the **collection** host variable contains valid elements, you update the collection column with the contents of the host variable. For more information, see [“Saving into a Collection Column” on page 9-40](#). For more information on the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the [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, 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 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 9-60](#).

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 ESQL/C with a description of the collection declaration. For more information, see [“The Untyped Collection Variable” on page 9-9](#).*

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 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 9-16](#) defines:

```
EXEC SQL BEGIN DECLARE SECTION;
      client collection a_collection;
EXEC SQL END DECLARE SECTION;

te 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 on the types of values you can insert into a **collection** variable, see [“Specifying Element Values” on page 9-32](#).

Inserting One Element

The INSERT statement and the collection-derived table clause allow you to insert *one* element into a collection. 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.

To Insert into SET and MULTISSET Collections

For SET and MULTISSET 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_datasetMULTISSET(INT8 NOT NULL)
);
```

To access the **time_dataset** column, the typed 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 on the **ifx_int8cvint()** function and the INT8 data type, see [Chapter 5, “Working with Numeric Data Types.”](#)

To Insert 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 within 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 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 on the AT clause, see the description of the INSERT statement in the [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 ESQL/C program.
For more information, see “[Declaring Collection Variables](#)” on page 9-5 and “[Managing Memory for Collections](#)” on page 9-12.
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 “[Saving into a Collection Column](#)” on page 9-40.



Tip: Instead of an insert cursor, you can use an INSERT statement to insert elements one at a time into a **collection** variable. For more information, see “[Inserting One Element](#)” on page 9-18. However, an insert cursor is more efficient for large insertions.

The following sections provide information on 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 9-23](#).

To Declare 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 following items:

- The value(s) 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(:a_list) values (?)';
EXEC SQL declare list_curs cursor for ins_stmt;
EXEC SQL open list_curs;
```

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 9-23](#).



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. Up until the OPEN statement, 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 on how to use the PUT statement, see “To Put 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.

To Put 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. However, you must prepare the INSERT statement when it contains input parameters. 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 9-16](#) 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(:a_list) values (?)';
EXEC SQL declare list_curs cursor for ins_stmt;
EXEC SQL open list_curs;

/* 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, 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, 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.

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.

To Free 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 [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 9-16](#)) 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. 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 on how to use a select cursor, see [“Selecting More Than One Element” on page 9-26](#).

If the element of the collection is itself a complex type (collection or row type), the collection is a *nested collection*. For information on how to use a cursor to select elements from a nested collection, see [“Selecting Values from a Nested Collection” on page 9-37](#). 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 tabl
      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 ESQL/C program.
For more information, see [“Declaring Collection Variables” on page 9-5](#) and [“Managing Memory for Collections” on page 9-12](#).
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 [“Saving into a Collection Column” on page 9-40](#).
5. Close the select cursor with the CLOSE statement, and if you no longer need the cursor, free it with the FREE statement.

To Declare 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 [“To Fetch Elements from the Select Cursor” on page 9-28](#).

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 9-35](#) and [“Updating One Element” on page 9-30](#).

To Fetch 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 9-16](#)) 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 [page 9-16](#)), 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 on the types of values you can specify when you update a **collection** variable, see [“Specifying Element Values” on page 9-32](#).

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** 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 has been 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 ESQL/C program.
For more information, see [“Declaring Collection Variables” on page 9-5](#) and [“Managing Memory for Collections” on page 9-12](#).
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 [page 9-27](#).
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 9-26](#).
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 [“Saving into a Collection Column” on page 9-40](#).
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 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 9-16](#)).

```
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 9-41](#).

- An 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 specific values.

For information on how to insert elements into a **collection** variable, see [“Inserting into a Collection Variable” on page 9-18](#). For information on how to update elements in a **collection** variable, see [“Updating a Collection Variable” on page 9-29](#). 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 on literal-collection values for a collection column, see [“Inserting into and Updating a Collection Column” on page 9-41](#).

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 9-41](#).

ESQL/C Host Variables as Elements

You can use an 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 9-20](#).

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 9-30](#).

Deleting Elements from a Collection Variable

Once you have initialized a **collection** host variable with a collection column (see [page 9-16](#)), 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** 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 has been 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 9-30](#)). 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 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 9-16](#)).

```
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

Universal Server 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
Nested-Collection
Column

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;
```

Operating on a Collection Variable

```
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 9-37](#) 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;
```

Saving into a Collection Column

The **collection** variable stores the elements of the collection. However, it has no intrinsic connection with a database column. Once the **collection** variable contains the correct elements, you must then explicitly save the contents of the variable into the collection column with one of the following statements:

- To *update* the collection column in the table with the **collection** variable, use an UPDATE statement on a table or view name and specify the **collection** variable in the SET clause.
- To *insert* a collection into a column, use the INSERT statement on a table or view name and specify the **collection** variable in the VALUES clause.

[Figure 9-5 on page 9-31](#) shows the UPDATE statement that saves the contents of the **a_set** variable in the **set_col** collection column.

[Figure 9-3 on page 9-23](#) shows the INSERT statement that saves the contents of the **a_list** variable in the **list_col** collection column.

In the INSERT and UPDATE statements, a **collection** host variable is valid anywhere an ESQL/C host variable is valid. For more information, see [“Inserting into and Updating a Collection Column” on page 9-41](#).

Operating on a Collection Column

You can use the SELECT, UPDATE, INSERT, and DELETE statements to access a collection column (SET, MULTISSET, 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.
- The UPDATE statement updates the entire collection in a collection column with new values.
- The DELETE statement deletes from a table a row that contains a collection column, thereby deleting all elements from the collection column.

The following sections summarize these SQL statements with collection columns. For more information on how to use these statements with collection columns, see the [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 ESQL/C application. For more information, see “[Initializing a Collection Variable](#)” on page 9-16.

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 ESQL/C **collection** host variable
For more information, see [“Saving into a Collection Column” on page 9-40.](#)
- 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, MULTISSET, 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 9-16](#) 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 int1 = 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 within this literal-collection value, it too must be enclosed in quotes; this condition creates nested quotes. Nested quotes can occur in the following cases:

- If the collection possesses a character-element type
For a **col1** column of type SET(CHAR(5)), a literal value would be:
`'SET{"abcde"}'`
- If the collection is nested; that is, it contains a row type or a collection as a member
For a **col2** column whose data type is LIST(ROW(a INTEGER, b SMALLINT) NOT NULL), a literal value would be:
`'LIST{"ROW(80, 3)"}'`

Rules for grouping nested quotes can be found in the descriptions of the Literal Collection segment in the [Informix Guide to SQL: Syntax](#).

To specify nested quotes within an SQL statement in an ESQL/C program, you must escape every double quote when it appears within a single-quote string. The following two INSERT statements show how to use escape characters for inner double quotes:

```
EXEC SQL insert into (col1) tabl
values ('SET{"abcde"}');

EXEC SQL insert into (col2) tabl
values ('LIST{"ROW(80, 3)"}');
```

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 on the syntax of literal values for **collection** variables, see [“Literal Values as Elements” on page 9-32](#). For more information on the syntax of literal-collection values for collection columns, see the Literal Collection segment in the [Informix Guide to SQL: Syntax](#).

Deleting an Entire Collection

To *delete* the entire collection in a collection column, specify the table, view, or synonym name after the FROM keyword of the DELETE statement and use the WHERE clause to identify the row or rows that you want to delete.

The following DELETE statement deletes the **tab_set** row that contains the LIST collection that the UPDATE statement in [Figure 9-7 on page 9-42](#) saves:

```
EXEC SQL delete from tab_set
      where int1 = 5;
```

ifx_cl_card()

The **ifx_cl_card()** function returns the cardinality of the specified collection type host variable.

Syntax

```
int ifx_cl_card(ifx_collection_t *collp, int *isnull)
           ifx_collection_t *collp;
           int *isnull;
```

COLLP is a pointer to the name of the **collection** host variable in the application.

isnull is 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 Values

- 0 The collection is empty.
- >0 The number of elements in the collection.
- <0 An error occurred.

Example

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;
    int numelems = 0;
    int 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");
}

```


Example Output

```
collection is empty  
number of elements is 3  
collection is null
```

Accessing Row Types

ESQL/C supports the SQL row types with the ESQL/C **row** data type for host variables. 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. Universal Server supports the following two kinds of row types:

- 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.

- 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

The database server checks data type values.

For more information about row types, see the CREATE ROW TYPE statement in the [Informix Guide to SQL: Syntax](#) and the [Informix Guide to SQL: Reference](#).

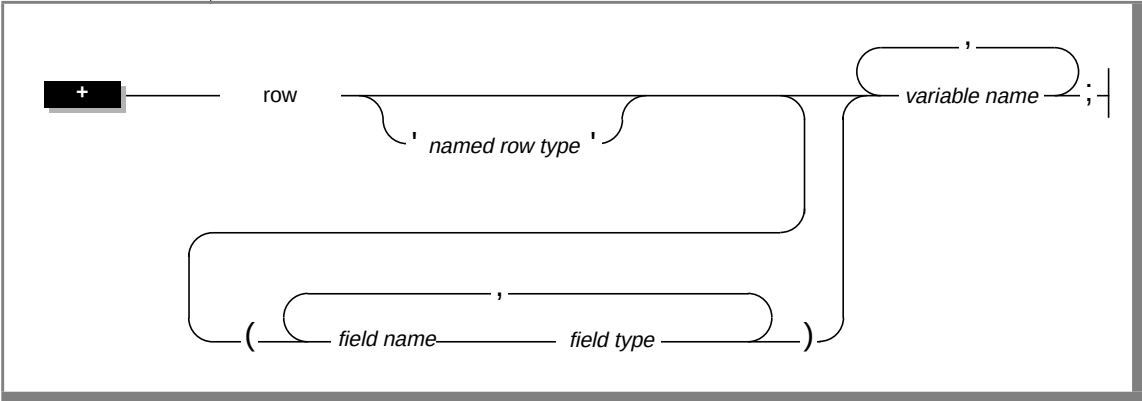
To access a column in a Universal Server table that has a row type (named or unnamed) as its data type, perform the following steps:

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

Use the **row** data type to declare host variables for database columns of named or unnamed row types. To declare a **row** host variable, use the **row** keyword as the variable data type, as the following syntax shows.



Element	Purpose	Restrictions	SQL Syntax
<i>field name</i>	Name of a field in the row variable	Must match the corresponding field name in any associated row-type column.	Identifier segment in the Informix Guide to SQL: Syntax
<i>field type</i>	Data type of the <i>field name</i> field in the row variable	Can be any data type except SERIAL, SERIAL8, TEXT, or BYTE.	Data Type segment in the Informix Guide to SQL: Syntax
<i>named row type</i>	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 Informix Guide to SQL: Syntax
<i>variable name</i>	Name of the ESQL/C variable to declare as a row variable		Name must conform to language-specific rules for variable names.



Tip: This manual uses the ANSI-compliant syntax for the declaration of host variables. However, ESQL/C also supports the Informix-specific declaration syntax for the declaration of **row** host variables. For more information on ANSI-compliant and Informix-specific syntax for variable declaration, see [“Declaring a Host Variable” on page 1-27](#).

Typed and Untyped Row Variables

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.

ESQL/C handles **row** variables as client-side **collection** variables.

The Typed Row Variable

A *typed row* variable provides an exact description of the row. This declaration 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

Tip: You cannot use the **BYTE**, **TEXT**, **SERIAL**, or **SERIAL8** built-in types as the field type for a field in a **row** variable.



When you specify the field type of a field in the **row** variable, use the SQL data types, not 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 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 whose data 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.



Tip: If you do not know the exact data type of the row-type column you want to access, use an untyped **row** variable. For more information, see [“The Untyped Row Variable” on page 9-52](#)

The benefit of a typed **row** variable is that ESQL/C is more restrictive with respect to type matching between the information you put in the typed **row** variable (with the SELECT statement or with C assignment statements) and the definition of the host variable. If you try to place anything incompatible with the definition into the typed **row** variable, ESQL/C returns an error.

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;
```

The Untyped Row Variable

An *untyped row* variable provides a general description of a row. This declaration includes only the **row** keyword and the variable 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 field types of the fields in the row) to the client application. The client application then uses this information to assign a data type to the untyped **row** variable.



Important: You must obtain the data type of an untyped **row** variable before you use it in an SQL statement.

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 allocation row :a_row;

/* select ROW column into the untyped row variable
 * to 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;
```

Tip: For more information about the **ALLOCATE ROW** statement, see [“Managing Memory for Rows” on page 9-57](#).



You can use an untyped **row** variable to store row types with different data-type information, as long as you select the associated row-type column description into the **row** variable before you use the variable in an SQL statement.

The untyped **row** variable has the following restrictions:

- Because you determine the data type of an untyped **row** variable at runtime, ESQL/C can perform the more restrictive type matching with a typed **row** variable.
- Before the **row** variable can hold any values, you must use a SELECT statement to obtain a description of the row type from a row-type column in the database. Therefore, you cannot insert or select values directly into an untyped **row** variable.

Tip: If you know the exact data type of the row-type column you want to access, use a typed **row** variable. For more information, see [page 9-50](#).



Using Named Row Types

ESQL/C supports the specification of a named row type as part of the declaration of a **row** variable. A named row type has a name associated with the row structure. You create a named row type in the database 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 ESQL/C **row** variable can be typed or untyped. A *typed* named **row** variable includes a list of declarations for the fields in the row. An *untyped* named **row** variable omits the field declarations. For more information, see “[Typed and Untyped Collection Variables](#)” on [page 9-6](#).

The ESQL/C preprocessor does not check the validity of a row-type name and ESQL/C does not use this name at runtime. ESQL/C just sends this name to the database server to provide information for type resolution. Therefore, 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. 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 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 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, 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 ESQL/C gets 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 the **tab1** table.

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

When you declare a **row** variable (typed or untyped), the client application declares the **row** variable name, allocates the memory, and performs operations on the row. Therefore, a **row** variable is sometimes called a *client-side 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 contain a collection-derived table clause, 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, 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;
```

Because **row** variables use client-side processing, SQL statements have the following syntax restrictions when they include a **row** variable:

- You must use SELECT, INSERT, UPDATE, and DELETE statements with the collection-derived table clause to access fields of a row type.
- An INSERT statement cannot have a SELECT, an EXECUTE FUNCTION, or an EXECUTE PROCEDURE statement in the VALUES clause.

For more information on the collection-derived table clause, see [“Operating on a Row Variable” on page 9-58](#).

Managing Memory for Rows

Once you declare a **row** variable, ESQL/C recognizes the variable name. For typed **row** variables, ESQL/C also recognizes the associated data type. However, 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.

Important: *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;
```



Tip: The **:a_name** syntax in the `ALLOCATE ROW` and `DEALLOCATE ROW` statements is the ANSI-compliant way to specify a host variable name. ESQL/C also supports the Informix-specific syntax of **\$a_name** with these statements. For more information, see [“Declaring a Host Variable” on page 1-27](#).

You must explicitly manage the client-side memory for a **row** variable with the `ALLOCATE ROW` and `DEALLOCATE ROW` statements. For syntax information for the `ALLOCATE ROW` and `DEALLOCATE ROW` statements, refer to their descriptions in the [Informix Guide to SQL: Syntax](#).

Operating on a Row Variable

The `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements allow you to access a row-type column as a whole. Universal Server can also access individual fields in a row-type column, as follows:

- Review the contents of rows with the `IN` predicate in the `WHERE` clause of a `SELECT` statement.
- Access fields individually with the dot notation of field projection in the `SELECT` statement.

In addition, an 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 with collection columns, the `SELECT` statement can access individual members of row-type columns. Therefore, an 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 9-73](#).

- 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, the ESQL/C application accesses the row-type column as a *collection-derived table*. A collection-derived table allows the application to decompose a row into its fields, which the application can then access individually. The collection-derived table contains a single table row and each column in this table row is a field of the row-type column.

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 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 insert field values into a **row** host variable
- How to select fields from a **row** host variable
- How to update field values in a **row** host variable
- How to delete field values from 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 )
```

In the preceding syntax, *row_var* is a **row** host variable. It can be either a typed or untyped **row** host variable but it must already be declared.



Tip: The “:row_var” syntax in the collection-derived table clause is the ANSI-compliant way to specify a host variable name. ESQL/C also supports the Informix-specific syntax of “\$row_var” with the *TABLE* specifier. For more information, see [“Declaring a Host Variable” on page 1-27](#).

For more information on the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the [Informix Guide to SQL: Syntax](#).

Accessing a Row Variable

In SQL statements, use a collection-derived table in place of a table name to access the **row** host 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 9-64](#).

- You can *update* all or some fields in the **row** host variable collection-derived table clause (instead of a table name) after the UPDATE keyword in an UPDATE statement.

For more information, see [“Updating a Row Variable” on page 9-66](#).

The insert and delete operations are *not* supported on **row** variables. For more information, see [page 9-64](#) and [page 9-68](#).



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 9-74](#).

Once the **row** host variable contains valid elements, you update the row-type column with the contents of the host variable. For more information, see [“Saving into a Row-Type Column” on page 9-71](#). For more information on the syntax of the collection-derived table clause, see the description of the collection-derived table segment in the [Informix Guide to SQL: Syntax](#).

Distinguishing Between Columns and Row Variables

When you use the collection-derived table clause with an SQL statement (such as SELECT, INSERT, or UPDATE), the statement is not sent to the database server for processing. Instead, ESQL/C processes the SQL statement. 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 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 values to modify a **row** host variable, the preprocessor does not check that you correctly specify any host variables. You must ensure that you use valid host-variable syntax for any host variables that contain values in:

- The SET clause of an UPDATE statement
- The VALUES clause of an INSERT statement

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);
```

The ESQL/C preprocessor assumes that **clob_ins** is a column and does not catch the missing colon for the **clob_ins** host variable. The client application does not return an error when it executes the UPDATE, because nothing is syntactically wrong with the statement. Therefore, you might think that the UPDATE successfully added the contents of the **clob_ins** host variable to the **clob_fld** field of the **named_row1** row variable. Instead, ESQL/C has stored a null value in the **clob_fld** field because no valid value was provided in the SET clause and no column named **clob_ins** exists.

You must ensure that the syntax for host variables is correct when you use these variables to specify values for **row** variables.

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 9-62](#) 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, ESQL/C does not perform type checking on an untyped **row** host variable because its elements do not have a predefined data type.

Important: For an untyped **row** variable, the SELECT statement is required, regardless of the type of operation you wish to perform on the row. Selecting the row-type column into the untyped **row** variable provides ESQL/C with a description of the row declaration. For more information, see [“The Untyped Row Variable” on page 9-52](#).

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

An insert operation does not apply to a **row** variable because an insert normally adds a row to 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 add new table rows to the collection-derived table. Therefore, the INSERT statement does not support a **row** variable in the collection-derived table clause. ESQL/C returns an error if you attempt to perform an insert operation on a **row** variable.

However, you can use the UPDATE statement to insert new field values into a **row** variable. For more information, see [“Updating a Row Variable” on page 9-66](#).

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 **row** variable that holds the field value(s) selected from the row-type column. 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;

t 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 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 9-74](#),

Updating a Row Variable

Once you have initialized a **row** host variable with a row-type column (see [page 9-62](#)), you can use the UPDATE statement with the collection-derived table clause to update the fields in the row type. The collection-derived table clause identifies the **row** variable whose fields are to be updated.

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 value(s) 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 **myrect** 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;
:
EXEC SQL select into :myrect from rectangles
      where area = 64;
new_y = 4;
EXEC SQL update table(:myrect)
      set x=3, y=:new_y;
```

Suppose that after the SELECT statement, the **myrect2** variable has the values **x=0, y=0, length=8, and width=8**. After the UPDATE statement, **myrect2** variable has field values of **x=3, y=4, length=8, and width=8**.

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);
```



Tip: If the **row** variable is an untyped variable, you must use a SELECT statement before the update so that ESQL/C can determine the data types of the fields. For more information, see [“The Untyped Row Variable” on page 9-52](#).

For information on the types of values you can specify when you update a **row** variable, see [“Specifying Field Values” on page 9-69](#).

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. 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 modify existing field values in a **row** variable. For more information, see [“Updating a Row Variable” on page 9-66](#).

Specifying Field Names

ESQL/C is case insensitive with regard to the field names of a **row** variable. In a SELECT, INSERT, UPDATE, or DELETE statement, ESQL/C always interprets field names of a **row** variable as lowercase. For example, in the following SELECT statement, ESQL/C interprets the fields to select as **x** and **y** (lowercase), 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.

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 on delimited identifiers and the **DELIMIDENT** environment variable, see [“SQL Identifiers” on page 1-32](#).

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 9-74](#).
- 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 ESQL/C host variable

You cannot include complex expressions directly to specific values.

For information on how to update field values into a **row** variable, see [“Updating a Row Variable” on page 9-66](#). The following sections describe the values you can assign to an element 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 that [Figure 9-8 on page 9-50](#) defines:

```
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 9-66](#) defines) with a literal row:

```
EXEC SQL insert into table(:nested_row)
values (1, row(2,3));
```



Tip: 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. For more information on literal-row values for a row-type column, see [“Inserting into and Updating Row-Type Columns” on page 9-74](#).

The following UPDATE statement updates the **nested_row** variable with a new literal-row value:

```
EXEC SQL update table(:nested_row)
set b = row(7, 12);
```



Tip: If you only need to insert or update the row-type column with literal values, you do not need to use a **row** host variable. Instead, you can explicitly list the literal values as a 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 9-74](#).

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 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 + 6, a_func(:a_strng));
```

For more information on the syntax of a row constructor, see the Expression segment in the [Informix Guide to SQL: Syntax](#).

ESQL/C Host Variables as Field Values

You can use an 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 field 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 that was declared in [“Literal Values as Field Values” on page 9-69](#):

```
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 UPDATE statement performs the same task as the preceding UPDATE:

```
EXEC SQL update table(:a_row) set (fld1, fld2) =
    (:one_fld, :second_fld);
```

Saving into a Row-Type Column

The **row** variable stores the fields of the row type. However, it has no intrinsic connection with a database column. Once the **row** variable contains the correct field values, you must then explicitly save the contents of the variable into the row-type column with one of the following statements:

- To *update* the row-type column in the table with the contents of a **row** variable, use an UPDATE statement on a table or view name and specify the **row** variable in the SET clause.
- To *insert* a row into a row-type column, use the INSERT statement on a table or view name and specify the **row** variable in the VALUES clause.

In the INSERT and UPDATE statements, a **row** host variable is valid anywhere that an ESQL/C host variable is valid.

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 has been 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 9-62](#) 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 table 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** column value of the **names** row through the **row** variable, **a_name**. For more information about typed tables, see the CREATE TABLE statement in the [Informix Guide to SQL: Syntax](#) and the [Informix Guide to SQL: Tutorial](#).

Operating on a 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.
- 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.

The following sections summarize these SQL statements with row-type columns. For more information on how to use these statements with row-type columns, see the [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. If you put a **row** host variable in the INTO clause of the SELECT statement, you can access these fields from an ESQL/C application. For more information, see “[Initializing a Row Variable](#)” on [page 9-62](#).

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 code fragment uses a single SELECT statement to perform the same task as the two SELECT statements in [Figure 9-10 on page 9-65](#):

```
EXEC SQL select rect.width into :rect_width from rectangles;
```

For more information on dot notation, see the Column Expression section of the Expression segment in the [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* new field values into an empty 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 ESQL/C **row** host variable
For more information, see [“Saving into a Row-Type Column” on page 9-71](#).
- A constructed row
Constructed rows are described with respect to **row** variables in [“Constructed Rows” on page 9-70](#). For information on the syntax of a constructed row, see the Constructed Row segment in the [Informix Guide to SQL: Syntax](#).
- A literal-row value
For more information on the syntax of a literal-row value, see the Literal Row segment in the [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 9-62](#) 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 within this literal-row value, it too must be enclosed in quotes; this condition creates nested quotes. Nested quotes can occur in the following cases:

- If the row type contains a character-type field
For a **row1** column of row type ROW(id INTEGER, name CHAR(5)), a literal value would be:
`'ROW(6, "dexter")'`
- If the row type is nested; that is, it contains a row type or a collection as a member
For a **col2** column whose data type is ROW(a INTEGER, b SET (INTEGER)), a literal value would be:
`'ROW(1, "SET{80, 81, 82, 83}")'`

Rules for grouping nested quotes can be found in the descriptions of the Literal Row segment in the [Informix Guide to SQL: Syntax](#).

To specify nested quotes within an SQL statement in an ESQL/C program, you must escape every double quote when it appears within 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 on the syntax of literal values for **row** variables, see [“Literal Values as Field Values” on page 9-69](#). For more information on the syntax of literal-row values, see the Literal Row segment in the [Informix Guide to SQL: Syntax](#).

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 9-75](#) saves:

```
EXEC SQL delete from tab_unmrow
      where area = 20;
```

Working with INFORMIX- Universal Server Opaque Data Types

The SQL Opaque Data Type.	10-3
Accessing the External Format of an Opaque Type	10-6
Declaring lvarchar Host Variables	10-7
An lvarchar Host Variable of a Fixed Size	10-9
The lvarchar Pointer Host Variable.	10-9
The Opaque Type Name	10-11
Using lvarchar Host Variables.	10-11
Using Fixed-Size lvarchar Host Variables	10-12
Using lvarchar Pointer Variables	10-14
Accessing the Internal Format of an Opaque Type	10-17
Accessing a Fixed-Length Opaque Type	10-18
Declaring fixed binary Host Variables.	10-18
Using fixed binary Host Variables	10-21
Accessing a Varying-Length Opaque Type	10-23
Declaring var binary Host Variables	10-24
Using var binary Host Variables	10-25
The lvarchar pointer and var binary Library Functions	10-32
ifx_var_alloc()	10-33
ifx_var_dealloc()	10-35
ifx_var_flag()	10-37
ifx_var_getdata()	10-39
ifx_var_getlen().	10-40
ifx_var_isnull()	10-41
ifx_var_setdata()	10-42
ifx_var_setlen().	10-44
ifx_var_setnull()	10-46
Accessing Predefined Opaque Data Types	10-48

T

he information in this chapter applies only if you are using INFORMIX-Universal Server as your database server.

This chapter explains how to use the **lvvarchar**, **fixed binary**, and **var binary** data types to access an opaque data type from an INFORMIX-ESQL/C program. Use these ESQL/C data types to represent an opaque data type as it is transferred to and from Universal Server.

For information about all the data types that are available for use in an ESQL/C program, see [Chapter 2, “INFORMIX-ESQL/C Data Types,”](#) of this manual. For information about SQL complex data types, see the [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.



Tip: For more information on how to create an opaque data type, see the description of the CREATE OPAQUE TYPE statement in the “[Informix Guide to SQL: Syntax](#)” and in “[Extending INFORMIX-Universal Server: Data Types](#).”

You can access the value of an opaque data type from an 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 ESQL/C data types you can use to access an opaque data type.

Universal Server Data Type	ESQL/C Host Variable
External format of an opaque data type	lvvarchar 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 ESQL/C **lvvarchar** 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.

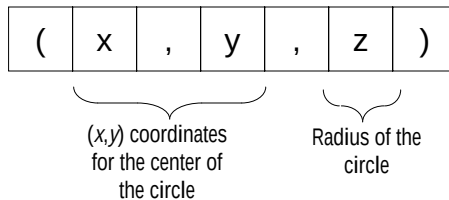


Figure 10-3
External Format of
the circle Opaque
Data Type

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 **lvvarchar** 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 **lvvarchar** 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 **lvvarchar** 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 **lvvarchar** 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.

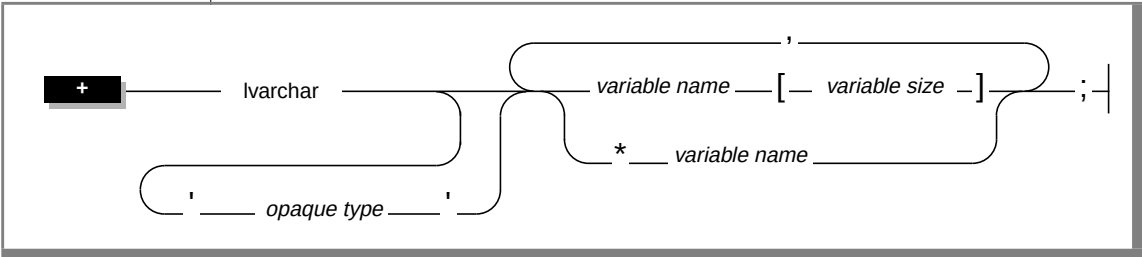
Follow these steps to transfer the external format of an opaque-type column between the database server and the ESQL/C application:

- 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 Ivarchar 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.



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 <i>Informix Guide to SQL: Syntax</i>
variable name	Name of the ESQL/C variable to declare as an lvarchar variable		Name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	SQL Syntax
<i>*variable name</i>	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 10-32.	Name must conform to language-specific rules for variable names.
<i>variable size</i>	Number of bytes to allocate for the lvarchar variable	Integer value can be between 1 and 32,000 bytes (32 kilobytes).	

(2 of 2)



Tip: To declare an **lvarchar** host variable for an **LVARCHAR** column, use the syntax that [“The lvarchar Data Type” on page 3-9](#) 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 Ivarchar 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 on how to use an **lvarchar** host variable of a fixed size, see [Chapter 3, “Working with Character and String Data Types.”](#)

The Ivarchar 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. 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 more information on the **ifx_var()** functions, see [“The Ivarchar pointer and var binary Library Functions” on page 10-32.](#)

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 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

If you choose to allocate the memory to store data for an **lvarchar** host variable you must first fetch the data into **sqllda** 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_getllen(&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_getllen(&mypoly1) );
        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 Ivarchar Host Variables

Your 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 Ivarchar Host Variables

Figure 10-6 shows how to use a fixed-size **lvvarchar** host variable to insert and select data in the **circle_col** column of the **circle_tab** table (see [Figure 10-4 on page 10-6](#)).

```
EXEC SQL BEGIN DECLARE SECTION;
    lvvarchar '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 lvvarchar 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
lvvarchar 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 Ivarchar Host Variable

To insert the data from a fixed-size **lvvarchar** host variable into an opaque-type column, take the following steps, which are illustrated in Figure 10-6:

1. Define the fixed-size **lvvarchar** 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 **lvvarchar** host variable.

When you put data into an **lvvarchar** host variable, you must know the external format of the opaque type. For the INSERT statement to succeed, the data in the **lvvarchar** host variable **lv_circle** must conform to the external format of the opaque data type (which [Figure 10-3 on page 10-5](#) shows).

3. Insert the data that the **lvvarchar** 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 ESQL/C client application has sent, to the internal format that it stores on disk.

[Figure 10-6 on page 10-12](#) 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 **lvvarchar** host variable to insert a null value into an opaque-type column with the following steps:

- Set the **lvvarchar** host variable to an empty string.
- Set an indicator variable for the **lvvarchar** 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;
    lvvarchar 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 on indicator variables, see [page 1-42](#).

Selecting into a Fixed-Size Ivarchar Host Variable

To select data from an opaque type column into a fixed-size **lvvarchar** host variable, the code fragment in [Figure 10-6 on page 10-12](#) 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 ESQL/C application requests. This SELECT statement also uses a user-defined function called **radius** (see [Figure 10-2 on page 10-5](#)) 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 **lvvarchar** 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 **lvvarchar** host variable, ESQL/C sets any accompanying indicator variable to -1.

Using Ivarchar Pointer Variables

The following sections illustrate how to insert to and select from an opaque type column with an **lvvarchar** 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"
```

Inserting from an Ivarchar 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(&myplay1, size+1); /* Allocate memory for data in mypoly1 */
    ifx_var_setlen(&myplay1, 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 Ivarchar Pointer Host Variable

The following example code illustrates the steps to select data from an opaque type column into an **lvvarchar** 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;
        lvvarchar '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 **lvvarchar** pointer host variable, ***mypoly1**.
2. It calls the **ifx_var_flag()** function to specify that it will let ESQL/C allocate memory for the data buffer (**flag** equals 1). 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 ESQL/C allocated for ***mypoly1**.

Accessing the Internal Format of an Opaque Type

You can access the internal or binary format of an opaque data type with an 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 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.

```
+ fixed binary structure name variable name ;
```

' opaque type '

Element	Purpose	Restrictions	SQL Syntax
<i>opaque type</i>	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 Informix Guide to SQL: Syntax
<i>structure name</i>	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.
<i>variable name</i>	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 10-23.

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.

Informix suggests 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 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 10-4](#)), 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 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 10-6](#)).

```
/* 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 10-21](#) 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 10-4](#) shows, must be available to your 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 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 on indicator variables, see [page 1-42](#).

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 10-21](#) 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 ESQL/C application uses. This SELECT statement also uses a user-defined function called **radius** (see [Figure 10-2 on page 10-5](#)) 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, 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

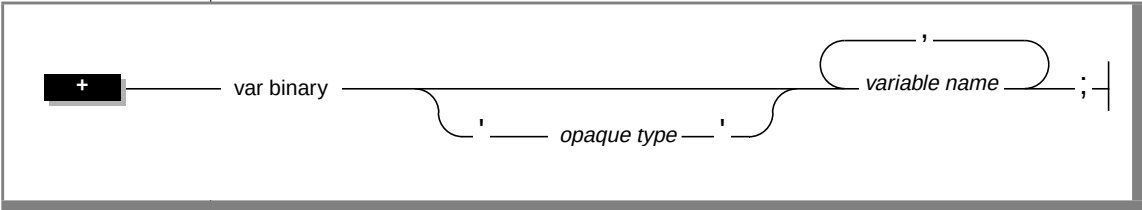
Follow these steps to transfer the internal format of either of these opaque data type columns between the database server and the ESQL/C application:

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.



Element	Purpose	Restrictions	SQL Syntax
<i>opaque type</i>	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 Informix Guide to SQL: Syntax
<i>variable name</i>	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 has been selected from 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 **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 10-24](#) shows.

Using var binary Host Variables

In an 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 ESQL/C programs. That is, you should not access its internal structure directly. Informix does not guarantee that the internal structure of **ifx_varlena_t** will not change in future releases. Therefore, to create portable code, always use the ESQL/C accessor functions for this structure to obtain and store values in the **ifx_varlena_t** structure. For a list of these ESQL/C accessor functions, see “[The lvarchar pointer and var binary Library Functions](#)” on page 10-32.

This section uses a varying-length opaque data type called **image** to demonstrate how an 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_timg_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 10-26](#)).

```
#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 on the **ifx_var_flag()**, **ifx_var_alloc()**, **ifx_var_setdata()**, **ifx_var_getdata()**, and **ifx_var_dealloc()** functions, see [“The Ivarchar pointer and var binary Library Functions” on page 10-32](#).

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 10-27](#) 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 10-26](#) shows, must be available to your 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 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 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 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 on indicator variables, see [page 1-42](#).

- 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 on the **ifx_var_setnull()** function, see [page 10-46](#).

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 10-27](#) 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 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.) 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 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 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 on indicator variables, see [page 1-42](#).

- 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 **circle_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 on the **ifx_var_isnull()** function, see [page 10-41](#).

The Ivarchar pointer and var binary Library Functions

The following library functions are available in ESQL/C to access the data buffer of an **lvvarchar pointer** or **var binary** host variable.

Function Name	Purpose
ifx_var_alloc()	Allocates memory for the data buffer.
ifx_var_dealloc()	Deallocates memory for the data buffer.
ifx_var_flag()	Determines whether ESQL/C or the application handles memory allocation for the data buffer.
ifx_var_getdata()	Returns the contents of the data buffer.
ifx_var_getlen()	Returns the length of the data buffer.
ifx_var_isnull()	Checks whether the data in the data buffer is null.
ifx_var_setdata()	Sets the data for the data buffer.
ifx_var_setlen()	Sets the length of the data buffer.
ifx_var_setnull()	Sets the data in the data buffer to a null value.

These **lvvarchar** 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 ESQL/C programs that use them. The following pages describe each of these functions.

ifx_var_alloc()

The **ifx_var_alloc()** function allocates memory for the data buffer of an **lvarchar** or **var binary** host variable.

Syntax

var binary

```
int ifx_var_alloc(var_bin, var_size)
    var binary **var_bin
    long var_size;
```

lvarchar

```
int ifx_var_alloc(lvar, var_size)
    lvarchar **lvar
    long var_size;
```

var_bin is the address of the **var binary** pointer host variable whose data buffer is allocated.

lvar is the address of the **lvarchar pointer** host variable whose data buffer is allocated.

var_size is the size, in bytes, of the data buffer to allocate.

Usage

The allocation flag of the **ifx_var_flag()** function notifies 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 of a **var binary** host variable with the **ifx_var_alloc()** function.

Important: Whether you allocate memory or allow ESQL/C to allocate the memory for you, you must free the allocated memory using the **ifx_var_dealloc()** function.



ifx_var_alloc()

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

ifx_var_dealloc(), ifx_var_flag()

ifx_var_dealloc()

The **ifx_var_dealloc()** function deallocates the memory that has been allocated for the data buffer of a **var binary** host variable.

Syntax

var binary

```
int ifx_var_dealloc(var_bin)
    var binary **var_bin;
```

lvarchar

```
int ifx_var_dealloc(lar)
    lvarchar **lvar;
```

var_bin is the address of the **var binary** pointer host variable whose data buffer is deallocated.

lvar is the address of the **lvarchar pointer** host variable whose data buffer is allocated.

Usage

The allocation flag of the **ifx_var_flag()** function tells ESQL/C which allocation method to use for the data buffer. Regardless of whether ESQL/C (allocation flag set to 1) or your application (allocation flag set to 0) allocates the memory, you must explicitly deallocate memory that has been allocated to an **lvarchar** or **var binary** host variable.

ifx_var_dealloc()

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

ifx_var_alloc(), ifx_var_flag()

ifx_var_flag()

The **ifx_var_flag()** function determines how memory is allocated for the data buffer of an **lvvarchar** or **var binary** host variable.

Syntax

var binary

```
int ifx_var_flag(var_bin, flag)
    var binary **var_bin;
    int flag;
```

lvvarchar

```
int ifx_var_flag(lvar, flag)
    lvvarchar **lvar;
    int flag;
```

flag is the integer value of the allocation flag, either 0 or 1.

var_bin is the address of the **var binary** host variable.

lvar is the address of the **lvvarchar 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 (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 returns.
- When *flag* is zero (0), 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 **lvvarchar** or **var binary** host variable, ESQL/C allocates memory for its data buffer. Whether you allocate memory for the **lvvarchar** or **var binary** variable, or allow 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

ifx_var_alloc(), **ifx_var_dealloc()**

ifx_var_getdata()

The **ifx_var_getdata()** function returns the data from an **lvvarchar** or **var binary** host variable.

Syntax

var binary

```
void *ifx_var_getdata(var_bin)
    var binary **var_bin;
```

lvvarchar

```
void *ifx_var_getdata(lvar)
    lvvarchar **lvar;
```

var_bin is the address of the **var binary** host variable whose data is retrieved.

lvar is the address of the **lvvarchar pointer** host variable whose data is retrieved.

Usage

The **ifx_var_getdata()** function returns the data as a **void *** pointer. Your ESQL/C application must cast this pointer to the correct data type. When you use **ifx_var_getdata()** on an **lvvarchar 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

ifx_var_getlen(), **ifx_var_setdata()**

ifx_var_getlen()

The **ifx_var_getlen()** function returns the length of the data in an **lvvarchar pointer** or **var binary** host variable.

Syntax

var binary

```
int ifx_var_getlen(var_bin)
    var binary **var_bin;
```

lvvarchar

```
int ifx_var_getlen(lvar)
    lvvarchar **lvar;
```

var_bin is the address of the **var binary** host variable whose length is returned.

lvar is the address of the **lvvarchar 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.

Return Codes

- ≥ 0 The length of the data buffer for the *var_bin* host variable.
- < 0 The function was *not* successful.

Related Functions

ifx_var_getdata(), **ifx_var_setlen()**

ifx_var_isnull()

The **ifx_var_isnull()** function checks whether an **lvvarchar** or **var binary** host variable contains a null value.

Syntax

var binary

```
int ifx_var_isnull(var_bin)
    var binary **var_bin;
```

lvvarchar

```
nt ifx_var_isnull(lvar)
    lvvarchar **lvar;
```

var_bin is the address of the **var binary** host variable.

lvar is the address of the **lvvarchar pointer** host variable.

Usage

The **ifx_var_isnull()** function checks for a null value in an **lvvarchar** or **var binary** host variable. To determine whether an ESQL/C host variable of any other data type contains null, use the **risnull()** library function. For more information, see [page 2-24](#).

Return Codes

- 0 The opaque-type data is *not* a null value.
- 1 The opaque-type data is a null value.

Related Functions

ifx_var_setnull()

ifx_var_setdata()

The **ifx_var_setdata()** function stores data in an **lvvarchar** or **var binary** host variable.

Syntax

var binary

```
int ifx_var_setdata(var_bin, buffer, buf_len)
    var binary **var_bin;
    char *buffer;
    int buf_len;
```

lvvarchar

```
int ifx_var_setdata(lvar, buffer, buf_len)
    lvvarchar **lvar;
    char *buffer;
    int buf_len;
```

buffer is a character buffer that contains the data to store in the *lvar* or *var_bin* host variable.

buf_len is the length, in bytes, of the buffer.

var_bin is the address of the **var binary** host variable.

lvar is the address of the **lvvarchar 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 **lvvarchar pointer** host variable, 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

`ifx_var_getdata()`, `ifx_var_setlen()`

ifx_var_setlen()

The **ifx_var_setlen()** function stores the length of the data buffer for an **lvvarchar** or **var binary** host variable.

Syntax

var binary

```
int ifx_var_setlen(var_bin, length)
var binary **var_bin;
int length
```

lvvarchar

```
int ifx_var_setlen(lvar, length)
lvvarchar **lvar;
int length
```

length is the length, in bytes, of the data buffer to allocate for the **var binary** data.

var_bin is the address of the **var binary** host variable.

lvar is the address of the **lvvarchar 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

ifx_var_getlen(), ifx_var_setdata()

ifx_var_setnull()

The **ifx_var_setnull()** function sets an **lvvarchar** or **var binary** host variable to a null value.

Syntax

var binary

```
int ifx_var_setnull(var_bin, flag)
    var binary **var_bin;
    int flag
```

lvvarchar

```
nt ifx_var_setnull(var_bin, flag)
    var binary **var_bin;
    int flag;
```

var_bin is the address of the **var binary** host variable.

lvar is the address of the **lvvarchar pointer** host variable.

flag is 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 **lvvarchar** or **var binary** to a null value. To set an ESQL/C host variable of any other data type to null, use the **rsetnull()** library function. For more information, see [page 2-27](#).

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

`ifx_var_isnull()`

Accessing Predefined Opaque Data Types

Universal Server implements several built-in data types as *predefined opaque data types*. These data types are opaque data types for which Informix provides the support functions and the database definition. For example, it implements the smart large-object data types, CLOB and BLOB, as an opaque data type called **clob** and **blob**, respectively. 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 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 tabl;
```

For more information on how to access smart large objects, see [Chapter 7, “Working with INFORMIX-Universal Server Smart Large Objects.”](#)

Database Server Communication

Section II



Exception Handling

Obtaining Diagnostic Information After an SQL Statement	11-4
Types of Diagnostic Information	11-4
Types of Database Exceptions	11-4
Descriptive Information	11-5
Types of Status Variables	11-6
Exception Handling with SQLSTATE	11-6
Using GET DIAGNOSTICS	11-7
Statement Information	11-7
Exception Information	11-9
Using the SQLSTATE Variable	11-12
Class and Subclass Codes	11-14
List of SQLSTATE Class Codes	11-15
Checking for Exceptions with SQLSTATE.	11-19
Success in SQLSTATE	11-20
NOT FOUND in SQLSTATE	11-20
Warnings in SQLSTATE	11-22
Runtime Errors in SQLSTATE	11-24
Multiple Exceptions	11-25
Exception Handling with the sqlca Structure	11-27
Fields of the sqlca Structure	11-28
.	11-31
Using the SQLCODE Variable.	11-32
SQLCODE in Pure C Modules	11-33
SQLCODE and the exit() Call	11-33
SQLCODE After a DESCRIBE Statement.	11-33

Checking for Exceptions with sqlca	11-34
Success in sqlca	11-34
NOT FOUND in SQLCODE	11-35
Warnings in sqlca.sqlwarn	11-37
Runtime Errors in SQLCODE.	11-38
Displaying Error Text	11-40
Choosing an Exception-Handling Strategy	11-41
Checking After Each SQL Statement	11-41
The WHENEVER Statement	11-45
Library Functions for Retrieving Error Messages.	11-47
rgetlmsg().	11-48
rgetmsg()	11-51
Displaying Error Text in a Windows Environment	11-54
A Program That Uses Exception Handling	11-55
Compiling the Program	11-55
Guide to the getdiag.ec File.	11-56
Guide to the exp_chk.ec File	11-59

P

roper 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 ESQL/C program must check that every SQL statement executes as you intend. This chapter discusses the following exception-handling topics:

- 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 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 ESQL/C program might generate
- How to choose an exception-handling strategy that consistently handles errors and warnings in your 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 this chapter presents 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.

Obtaining Diagnostic Information After an SQL Statement

After your 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 ESQL/C program
- The two methods that your 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 11-33](#).
- 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 11-7](#).

The [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, a **long** 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

Informix recommends 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. ESQL/C supports the **sqlca** structure and SQLCODE for backward compatibility and for Informix-specific exceptions. (For more information, see [page 11-27](#).)*

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 ESQL/C program. For a full description of the GET DIAGNOSTICS statement, refer to the [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;
```

Figure 11-1 summarizes the *statement_fields* of the diagnostics area.

Figure 11-1
Statement Information from the GET DIAGNOSTICS Statement

Field-Name Keyword	ESQL/C Data Type	Description
NUMBER	int	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 could store in the diagnostics area. At present, the database server always returns an N because the database server can store all exceptions.
ROW_COUNT	int	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 11-2 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;
    int 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 11-2
*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 11-25](#).

For more information on the statement fields of the diagnostics area, see “The Statement Clause” in the GET DIAGNOSTICS statement in the [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 11-25](#).

Figure 11-3 summarizes the *exception_fields* of the diagnostics area.

Figure 11-3
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 11-12 .
INFORMIX_SQLCODE	int	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 11-32 .
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".

(1 of 2)

Field Name Keyword	ESQL/C Data Type	Description
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	int	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 12-30.
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 DATABASE statement. This field is blank when no current 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 12-30.

(2 of 2)

Use the exception information to save detailed information about an exception. The code fragment in Figure 11-4 retrieves exception information on 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];
int 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 11-4
*Example of Using
GET DIAGNOSTICS
to Return Exception
Information*

For more information on the exception fields, see the GET DIAGNOSTICS statement in the [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 ESQL/C header file, **sqlca.h**, declares **SQLSTATE** as a global variable. Since the ESQL/C preprocessor automatically includes **sqlca.h** in an 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.
- 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, 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 11-25](#).

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 "IX000" 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:

- The **INFORMIX_SQLCODE** field of the diagnostics area
- The **SQLCODE** variable ([page 11-32](#))

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 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 11-5, IX is the class code and 000 is the subclass code. The value "IX000" indicates an Informix-specific error.

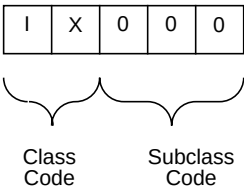


Figure 11-5
The Structure of the SQLSTATE Code with the Value IX000

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 Figure 11-6 summarizes.

Figure 11-6
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	INFORMIX-Universal Server INFORMIX-ESQL / C	Any of the Informix-specific error messages (those that the X / Open or ANSI / ISO reserved range does <i>not</i> support) have an SQLSTATE value of "IX000". If a user-defined routine returns an error message that has been defined by the routine, the SQLSTATE value is "U0001".

List of SQLSTATE Class Codes

Figure 11-7 lists the valid **SQLSTATE** class and subclass values. This figure lists the first entry for each class code in bold.

Figure 11-7
Class and Subclass Codes for SQLSTATE

Class	Subclass	Meaning
00	000	Success
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 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 has been 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

(1 of 4)

Class	Subclass	Meaning
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
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

(2 of 4)

Class	Subclass	Meaning
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
37	000	Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE
3C	000	Duplicate cursor name
40	000	Transaction rollback
40	003	Statement completion unknown

(3 of 4)

Class	Subclass	Meaning
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
U0	001	User-defined error returned by a user-defined routine

(4 of 4)

The ANSI or X/Open standards define all **SQLSTATE** values *except* the following:

- A "IX000" runtime error indicates an Informix-specific error message.
- 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 on non-standard error values, see [“Runtime Errors in SQLSTATE” on page 11-24](#). For more information on non-standard warning values, see [“Warnings in SQLSTATE” on page 11-22](#).

Checking for Exceptions with SQLSTATE

After an SQL statement executes, the **SQLSTATE** value can indicate one of the four conditions that Figure 11-8 shows.

Figure 11-8
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 Diagnostic Information” on page 11-4](#). 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 [Figure 11-10 on page 11-22](#).

The **getdiag** sample program on [page 11-56](#) 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 11-19 on page 11-43](#) 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"). [Figure 11-9](#) lists the conditions that cause SQL statements to yield **NOT FOUND**.

Figure 11-9
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 has been 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"

(1 of 2)

SQL Statement That Generates the Indicated SQLSTATE Result	Result for ANSI-Compliant Database	Result for non-ANSI-Compliant Database
INSERT INTO <i>tablename</i> 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"

(2 of 2)

Figure 11-9 on page 11-20 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 on page 11-56 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 Figure 11-10).
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.

Figure 11-10 lists the Informix-specific warning messages and the SQL statements and conditions that generate the warning.

Figure 11-10
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.

(1 of 2)

Warning Value	SQL Statement	Warning Condition
"01I04"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	Your application opened a database that uses the Universal Server or OnLine database server.
"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 (<i>only</i> 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) has been skipped during query processing.

(2 of 2)

For a list of the ANSI and X/Open warning messages, see [“List of SQLSTATE Class Codes” on page 11-15.](#)

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 11-11 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(!strcmp(SQLSTATE, "02", 2)) /* < 0 is an error */
    err_chk(msg);
if (!strcmp(SQLSTATE, "01", 2))
{
    if (!strcmp(SQLSTATE[2], "I01", 3))
        trans_db = 1;
    if (!strcmp(SQLSTATE[2], "I03", 3))
        ansi_db = 1;
    if (!strcmp(SQLSTATE[2], "I04", 3))
        online_db = 1;
}
```

Figure 11-11
*Using SQLSTATE to
Check for
Informix-Specific
Warnings*

The code fragment in Figure 11-11 checks **SQLSTATE** with the system **strcmp()** function. The **getdiag** sample program ([page 11-56](#)) 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 12-26](#).

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. [Figure 11-7 on page 11-15](#) 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.

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 11-12 on page 11-26](#) 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);  
    printf("TEXT: %s\n",message);  
    printf("MESSAGE LENGTH: %d\n",messlen);  
}
```

Figure 11-12
*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 ([page 11-56](#)) 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 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: *ESQL/C supports the **sqlca** structure for backward compatibility. Informix recommends, 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 (page 11-6).*

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

Figure 11-13 contains the declaration of the **sqlca** structure from the **sqlca.h** header file. The ESQL/C preprocessor automatically includes the **sqlca.h** header file in an ESQL/C program.

```
struct sqlca_s
{
    long sqlcode;
    char sqlerrm[72]; /* error message parameters */
    char sqlerrp[8];
    long sqlerrd[6];
        /* 0 - estimated number of rows returned */
        /* 1 - serial value after insert or ISAM error code */
        /* 2 - number of rows processed */
        /* 3 - estimated cost */
        /* 4 - offset of the error into the SQL statement */
        /* 5 - rowid after insert */
    struct sqlcaw_s
    {
        char sqlwarn0; /* = W if any of sqlwarn[1-7] = W */
        char sqlwarn1; /* = W if any truncation occurred or
            database has transactions */
        char sqlwarn2; /* = W if a null value returned or
            ANSI database */
        char sqlwarn3; /* = W if no. in select list != no. in into list or
            OnLine backend */
        char sqlwarn4; /* = W if no where clause on prepared update, delete, or
            incompatible float format */
        char sqlwarn5; /* = W if non-ANSI statement */
        char sqlwarn6; /* = W if server is in data replication secondary mode */
        char sqlwarn7; /* reserved */
    } sqlwarn;
};

extern struct sqlca_s sqlca;

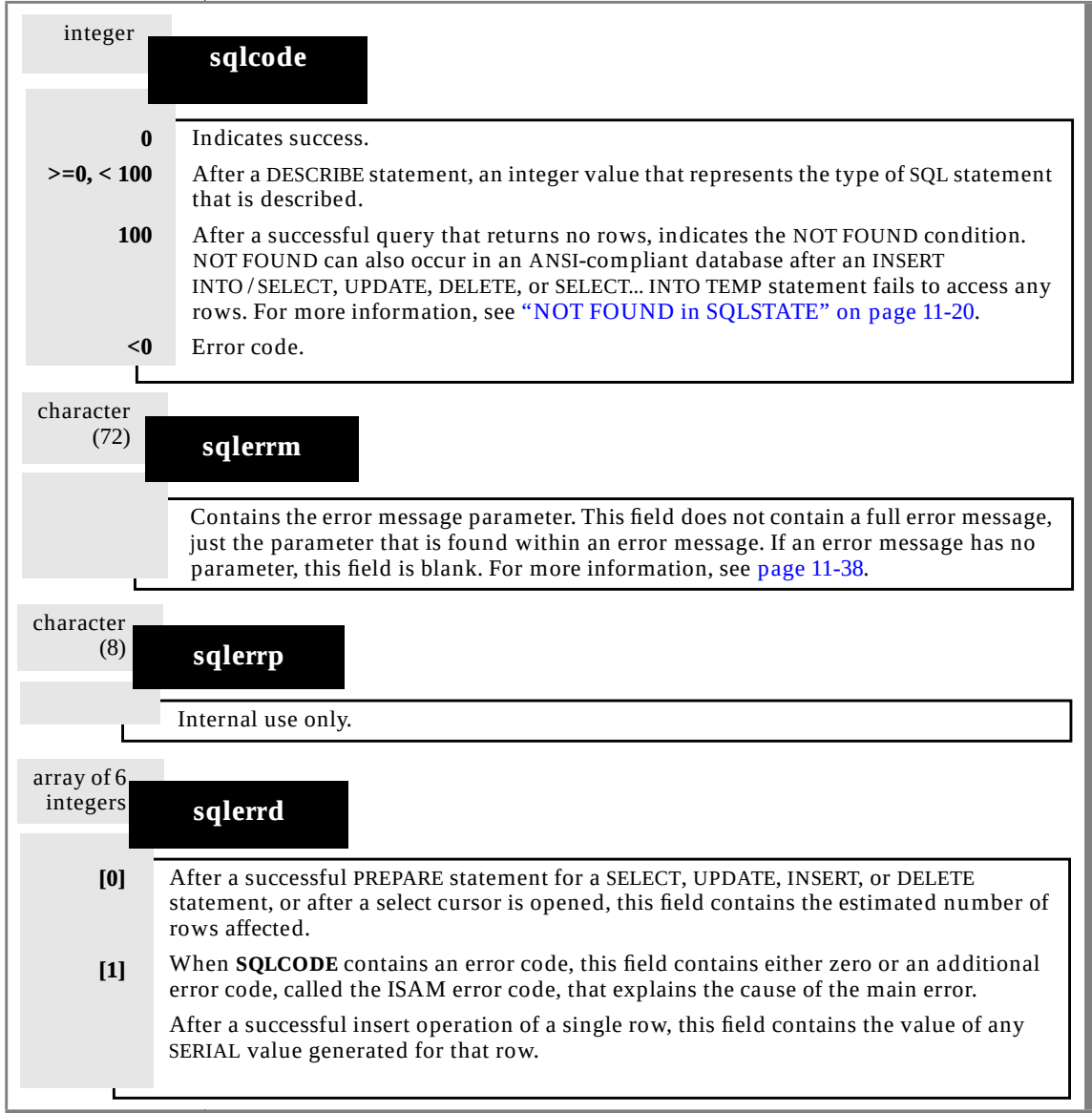
extern long SQLCODE;

extern char SQLSTATE[];
```

Figure 11-13
*Declaration of sqlca
in the sqlca.h
Header File*

Figure 11-14 illustrates the fields of the **sqlca** structure.

Figure 11-14
Fields of the *sqlca* Structure



cont'd

sqlerrd

- [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 has been 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.

array of 8
characters

sqlwarn

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 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 ESQL/C in the [Informix Guide to GLS Functionality](#).

cont'd

sqlwarn**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 a **long** integer that indicates whether the SQL statement succeeded or failed. The ESQL/C header file, **`sqlca.h`**, declares **`SQLCODE`** as a global variable. Since the ESQL/C preprocessor automatically includes **`sqlca.h`** in an 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. ESQL/C copies the value of **`sqlca.sqlcode`** to the global **`SQLCODE`** variable.

Tip: For readability and brevity, use **`SQLCODE`** in your ESQL/C program in place of **`sqlca.sqlcode`**.

The **`SQLCODE`** value can indicate the following types of exceptions:

<code>SQLCODE</code> = 0	Success
<code>SQLCODE</code> = 100	NOT FOUND condition
<code>SQLCODE</code> < 0	Runtime error

For information about the values of **`SQLCODE`** (and **`sqlca.sqlcode`**) and their corrective actions, refer to the *Informix Error Messages* manual. For information about how to handle these exceptions, see [“Checking for Exceptions with `sqlca`”](#) on page 11-34.

The following sections provide additional information about **`SQLCODE`**.



SQLCODE in Pure C Modules

To return the same values that the **SQLCODE** status variable in ESQL/C modules returns, you can use **SQLCODE** in pure C modules (modules with the `.c` extension) that you link to an ESQL/C program. To use **SQLCODE** in a pure C module, declare **SQLCODE** as an external variable, as follows:

```
extern long 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 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 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 on the actual error, use a temporary file, a database table, or some form of inter-process 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 **`sqltype.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 Statement Type” on page 15-16](#).

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 Figure 11-15 shows.

Figure 11-15
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 Diagnostic Information” on page 11-4](#). 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 11-37](#).

For more information on these additional fields, see [“Fields of the sqlca Structure” on page 11-28](#). In addition, the **SQLCODE** value for some SQL statements has special meaning. For more information, see [“Using the SQLCODE Variable” on page 11-32](#).

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. Figure 11-16 lists conditions that cause SQL statements to yield NOT FOUND.

Figure 11-16
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 has been 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

(1 of 2)

SQL Statement Where SQLCODE Gets the Indicated Result	Result for ANSI-Compliant Database	Result for Non-ANSI-Compliant Database
INSERT INTO <i>tablename</i> 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

(2 of 2)

Figure 11-16 on page 11-35 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. [Figure 11-14 on page 11-29](#) contains two tables that describe the fields of the ***sqlca.sqlwarn*** structure and their associated warning conditions. The first ***sqlwarn*** table in [Figure 11-14](#) lists the warnings that occur after the database server opens a database or establishes a connection. For more information, see [“Determining Features of the Database Server” on page 12-26](#).

The second ***sqlwarn*** table in [Figure 11-14](#) lists warnings that other SQL statements might generate.

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 block of code that [Figure 11-17](#) 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 11-17
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 *Informix Error Messages* manual lists these Informix-specific error codes and their corrective actions. You can also use the command-line **finderr** utility to obtain information about an Informix error. For more information about **finderr**, see the *Informix Error Messages* manual.

From within your 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 11-47.

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** to save an error message parameter. This value occupies a **%s** parameter in the 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 11-39.

Tip: You can also test for errors with the **WHENEVER SQLERROR** statement. For more information, see [“The WHENEVER Statement”](#) on page 11-45.



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 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 one:

```
" 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 ESQL/C preprocessor has also dropped a new-line 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.

WIN NT/95

Displaying Error Text

Your 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 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 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 11-18 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;
    printf("SQLSTATE: %s, %s\n", SQLSTATE, message);
    exit(1);
}
```

Figure 11-18
*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 11-19 on page 11-43](#) 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.

Figure 11-19*Example of an Exception-Handling Function That Uses SQLSTATE*

```

EXEC SQL select * from customer where fname not like "%y";
sqlstate_exception("select");
:
:
long 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);
}

```

The **sqlstate_exception()** function, which [Figure 11-19 on page 11-43](#) 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 [page 11-62](#).)
- 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 could 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".

Do not use the WHENEVER ERROR STOP construction in an 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 ESQL/C preprocessor uses CONTINUE as the default action. To execute the **sqlstate_exception()** function (shown in [Figure 11-19 on page 11-43](#)) 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 [Informix Guide to SQL: Syntax](#).



Library Functions for Retrieving Error Messages

Each **SQLCODE** value has an associated message. Informix message files in the **\$INFORMIXDIR/msg** directory store the message number and its text. This information, along with corrective actions, also appears in the *Informix Error Messages* manual.

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 11-9](#).

The following pages describe the **rgetlmsg()** and **rgetmsg()** functions.

Important: Use **rgetlmsg()** in any new ESQL/C code that you write. ESQL/C provides the **rgetmsg()** function primarily for backward compatibility.



rgetlmsg()

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

```
int rgetlmsg(msgnum, msgstr, lenmsgstr, msglen)
    long msgnum;
    char *msgstr;
    int lenmsgstr;
    int *msglen;
```

<i>msgnum</i>	is the error number. The four-byte parameter provides for the full range of Informix-specific error numbers.
<i>msgstr</i>	is a pointer to the buffer that receives the message string (the output buffer).
<i>lenmsgstr</i>	is the size of the <i>msgstr</i> output buffer. Make this value the size of the largest message that you expect to retrieve.
<i>msglen</i>	is a pointer to the integer 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 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 more information, see the *Informix Error Messages* manual.

Example

This sample program is in the **rgetlmsg.ec** file in the 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()
{
    int 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.

Example Output

```
RGETLMSG Sample ESQL Program running.
```

```
Error -310: Table (informix.customer) already exists in  
database.
```

```
RGETLMSG Sample Program over.
```

rgetmsg()

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 ESQL/C code.

Syntax

```
int rgetmsg(msgnum, msgstr, lenmsgstr)
    short msgnum;
    char *msgstr;
    short lenmsgstr;
```

- msgnum** is the error number. The two-byte parameter restricts error numbers to between -32768 and +32767.
- msgstr** is a pointer to the buffer that receives the message string (the output buffer).
- lenmsgstr** is 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: ESQL/C supports the **rgetmsg()** function for backward compatibility. Some Informix error numbers currently exceed the range that the **short** integer, **msgnum**, supports. Informix recommends the **rgetlmsg()** function, which supports **long** integers as error numbers, 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 more information, see the *Informix Error Messages* manual.

Example

This sample program is in the **rgetmsg.ec** file in the 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");
}

```

Example Output

```
RGETMSG Sample ESQL Program running.
```

```
Error -310: Table (informix.customer) already exists in  
database.
```

```
RGETMSG Sample Program over.
```

Displaying Error Text in a Windows Environment

Your 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 manual 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 ([page 11-56](#)) includes the **exp_chk.ec** file ([page 11-59](#)).

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 on the **esql** preprocessor command, see [“Using the esql Command” on page 1-47](#).

Guide to the getdiag.ec File

This section describes only the exception-handling statements. For information about the other statements, refer to [“A Sample INFORMIX-ESQL/C Program”](#) in [Chapter 1](#) of this manual.

```
1 #include <stdio.h>

2 EXEC SQL define FNAME_LEN 15;
3 EXEC SQL define LNAME_LEN 15;

4 extern char statement[20];

5 main()
6 {
7 EXEC SQL BEGIN DECLARE SECTION;
8     char fname[ FNAME_LEN + 1 ];
9     char lname[ LNAME_LEN + 1 ];
10 EXEC SQL END DECLARE SECTION;

11     EXEC SQL whenever sqlerror CALL whenexp_chk;
12     EXEC SQL whenever sqlwarning CALL whenexp_chk;

13     printf("GETDIAG Sample ESQL program running.\n\n");
14     strcpy (statement, "CONNECT stmt");
15     EXEC SQL connect to 'stores7';

16     strcpy (statement, "DECLARE stmt");
17     EXEC SQL declare democursor cursor for
18         select fname, lname
19         into :fname, :lname;
20         from customer
21         where lname < 'C';

22     strcpy (statement, "OPEN stmt");
23     EXEC SQL open democursor;

24     strcpy (statement, "FETCH stmt");
25     for (;;)
26     {
27         EXEC SQL fetch democursor;
28         if(sqlstate_err() == 100)
29             break;

30         printf("%s %s\n", fname, lname);
31     }

32     strcpy (statement, "CLOSE stmt");
33     EXEC SQL close democursor;
```

Continued on page 11-58

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 171 to 215](#)” on [page 11-71](#)).

Lines 11 and 12

The `WHENEVER SQLERROR` statement tells the 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 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 14

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 16, 22, 24, and 32

Lines 16, 22, 24, and 32 copy the name of the current SQL statement into the **statement** variable before the `DECLARE`, `OPEN`, `FETCH`, and `CLOSE` statements, respectively, execute. This action enables the **whenexp_chk()** function to identify the statement that failed if an error occurs.

```
34 strcpy (statement, "FREE stmt");
35     EXEC SQL free democursor;

36     strcpy (statement, "DISCONNECT stmt");
37     EXEC SQL disconnect current;

38     printf("\nGETDIAG Sample Program Over.\n");
39 }      /* End of main routine */

40 EXEC SQL include exp_chk.ec;
```

Lines 34 and 36

These lines copy the name of the current SQL statement into the **statement** variable before the FREE and DISCONNECT statements, respectively, execute. The **whenexp_chk()** function uses the **statement** variable to identify the statement that failed if an error occurs.

Line 40

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 **exp_chk.ec** source file. The **getdiag** program must include this file with the ESQL/C **include** directive because the exception-handling functions use ESQL/C statements. For a description of the **exp_chk.ec** source file, see [page 11-59](#).



Tip: In a production environment, you would put functions such as **whenexp_chk()** into a library and include this library on the command line when you compile the ESQL/C program.

Guide to the *exp_chk.ec* File

The **exp_chk.ec** file contains exception-handling functions for the 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 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 8](#)) 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 int sqlstate_err()
11 {
12     int 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 }
```

Continued on [page 11-62](#)

Lines 1 to 4

These ESQ/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, respectively. The **sqlstate_err()** function checks the first two characters of the global **SQLSTATE** variable. Because 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     int j;

40     EXEC SQL BEGIN DECLARE SECTION;
41         int 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         int messlen;
48         char sqlstate_code[6];
49         int 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 }

```

Continued on page 11-63

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 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 int sqlerr_code;
98 int 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 }
```

Continued on page 11-66

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
119 * The exp_chk() function calls sqlstate_err() to check the SQLSTATE
120 * status variable to see if an error or warning has occurred following
121 * an SQL statement. If either condition has occurred, exp_chk()
122 * calls disp_sqlstate_err() to print the detailed error information.
123 *
124 * This function handles exceptions as follows:
125 *   runtime errors - call exit()
126 *   warnings - continue execution, returning "1"
127 *   success - continue execution, returning "0"
128 *   Not Found - continue execution, returning "100"
129 */
130 long exp_chk(stmt, warn_flg)
131 char *stmt;
132 int warn_flg;
133 {
134     int sqlerr_code = SUCCESS;

135     sqlerr_code = sqlstate_err();
136     disp_exception(stmt, sqlerr_code, warn_flg);

137     if(sqlerr_code == RTERROR)    /* Exception is a runtime error */
138     {
139         /* Exit the program after examining the error */
140         printf("*****Program terminated*****\n\n");
141         exit(1);
142     }
143     /*else                                /* Exception is "success", "Not Found", */
144         return(sqlerr_code);              /* or "warning" */
145 }
```

Continued on page 11-68

Lines 118 to 145

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 nonserious 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 **dyn_sql** sample program (see “[The dyn_sql Program](#)” on page 16-47) uses **exp_chk()** to handle exceptions.

The **sqlstate_err()** function (line 135) determines the type of exception that **SQLSTATE** contains. The function then calls **disp_exception()** (line 136) 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 137 to 142) to terminate the program. This behavior is the same as what the **whenexp_chk()** function (see lines 171 to 215) provides for runtime errors.

```
146
147 * The exp_chk2() function calls sqlstate_err() to check the SQLSTATE
148 * status variable to see if an error or warning has occurred following
149 * an SQL statement. If either condition has occurred, exp_chk2()
150 * calls disp_sqlstate_err() to print the detailed error information.
151 *
152 * This function handles exceptions as follows:
153 *   runtime errors - continue execution, returning SQLCODE (<0)
154 *   warnings - continue execution, returning one (1)
155 *   success - continue execution, returning zero (0)
156 *   Not Found - continue execution, returning 100
157 */
158 long exp_chk2(stmt, warn_flg)
159 char *stmt;
160 int warn_flg;
161 {
162   int sqlerr_code = SUCCESS;
163   long sqlcode;

164   sqlcode = SQLCODE;   /* save SQLCODE in case of error */
165   sqlerr_code = sqlstate_err();
166   disp_exception(stmt, sqlerr_code, warn_flg);

167   if(sqlerr_code == RTERROR)
168 /*   sqlerr_code = sqlcode;

169   return(sqlerr_code);
170 }
```

Continued on page 11-70

Lines 146 to 170

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 141), while the **exp_chk2()** function returns the **SQLCODE** value to the calling program (lines 167 to 168).

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 function, which [Chapter 12](#) describes, uses **exp_chk2()** to handle exceptions.

```
171 *
172 * The whenexp_chk() function calls sqlstate_err() to check the SQLSTATE
173 * status variable to see if an error or warning has occurred following
174 * an SQL statement. If either condition has occurred, whenerr_chk()
175 * calls disp_sqlstate_err() to print the detailed error information.
176 *
177 * This function is expected to be used with the WHENEVER SQLERROR
178 * statement: it executes an exit(1) when it encounters a negative
179 * error code. It also assumes the presence of the "statement" global
180 * variable, set by the calling program to the name of the statement
181 * encountering the error.
182 */
183 whenexp_chk()
184 {
185     int sqlerr_code = SUCCESS;
186     int disp = 0;

187     sqlerr_code = sqlstate_err();

188     if(sqlerr_code == WARNING)
189     {
190         disp = 1;
191         printf("\n*****Warning encountered in %s*****\n",
192             statement);
193     }
194     else
195     {
196         if(sqlerr_code == RTERROR)
197         {
198             printf("\n*****Error encountered in %s*****\n",
199                 statement);
200             disp = 1;
201         }
202         if(disp)
203             disp_sqlstate_err();

204         if(sqlerr_code == RTERROR)
205         {
206             /* Exit the program after examining the error */
207             printf("*****Program terminated*****\n\n");
208             exit(1);
209         }
210         else
211         {
212             if(sqlerr_code == WARNING)
213                 printf("\n*****Program execution continues*****\n\n");
214             return(sqlerr_code);
215         }
216     }
217 }
```

Lines 171 to 215

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 has generated the exception (lines 192 and 198).

To use arguments with the **whenexp_chk()** function, you could 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 could 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 ([page 11-56](#)).

The **sqlstate_err()** function (line 187) returns an integer that indicates the success of the most-recently executed SQL statement. This return value is based on the **SQLSTATE** value.

Lines 188 to 200 display a special line to bring attention to the exception information that has been 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 191 and 197) 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 201 and 202) displays the information that the diagnostics area contains *only* if **SQLSTATE** indicates a warning or a runtime error (**disp** = 1).

Lines 203 to 208 handle a runtime error. They notify the user of the program termination and then use the **exit()** system call (line 207) to terminate the program. The call to the **disp_sqlstate_err()** function (line 202) has already displayed information about the cause of the runtime error.

Working with the Database Server

The Client-Server Architecture of ESQL/C Applications	12-3
The Client-Server Connection	12-5
Sources of Connection Information on a UNIX	
Operating System	12-6
Accessing the sqlhosts File	12-6
Specifying the Default Database Server	12-7
Sources of Connection Information in a Windows	
Environment	12-7
Setting Environment Variables for Connection in a	
Windows Environment	12-8
sqlhosts Information in a Windows Environment.	12-9
Using a Central Registry	12-12
Connection Authentication Functionality in a Windows	
Environment	12-13
Connecting to a Database Server.	12-15
Establishing a Connection.	12-15
Establishing an Explicit Connection in a Windows	
Environment	12-19
Using Multiplexed Connections.	12-22
Identifying the Database Server	12-23
Interacting with the Database Server.	12-26
Determining Features of the Database Server	12-26
Switching Between Multiple Database Connections	12-28
Making a Connection Current	12-28
Handling Transactions	12-29
Identifying an Explicit Connection	12-30
Obtaining Available Databases	12-32
Checking the Status of the Database Server	12-32
Detaching from a Connection	12-32

Interrupting an SQL Request	12-33
Interruptible SQL Statements.	12-33
Allowing a User to Interrupt	12-34
Setting Up a Time-Out Interval	12-36
Terminating a Connection	12-40
Optimized Message Transfers	12-41
Restrictions on Optimized Message Transfers	12-41
Enabling Optimized Message Transfers	12-42
Setting the OPTMSG Environment Variable.	12-42
Setting the OptMsg Global Variable	12-43
Error Handling with Optimized Message Transfers	12-44
Using Database Server Control Functions	12-46
ifx_getcur_conn_name().	12-47
sqgetdbs().	12-48
sqlbreak().	12-51
sqlbreakcallback().	12-53
sqldetach().	12-56
sqldone().	12-63
sqlexit().	12-64
sqlsignal().	12-65
sqlstart().	12-67
The timeout Program	12-69
Compiling the Program	12-69
Guide to the timeout.ec File	12-70
Example Output	12-88
Using ESQL / C Connection Library Functions in a Windows	
Environment	12-92
GetConnect().	12-94
ReleaseConnect().	12-97
SetConnect().	12-99

T

his chapter explains how an INFORMIX-ESQL/C program can interact with a database server. It contains the following information:

- A description of the client-server architecture of an ESQL/C application
- An overview of the ways an ESQL/C program can interact with the database server
- The syntax of the ESQL/C library functions that control the database server

The end of this chapter presents 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 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 ESQL/C program and the database server communicate with each other through an interprocess-communication mechanism. The 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 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 ESQL/C application and local database servers.

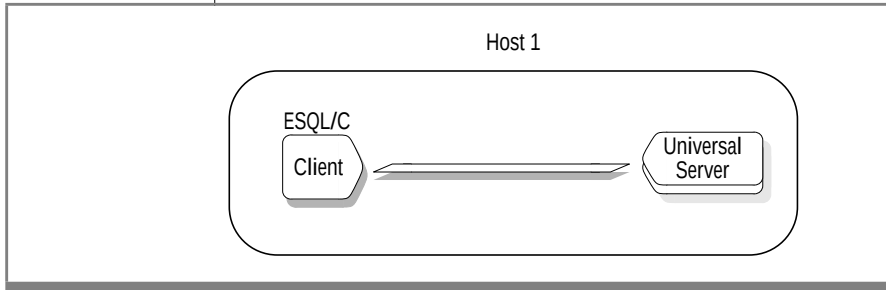


Figure 12-1
*ESQL/C Application
That Connects to a
Local Database
Server*

Figure 12-2 illustrates an ESQL/C application that connects across a network to a remote database server.

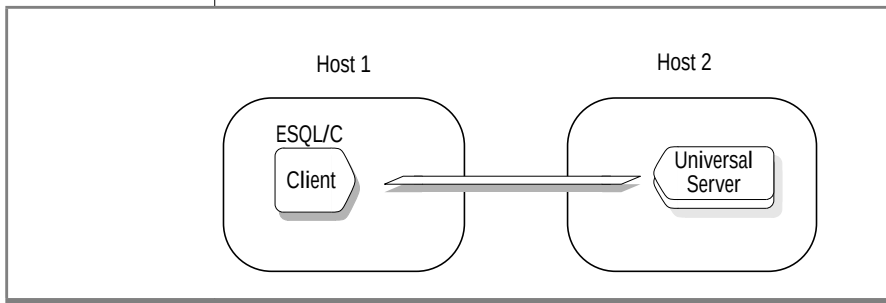


Figure 12-2
*ESQL/C Application
That Connects to a
Remote Database
Server*

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 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. 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 will be 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 to use to establish the database connection.

The client application connects to the default database server when the application does not explicitly specify a database server for the connection. For more information, see [“The Default Database Server” on page 12-25](#).

Important: You must set the **INFORMIXSERVER** environment variable even if the application does not establish a connection to the default database server.



GLS

The client also sends environment variables so that the database server can determine the server-processing locale. For more information on how the database server establishes the server-processing locale, see the [Informix Guide to GLS Functionality](#).

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 on how to set environment variables, see the [Informix Guide to SQL: Reference](#) for your operating system.

UNIX

Sources of Connection Information on 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, refer to the [Informix Guide to SQL: Reference](#) and to the [Administrator's Guide](#) for your database server.

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 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. The [Administrator's Guide](#) for your database server describes how to create a database server entry in the **sqlhosts** file.

Specifying the Default Database Server

For your 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 on **sqlhosts**, see [“Accessing the sqlhosts File” on page 12-6](#). ♦

WIN NT/95

Sources of Connection Information in a Windows Environment

To establish a connection to a database server, an ESQL/C application in a Windows environment can perform 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
- Establish a connection to the database server

In Windows environments, 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**, ESQL/C sends this value to the database server. For any field the application has *not* set in the **InetLogin** structure, 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 1-97](#) and [“Precedence of Configuration Values” on page 1-111](#).)

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 on UNIX operating systems)

When a client ESQL/C application establishes a connection to a database server (usually with the SQL statements CONNECT or DATABASE. SEE [“Connecting to a Database Server” on page 12-15](#)), 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, refer to the [Informix Guide to SQL: Reference](#) and the [Informix Guide to GLS Functionality](#). To change the value of an environment variable, use the Environment tab of the **Setnet32** utility, which the [Informix Client Products Installation Guide](#) describes.

To change the value of an environment variable in the Registry, use the Environment tab of the **Setnet32** utility, which the [Informix Client Products Installation Guide](#) describes. For more information on how to change the environment variable for the current process, use the `ifx_putenv()` function. See “[Setting and Retrieving Environment Variables in Windows Environments](#)” on page 1-97. For more information on **InetLogin**, see “[InetLogin Structure](#)” on page 1-103.

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.

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, ESQL/C obtains the values from the in-memory copy of the Registry. (For more information, see [“Precedence of Configuration Values” on page 1-111.](#))

For example, the code fragment in Figure 12-3 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*, respectively, 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*, respectively, for the **bcksrvr** database server).

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 ESQL/C application.
- The central registry is a Registry that two or more ESQL/C applications can access to obtain **sqlhosts** information.

The central registry can be on the Domain Server or on any Windows NT or Windows 95 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 ESQL/C applications in Windows environments.

WIN 95

To use a central registry in Windows 95, you must install special Microsoft software to support remote registry services. For more information, see your Informix Release Notes for ESQL/C and your Microsoft documentation. ♦

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 1-97](#)), or the **Inetlogin** structure ([“InetLogin Structure” on page 1-103](#)).

In a Windows environment, an 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 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 Figure 12-4).
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 Figure 12-4 shows.

Figure 12-4
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

(1 of 2)

HostInfoStruct Field	Data Type	Purpose
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]	<i>Reserved for future use</i>

(2 of 2)

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 12-5 shows the default **sqlauth()** function, which the **esqlauth.c** file contains.

Figure 12-5
Default Code for sqlauth() Function

```
BOOL __declspec( dllexport ) sqlauth ( HostInfoStruct *pHostInfo )
{
    return TRUE;
}
```

This default action of **sqlauth()** means that 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 [Figure 12-4 on page 12-13](#), 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.
3. 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** demo program in the `%INFORMIXDIR%\demo\esqlauth` directory. ♦

Connecting to a Database Server

When an 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 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: Informix recommends 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 backward compatibility.

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 CONNECTION`) 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 [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 **INFORMIX-SERVER** 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 **ESQL/C** source file with the **-ansi** preprocessor option

The **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 ([page 12-16](#)), 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 on implicit connections, see the CONNECT statement in the [Informix Guide to SQL: Syntax](#).

Summary of Connection Types

Figure 12-6 summarizes the methods that ESQL/C supports to connect to a database server.

Figure 12-6
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	✓		✓	✓
CREATE DATABASE	✓		✓	✓
START DATABASE	✓		✓	✓
DROP DATABASE	✓		✓	
sqlstart()	✓		✓	
CONNECT TO DEFAULT		✓	✓	
CONNECT TO '@servername'		✓	✓	
CONNECT TO 'dbname'		✓	✓	✓
CONNECT TO dbname@srvname where dbname@srvname is 'dbname@servername'		✓	✓	✓

For information on the **sqlstart()** library function, see [page 12-67](#).

WIN NT/95

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 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 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 ESQL/C modules (either **.exe** or **.dll**) to use the same connection to manipulate database data

Figure 12-7 on page 12-21 shows scenarios in which multiple applications use the same connection to a database server.

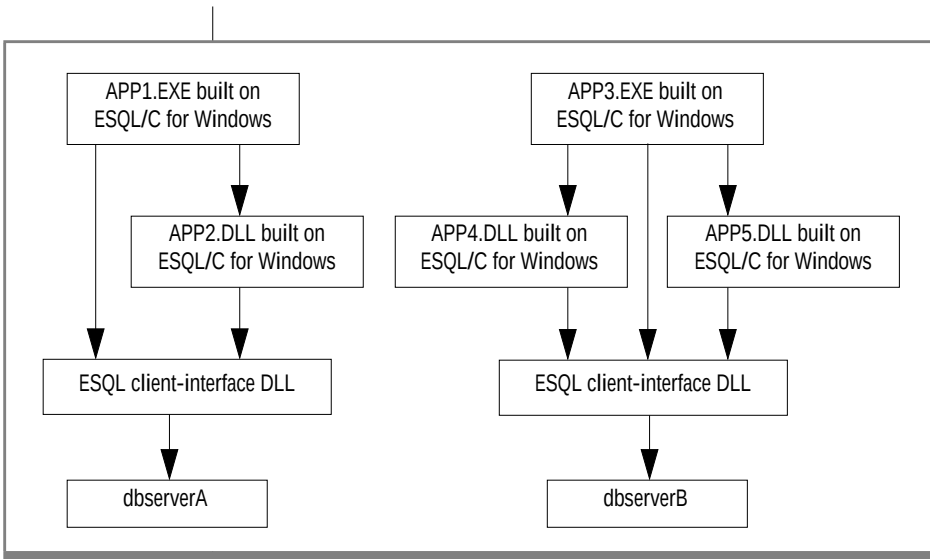
- When you want one ESQL/C module to create two or more connections to one or more databases, which includes sharing an ESQL DLL between two C applications

Figure 12-8 on page 12-21 shows a single application that establishes connections to multiple database servers.

Figure 12-7 on page 12-21 shows the following two scenarios in which multiple applications share a single connection to the database server:

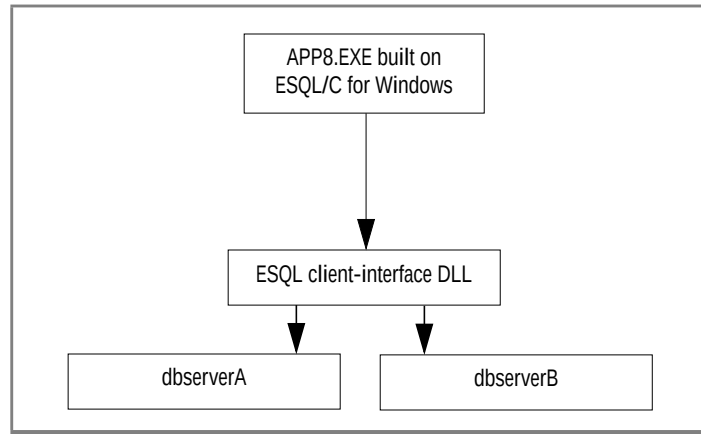
- The scenario on the left requires that **APP1.EXE** establish an explicit connection to the **dbserverA** database server. Once this connection is established, APP1 can pass the connection information required to set the connection in the APP2 DLL.
- The scenario on the right requires that **APP3.EXE** 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 12-7
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 12-8 shows.

Figure 12-8
One Application That Uses Connections to More Than One Database Server at the Same Time



◆

Using Multiplexed Connections

A multiplexed connection enables an 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 (CONNECT statement) creates a new database-server connection.

Client Requirements for Execution

To implement a multiplexed connection, set the multiplexing option, in the client **sqlhosts** file, 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_01	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 ESQL/C, you must relink it. Applications that you compiled prior to version 9.13 of ESQL/C can connect to a multiplexing database server, however. The database server establishes a non-multiplexed connection in this case.

For information on the requirements to implement multiplexing on the database server, see the [Administrator's Guide](#) for your database server.

Limitations for Multiplexed Connections

ESQL/C imposes the following limitations on multiplexed connections:

- Shared memory connections are not supported.
- Multithreaded applications, those compiled when the `IFX_THREAD` environment variable is set, 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 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 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;  
EXEC SQL database '/usr/dbapps/stores7@valley'; (On a  
UNIX system)  
EXEC SQL database 'C:\usr\dbapps\stores@valley'; (On a  
Windows system)
```

For more information about connections to a specific database server, see the CONNECT and DATABASE statements in the [Informix Guide to SQL: Syntax](#).

The Default Database Server

An 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;
EXEC SQL connect to '/usr/dbapps/stores7'; (On a UNIX
system)
EXEC SQL connect to 'C:\usr\dbapps\stores7'; (ON a
Windows system)
```

- 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 12-7](#).



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 [Informix Guide to SQL: Reference](#).

Interacting with the Database Server

Within your 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. Some actions taken on a database server require that a database server process be busy or 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.

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:

- 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
Database has transactions	"01I01"	sqlca.sqlwarn.sqlwarn1 is 'W'
Database is ANSI compliant	"01I03"	sqlca.sqlwarn.sqlwarn2 is 'W'
Database server is Universal Server, OnLine Dynamic Server, or OnLine Workgroup Server	"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	<i>none</i>	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.”](#)

Switching Between Multiple Database Connections

An 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 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 `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.
- The `SET CONNECTION` statement switches to a *dormant* connection and makes it 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 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 on thread-safe applications, see [Chapter 13, “Using Informix Libraries.”](#)

For more information, see the entries for `CONNECT` and `SET CONNECTION` in the [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.



Tip: For a sample ESQL/C program that uses the WITH CONCURRENT TRANSACTION clause, see the CONNECT statement in the “[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 on how to handle transactions, see the COMMIT WORK and ROLLBACK WORK statements. For more information on the WITH CONCURRENT TRANSACTION clause of CONNECT, see the CONNECT and SET CONNECTION statements. Entries for these SQL statements can be found in the [Informix Guide to SQL: Syntax](#).

Identifying an Explicit Connection

From within an 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, respectively.

Figure 12-9 shows a code fragment that saves connection information in the **srvrname** and **cnctname** host variables.

```
EXEC SQL connect to :dbname;
if(!strcmp(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 12-9
*Code Fragment That
Saves Connection
Information*

For more information, see the entry for GET DIAGNOSTICS in the [Informix Guide to SQL: Syntax](#).

From within an 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 ESQL/C application that has multiple threads.

For example, the following code consists of a callback function, **cb()**, that is used by two **sqlbreakcallback()** calls in two different threads:

```
void
cb(int status)
{
    int 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;
        int 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;
        int 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 [page 12-47](#). For more information on current and active connections, see [“Concurrent Active Connections” on page 13-18](#).

Obtaining Available Databases

From within an 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 on **sqgetdbs()**, see [page 12-48](#).

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 on **sqldone()**, see [page 12-63](#).

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 on the **sqldetach()** library function, see [page 12-56](#).

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 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 time-out 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 on the **sqlbreak()** function, see [page 12-51](#).

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 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 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 ESQL/C control functions **sqlbreak()** and **sqldone()**. For more information on the **sqlbreak()** and **sqldone()** functions, refer to [page 12-51](#) and [page 12-63](#), respectively. If you use any other ESQL/C control function or any SQL statement in the signal handler while the database server is processing, ESQL/C generates an error (-439).

The 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. Only use **longjmp()** in a signal-handling function if **squidone()** returns 0 (the database server is idle).

See your UNIX operating system documentation for more information on 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 ESQL/C application receives the SIGINT signal, it calls the function that *sigfunc_ptr* indicates. For more information on 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 on how your system handles signals, check your system documentation.



Setting Up a Time-Out 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 *time-out 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 time-out interval has elapsed.



Warning: Do not use the **sqlbreakcallback()** function if your ESQL/C application uses shared memory (**olipcshm**) as the nettype in a connection to an instance of INFORMIX-Universal 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 Time-Out Interval

With the **sqlbreakcallback()** function, you specify a *time-out interval*. A time-out 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 on how the database server responds to **`sqlbreak()`**, see [page 12-51](#).
- The next time-out interval elapses. When the application resumes execution, it calls the callback function again.

The application calls the callback function each time the time-out 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 ESQL/C function that specifies actions to take during execution of an SQL request. This function must have the following declaration:

```
void callbackfunc(status)
int status;
```

The **integer** *status* variable identifies at what point in the execution of the SQL request the callback function has been called. Within the callback function, you can check this *status* variable to determine at which point the function has been called. [Figure 12-10 on page 12-38](#) summarizes the valid *status* values.

Figure 12-10
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 time-out 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 on message requests, see [“Interruptible SQL Statements” on page 12-33](#).

The callback function, and any of its subroutines, can contain only the following 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 on the **sqldone()** function, refer to [page 12-63](#).

- The **sqlbreakcallback()** library function disassociates the callback function from the time-out 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 ESQL/C control function other than those in the preceding list, or if you use any SQL statement while the database server is processing, ESQL/C generates an error (-439).

If the application calls a callback function because a time-out 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 time-out 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 [page 12-33](#)). 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 on how the database server responds to an interrupt request, see the description of **sqlbreak()** on [page 12-51](#).

Use the **sqlbreakcallback()** function to set the time-out 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 [page 12-37](#)). 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 on the **sqlbreakcallback()** function, see [page 12-53](#). The **timeout** demonstration program, which [page 12-69](#) describes, uses the **sqlbreakcallback()** function to establish a time-out interval for a database query.

Terminating a Connection

An 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 on the CLOSE DATABASE and DISCONNECT statements, see of the [Informix Guide to SQL: Syntax](#). For information on the **sqldetach()** and **sqlexit()** library functions, see [page 12-56](#) and [page 12-64](#), respectively.

Optimized Message Transfers

ESQL/C provides a feature called *optimized message transfers*, which allow you to minimize message transfers with the database server for most ESQL/C statements. 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, ESQL/C *expects* that SQL statements will succeed. Consequently, ESQL/C chains, and in some cases eliminates, confirmation messages from the database server.

Restrictions on Optimized Message Transfers

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 ESQL/C reaches one of the preceding statements, it forces the message out to the database server. ESQL/C then continues message chaining for subsequent SQL statements. Only SQL statements that require network traffic cause ESQL/C to flush the message queue.

SQL statements that do not require network traffic, such as the DECLARE statement, do not cause 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 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 **OPTMSG** 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 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 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, 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 12-41](#) for more information on 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. ESQL/C delays sending the message for the DELETE statement. The example disables message chaining after the DELETE statement so that 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, 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 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, 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 sections describe the ESQL/C library functions that you can use to control the database server sessions.

Function Name	Description
ifx_getcur_conn_name()	Returns the name of the current connection.
sqgetdbs()	Returns the names of databases that a database server can access.
sqlbreak()	Sends the database server a request to stop processing.
sqlbreakcallback()	Establishes a time-out interval and a callback function for interrupting an SQL request.
sqldetach()	Detaches a child process from a database server connection.
sqldone()	Determines whether the database server is currently processing an SQL request.
sqlexit()	Terminates a database server connection.
sqlsignal()	Performs signal handling and cleanup of child processes.
sqlstart()	Starts a database server connection.

ifx_getcur_conn_name()

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

sqgetdbs()

The **sqgetdbs()** function returns the names of databases that a database server can access.

Syntax

```
int sqgetdbs(ret_fcnt, dbnarray, dbnsize, dbnbuffer, dbnbufsz)
    int *ret_fcnt;
    char **dbnarray;
    int dbnsize;
    char *dbnbuffer;
    int dbnbufsz;
```

ret_fcnt is a pointer to the number of database names that the function returns.

dbnarray is a user-defined array of character pointers.

dbnsize is the size of the *dbnarray* user-defined array.

dbnbuffer is a pointer to a user-defined buffer that contains the names of the databases that the function returns.

dbnbufsz is 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' */
    int num_returned;             /* number of database names returned */
    int 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,”](#) of this manual.

Example 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.

```

sqlbreak()

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

```
int 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.

SELECT	OPEN	ALTER INDEX
UPDATE	CREATE TABLE	EXECUTE PROCEDURE
DELETE	CREATE INDEX	
INSERT	ALTER TABLE	

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 **sqllda** 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 [page 12-34](#).
- Specify a time-out interval with the **sqlbreakcallback()** function.
After the time-out 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 [page 12-36](#).

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() .



sqlbreakcallback()

The **sqlbreakcallback()** function allows you to specify a time-out 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.

Warning: Do not use the **sqlbreakcallback()** function if your ESQL/C application uses shared memory (**olipcshm**) as the **nettype** to connect to an Universal Server 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

```
int sqlbreakcallback(timeout, callbackfunc_ptr);
    long timeout;
    void (* callbackfunc_ptr)(int status);
```

timeout is 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 time-out value.
- 0 immediately calls the function that *callbackfunc_ptr* indicates.
- >0 sets the time-out 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 could be a variable with an **int**, **long**, or **short** data type.

callbackfunc_ptr is 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 <i>status</i> Argument
When the database server begins processing an SQL request	<i>status</i> = 1
While the database server executes an SQL request, when the time-out interval has elapsed	<i>status</i> = 2
When the database server completes the processing of an SQL request	<i>status</i> = 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 ESQL/C control functions: **sqldone()**, **sqlbreak()**, and **sqlbreakcallback()**. For more information about the callback function, see [page 12-37](#).

If you call **sqlbreakcallback()** with a time-out value of zero (0), 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 time-out value to a nonzero value, as follows:

```
sqlbreakcallback(timeout, callbackfunc_ptr);
```

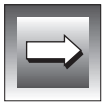
Important: *Small time-out values might adversely affect the performance of your application.*

For more information about the time-out interval, see [page 12-36](#).

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.



sqldetach()

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

```
int 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;
        int 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()
    {
        int rc, status, pid;

        EXEC SQL BEGIN DECLARE SECTION;
        int 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;
{
    int 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);
        }
    }
}
```

Example 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.
```

sqldone()

The **sqldone()** function determines whether the database server is currently processing an SQL request.

Syntax

```
int 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 <i>not</i> currently processing an SQL request: it is idle.
-439	The database server is currently processing an SQL request.

sqlexit()

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

```
int 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 re-establishes 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.

sqlsignal()

The **sqlsignal()** function enables, disables, or reenables signal handling of the signals that the ESQL/C library handles.

Syntax

```
void sqlsignal(sigvalue, sigfunc_ptr, mode)
               int sigvalue;
               void (*sigfunc_ptr) (void);
               short mode;
```

sigvalue is the integer 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 is 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 is 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 **tlipcp**).

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 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 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 INFORMIX-ESQL library to handle signals:

```
sqlsignal(-1, (void (*)()), 2);
```

When you initialize SIGCHLD signal handling with **sqlsignal()**, you allow the ESQL/C library to process SIGCHLD cleanup. Otherwise, your application must perform the cleanup for these processes if defunct child processes are a problem.



sqlstart()

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. ESQL/C continues to support this function for backward compatibility with these applications. For applications of Version 6.0 and later, use the **CONNECT** statement to establish explicit connections to a default database server.

Syntax

```
int sqlstart();
```

Usage

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 has been 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 on disconnecting, see [“Terminating a Connection” on page 12-40](#). For more information on explicit and implicit connections, see [“Establishing a Connection” on page 12-15](#).

Return Codes

- 0 The call to **sqlstart()** was successful.
- <0 The call to **sqlstart()** was *not* successful.

The timeout Program

The **timeout** program demonstrates how to set up a time-out interval. This program uses the **sqlbreakcallback()** function to perform the following actions:

- To specify a time-out 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 time-out interval elapses

If execution of an SQL request exceeds the time-out 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 1-47](#) for more information on 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
11 #define LCASE(c) (isupper(c) ? tolower(c) : (c))

12 /* Defines for callback mechanism */
13 #define DB_TIMEOUT      200      /* number of milliseconds in time-out */
14 #define SQL_INTERRUPT -213      /* SQLCODE value for interrupted stmt */

15 /* These constants are used for the canceltst table, created by
16  * this program.
17  */
18 #define MAX_ROWS      10000      /* number of rows added to table */
19 EXEC SQL define CHARFLDSIZE  20; /* size of character columns in table */

20 /* Define for sqldone() return values */
21 #define SERVER_BUSY  -439

22 /* These constants used by the exp_chk2() function to determine
23  * whether to display warnings.
24  */
25 #define WARNNOTIFY      1
26 #define NOWARNNOTIFY    0

27 long dspquery();
28 extern long exp_chk2();
29 void on_timeout();

30 main()
31 {
32     char ques[80], prompt_ans();
33     long ret;
34     int create_tbl(), drop_tbl();

35     printf("TIMEOUT Sample ESQL Program running.\n\n");
36     /*
37      * Establish an explicit connection to the stores7 database
38      * on the default database server.
39      */
40     EXEC SQL connect to 'stores7';
```

Continued on page 12-72

Lines 4 to 9

Lines 4 to 8 include the UNIX header files from the **/usr/include** directory. The ESQ/C **sqltypes.h** header file (line 9) defines names for integer values that identify SQL and C data types.

Lines 11 to 21

Line 11 defines **LCASE**, a macro that converts an uppercase character to a lowercase character. The **DB_TIMEOUT** (line 13) constant defines the number of milliseconds in the time-out interval. The **SQL_INTERRUPT** constant (line 14) defines the **SQLCODE** value that the database server returns when it interrupts an SQL statement.

Lines 18 and 19 define constants that the **create_tbl()** function uses to create the **cancelst** table. This table holds the test data needed for the large query (lines 126 to 133). **MAX_ROWS** is the number of rows that **create_tbl()** inserts into **cancelst**. 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 **cancelst**.

Line 21 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 25 and 26

The **exp_chk2()** exception-handling function uses the **WARNNOTIFY** and **NOWARNNOTIFY** constants (lines 25 and 26). 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 on the **exp_chk2()** function, see [“Lines 349 to 356” on page 12-87](#).

Lines 30 to 34

The **main()** program block begins on line 30. Lines 32 to 34 declare variables local to the **main()** program block.

```
41 if (exp_chk2("CONNECT to stores7", NOWARNNOTIFY) < 0)
42     exit(1);
43 printf("Connected to 'stores7' on default server\n");
44 /*
45  * Create the canceltst table to hold MAX_ROWS (10,000) rows.
46  */
47 if (!create_tbl())
48     {
49         printf("\nTIMEOUT Sample Program over.\n\n");
50         exit(1);
51     }
52 while(1)
53     {
54         /*
55          * Establish on_timeout() as callback function. The callback
56          * function is called with an argument value of 2 when the
57          * database server has executed a single SQL request for number
58          * of milliseconds specified by the DB_TIMEOUT constant
59          * (0.00333333 minutes by default). Call to sqlbreakcallback()
60          * must come after server connection is established and before
61          * the first SQL statement that can be interrupted.
62          */
63         if (sqlbreakcallback(DB_TIMEOUT, on_timeout))
64             {
65                 printf("\nUnable to establish callback function.\n");
66                 printf("TIMEOUT Sample Program over.\n\n");
67                 exit(1);
68             }
69         /*
70          * Notify end user of time-out interval.
71          */
72         printf("Time-out interval for SQL requests is: ");
73         printf("%.8f minutes\n", DB_TIMEOUT/60000.00);
74         stcopy("Are you ready to begin execution of the query?",
75             ques);
76         if (prompt_ans(ques) == 'n')
77             {
78                 /*
79                  * Unregister callback function so table cleanup will not
80                  * be interrupted.
81                  */
82                 sqlbreakcallback(-1L, (void *)NULL);
83                 break;
84             }
```

Continued on page 12-74

Lines 44 to 51

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 50).

Line 52

This **while** loop (which ends on line 98), 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 54 to 68

The first task of the **while** loop is to use **sqlbreakcallback()** to specify a time-out 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 time-out intervals, you can change the DB_TIMEOUT constant value and recompile the **timeout.ec** source file.

Lines 69 to 73

These **printf()** functions notify the user of the time-out 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 74 to 84

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 314 to 323) from initiating the callback function. For a description of the **prompt_ans()** function, see [“Lines 338 to 348” on page 12-87](#).

```
85      /*
86      * Start display of query output
87      */
88      printf("\nBeginning execution of query...\n\n");
89      if ((ret = dspquery()) == 0)
90      {
91          if (prompt_ans("Try another run?") == 'y')
92              continue;
93          else
94              break;
95      }
96      else /* dspquery() encountered an error */
97          exit(1);
98      } /* end while */
99  /*
100  * Drop the table created for this program
101  */
102  drop_tbl();
103  EXEC SQL disconnect current;
104  if (exp_chk2("DISCONNECT for stores7", WARNNOTIFY) != 0)
105      exit(1);
106  printf("\nDisconnected stores7 connection\n");
107  printf("\nTIMEOUT Sample Program over.\n\n");
108}
109/* This function performs the query on the canceltst table. */
110long dspquery()
111{
112  int cnt = 0;
113  long ret = 0;
114  long sqlcode = 0;
115  int sqlerr_code, sqlstate_err();
116  void disp_exception(), disp_error(), disp_warning();
117  EXEC SQL BEGIN DECLARE SECTION;
118      char fld1_val[ CHARFLDSIZE + 1 ];
119      char fld2_val[ CHARFLDSIZE + 1 ];
120      long int_val;
121  EXEC SQL END DECLARE SECTION;
122  /* This query contains an artificially complex WHERE clause to
123  * keep the database server busy long enough for an interrupt
124  * to occur.
125  */
126  EXEC SQL declare cancel_curs cursor for
127      select sum(int_fld), char_fld1, char_fld2
128      from canceltst
129      where char_fld1 matches "*f*"
130          or char_fld1 matches "*h*"
131          or char_fld2 matches "*W*"
132          or char_fld2 matches "*l*"
133      group by char_fld1, char_fld2;
```

Continued on page 12-76

Lines 85 to 98

If the user chooses to continue the query, the program calls the **dspquery()** function (line 89) to run the **canceltst** query. The **prompt_ans()** function displays a prompt so the user can decide whether to run the program again.

Lines 99 to 102

The **drop_tbl()** function drops the **canceltst** table from the **stores7** database to clean up after the program.

Lines 109 to 121

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 122 to 133

Line 126 declares the **cancel_curs** cursor for the query. The actual **SELECT** (lines 127 to 133) 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_fld2 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.

```
134 EXEC SQL open cancel_curs;

135 sqlcode = SQLCODE;
136 sqlerr_code = sqlstate_err(); /* check SQLSTATE for exception */
137 if (sqlerr_code != 0)        /* if exception found */
138 {
139     if (sqlerr_code == -1)    /* runtime error encountered */
140     {
141         if (sqlcode == SQL_INTERRUPT) /* user interrupt */
142         {
143             /* This is where you would clean up resources */
144             printf("\n      TIMEOUT INTERRUPT PROCESSED\n\n");
145             sqlcode = 0;
146         }
147     } else                    /* serious runtime error */
148         disp_error("OPEN cancel_curs");

149     EXEC SQL close cancel_curs;
150     EXEC SQL free cancel_curs;
151     return(sqlcode);
152 }
153 else if (sqlerr_code == 1)    /* warning encountered */
154     disp_warning("OPEN cancel_curs");
155 }
```

Continued on page 12-78

Line 134

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 135 to 155

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 141, the program uses the SQL_INTERRUPT defined constant (which line 14 defines), for this SQLCODE value.

The **sqlstate_err()** function (line 136) 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 135 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 151).

The first **if** statement (line 137) checks if the OPEN encounters any type of exception (**sqlstate_err()** returns a nonzero value). The second **if** (line 139) 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 ESQL/C does *not* handle an interrupted SQL statement as a runtime error, the third **if** checks explicitly for the SQL_INTERRUPT value (line 141). If the OPEN was interrupted, line 144 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 147 and 148 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 149 to 151 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 159) if the OPEN has been interrupted.

If **sqlstate_err()** returns one (1), the OPEN has generated a warning. Lines 153 and 154 call the **disp_warning()** function to display warning information from the diagnostics area. For more information on the **disp_error()** and **disp_warning()** functions, see Lines 342 to 349 on [page 12-87](#).

```
156 printf("Displaying data...\n");
157 while(1)
158 {
159     EXEC SQL fetch cancel_curs into :int_val, :fld1_val, :fld2_val;
160     if ((ret = exp_chk2("FETCH from cancel_curs", NOWARNNOTIFY)) == 0)
161     {
162         printf("    sum(int_fld) = %ld\n", int_val);
163         printf("    char_fld1 = %s\n", fld1_val);
164         printf("    char_fld2 = %s\n\n", fld2_val);
165     }

166     /*
167     * Will display warning messages (WARNNOTIFY) but continue
168     * execution when they occur (exp_chk2() == 1)
169     */
170     else
171     {
172         if (ret==100)                /* NOT FOUND condition */
173         {
174             printf("\nNumber of rows found: %d\n\n", cnt);
175             break;
176         }
177         if (ret < 0)                /* Runtime error */
178         {
179             EXEC SQL close cancel_curs;
180             EXEC SQL free cancel_curs;
181             return(ret);
182         }
183     }

184     cnt++;

185 } /* end while */

186 EXEC SQL close cancel_curs;
187 EXEC SQL free cancel_curs;
188 return(0);
189}

190/*
191 * The on_timeout() function is the callback function. If the user
192 * confirms the cancellation, this function uses sqlbreak() to
193 * send an interrupt request to the database server.
194 */
195void on_timeout(when_called)
196int when_called;
197{
198int ret;
199static intr_sent;
```

Continued on page 12-80

Lines 156 to 183

This **while** loop executes for each row that the **cancel_curs** cursor contains. The **FETCH** statement (line 159) 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 177 to 182). 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 174).

Lines 186 to 188

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 63 registers this callback function and establishes a time-out 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 time-out interval elapses.

```
200 /* Determine when callback function has been called. */
201 switch(when_called)
202 {
203     case 0: /* Request to server completed */
204         printf("+-----SQL Request ends");
205         printf("-----+\n\n");
206
207         /*
208          * Unregister callback function so no further SQL statements
209          * can be interrupted.
210          */
211         if (intr_sent)
212             sqlbreakcallback(-1L, (void *)NULL);
213         break;
214
215     case 1: /* Request to server begins */
216         printf("+-----SQL Request begins");
217         printf("-----+\n");
218         printf("|                               ");
219         printf("|\n");
220         intr_sent = 0;
221         break;
222
223     case 2: /* Time-out interval has expired */
224         /*
225          * Is the database server still processing the request?
226          */
227         if (sqldone() == SERVER_BUSY)
228             if (!intr_sent) /* has interrupt already been sent? */
229             {
230                 printf("|    An interrupt has been received ");
231                 printf("by the application.|\n");
232                 printf("|                               ");
233                 printf("|\n");
234
235                 /*
236                  * Ask user to confirm interrupt
237                  */
238                 if (cancel_request())
239                 {
240                     printf("|          TIMEOUT INTERRUPT ");
241                     printf("REQUESTED          |\n");
242
243                     /*
244                      * Call sqlbreak() to issue an interrupt request for
245                      * current SQL request to be cancelled.
246                      */
247                     sqlbreak();
248
249                     }
250                 intr_sent = 1;
251             }
252         break;
```

Continued on page 12-82

Lines 200 to 250

This **switch** statement uses the callback function argument, **when_called**, to determine the actions of the callback function, as follows:

- Lines 203 to 212: If **when_called** is 0, the callback function has been 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 213 to 219: If **when_called** is 1, the callback function has been 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 on these message-request boxes, see [“Lines 26 to 37” on page 12-89](#). The function also initializes the **intr_sent** flag to 0 because the user has not yet sent an interrupt for this SQL request.

- Lines 220 to 246: If **when_called** is 2, the callback function has been called because the time-out interval has elapsed.

To handle the elapsed time-out interval, the callback function first calls the ESQ/C **sqldone()** function (line 224) 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 225 checks if the user has already attempted to interrupt the SQL request that is currently executing. If an interrupt has been 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 time-out interval has elapsed (lines 227 to 230). It then uses the **cancel_request()** function (line 234) to allow the user to confirm the interrupt. For more information on **cancel_request()**, see [“Lines 252 to 262” on page 12-83](#).

```
247     default:
248         printf("Invalid status value in callback: %d\n", when_called);
249         break;
250     }
251 }

252 /* This function prompts the user to confirm the sending of an
253  * interrupt request for the current SQL request.
254  */
255 int cancel_request()
256 {
257     char prompt_ans();

258     if (prompt_ans("Do you want to confirm this interrupt?") == 'n')
259         return(0);    /* don't interrupt SQL request */
260     else
261         return(1);    /* interrupt SQL request */
262 }

263 /* This function creates a new table in the current database. It
264  * populates this table with MAX_ROWS rows of data. */
265 int create_tbl()
266 {
267     char st_msg[15];
268     int ret = 1;

269     EXEC SQL BEGIN DECLARE SECTION;
270     int cnt;
271     int pa;
272     int i;
273     char fld1[ CHARFLDSIZE + 1 ], fld2[ CHARFLDSIZE + 1 ];
274     EXEC SQL END DECLARE SECTION;

275     /*
276     * Create canceltst table in current database
277     */
278     EXEC SQL create table canceltst (char_fld1 char(20),
279     char_fld2 char(20), int_fld integer);
280     if (exp_chk2("CREATE TABLE", WARNNOTIFY) < 0)
281         return(0);
282     printf("Created table 'canceltst'\n");

283     /*
284     * Insert MAX_ROWS of data into canceltst
285     */
286     printf("Inserting rows into 'canceltst'...\n");
287     for (i = 0; i < MAX_ROWS; i++)
288     {
```

Continued on page 12-84

Lines 200 to 250 (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 244 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 248).

Lines 252 to 262

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 263 to 282

The **create_tbl()** function creates the **canceltst** table and inserts the test data into this table. The CREATE TABLE statement (lines 278 and 279) 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 280) displays the diagnostics-area information and **create_tbl()** returns zero to indicate that an error has occurred.

Lines 283 to 288

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 127 to 133) because it executes too quickly, increase the value of MAX_ROWS and recompile the **timeout.ec** file.

```
289     if (i%2 == 1) /* odd-numbered rows */
290     {
291         stcopy("4100 Bohannon Dr", fld1);
292         stcopy("Menlo Park, CA", fld2);
293     }
294     else /* even-numbered rows */
295     {
296         stcopy("Informix", fld1);
297         stcopy("Software", fld2);
298     }
299     EXEC SQL insert into canceltst
300     values (:fld1, :fld2, :i);
301     if ( (i+1)%1000 == 0 ) /* every 1000 rows */
302     printf("    Inserted %d rows\n", i+1);
303     sprintf(st_msg, "INSERT #%d", i);
304     if (exp_chk2(st_msg, WARNNOTIFY) < 0)
305     {
306         ret = 0;
307         break;
308     }
309 }
310 printf("Inserted %ld rows into 'canceltst'.\n", MAX_ROWS);

311/*
312 * Verify that MAX_ROWS rows have added to canceltst
313 */
314 printf("Counting number of rows in 'canceltst' table...\n");
315 EXEC SQL select count(*) into :cnt from canceltst;
316 if (exp_chk2("SELECT count(*)", WARNNOTIFY) < 0)
317     return(0);
318 printf("Number of rows = %ld\n\n", cnt);

319 return (ret);
320}

321/* This function drops the 'canceltst' table */
322int drop_tbl()
323{
324 printf("\nCleaning up...\n");

325 EXEC SQL drop table canceltst;
326 if (exp_chk2("DROP TABLE", WARNNOTIFY) < 0)
327     return(0);
328 printf("Dropped table 'canceltst'\n");

329 return(1);
330}
```

Continued on page 12-86

Lines 289 to 293

This **if** statement generates the values for the **char_fld1** and **char_fld2** columns of the **canceltst** table. Lines 291 and 292 execute for odd-numbered rows. They store the strings “4100 Bohannon Dr” and “Menlo Park, CA” in the **fld1** and **fld2** variables, respectively.

Lines 294 to 298

Lines 296 and 297 execute for even-numbered rows. They store the strings Informix and Software in the **fld1** and **fld2** variables, respectively.

Lines 299 to 308

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, respectively. The function notifies the user after it inserts every 1000 rows (lines 301 and 302). If the INSERT encounters an error, the **exp_chk2()** function (line 304) displays the diagnostics-area information and **create_tbl()** returns zero to indicate that an error has occurred.

Lines 311 to 318

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 310 displays. If the SELECT encounters an error, the **exp_chk2()** function (line 316) displays the diagnostics-area information, and **create_tbl()** returns 0 to indicate that an error has occurred.

Lines 321 to 330

The **drop_tbl()** function drops the **canceltst** table from the current database. If the DROP TABLE statement (line 325) encounters an error, the **exp_chk2()** function displays the diagnostics-area information and **drop_tbl()** returns 0 to indicate that an error has occurred.

```
331/*
332 * The inpfuns.c file contains the following functions used in this
333 * program:
334 *     getans(ans, len) - accepts user input, up to 'len' number of
335 *                       characters and puts it in 'ans'
336 */
337#include "inpfuns.c"

338char prompt_ans(question)
339char * question;
340{
341    char ans = ' ';

342    while(ans != 'y' && ans != 'n')
343    {
344        printf("\n*** %s (y/n): ", question);
345        getans(&ans,1);
346    }
347    return ans;
348}

349/*
350 * The exp_chk() file contains the exception handling functions to
351 * check the SQLSTATE status variable to see if an error has occurred
352 * following an SQL statement. If a warning or an error has
353 * occurred, exp_chk2() executes the GET DIAGNOSTICS statement and
354 * displays the detail for each exception that is returned.
355 */
356EXEC SQL include exp_chk.ec;
```

Lines 331 to 337

Several of the 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 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 inpfuns.c File” on page 8-68](#).

Lines 338 to 348

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 349 to 356

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 ESQL/C **include** directive because the exception-handling functions use ESQL/C statements. For a description of the **exp_chk.ec** file, see [“Guide to the exp_chk.ec File” on page 11-59](#).



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 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
3 Connected to 'stores7' on default server
4 Created table 'canceltst'
5 Inserting rows into 'canceltst'...
6   Inserted 1000 rows
7   Inserted 2000 rows
8   Inserted 3000 rows
9   Inserted 4000 rows
10  Inserted 5000 rows
11  Inserted 6000 rows
12  Inserted 7000 rows
13  Inserted 8000 rows
14  Inserted 9000 rows
15  Inserted 10000 rows
16 Inserted 10000 rows into 'canceltst'.
17 Counting number of rows in 'canceltst' table...
18 Number of rows = 10000
19
20 Time-out interval for SQL requests is: 0.00333333 minutes
21
22 *** Are you ready to begin execution of the query? (y/n): y
23
24 Beginning execution of query...
25
26 +-----SQL Request begins-----+
27 |                                     |
28 +-----SQL Request ends-----+
29
30 +-----SQL Request begins-----+
31 |                                     |
32 |   An interrupt has been received by the application. |
33 |                                     |
34
35 *** Do you want to confirm this interrupt? (y/n): y
36 |   TIMEOUT INTERRUPT REQUESTED   |
37 +-----SQL Request ends-----+

```

Continued on page 12-90

Lines 4 to 18

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 263 to 282” on page 12-83](#).

Lines 20 to 22

Line 20 displays the time-out interval to indicate that **sqlbreakcallback()** has successfully registered the callback function and established the time-out interval of 200 milliseconds (0.00333333 minutes). Line 22 asks the user to indicate the beginning of the query execution. This prompt prepares the user for the confirmation prompt (lines 35 and 59), which must be answered quickly to send an interrupt while the database server is still executing the query.

Line 24

This line indicates the beginning of the **dspquery()** function, the point at which the database server begins the **canceltst** query.

Lines 26 to 37

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 200 to 250” on page 12-81](#).) 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 on message requests, see [“Interruptible SQL Statements” on page 12-33](#).

The first message-request box (lines 26 to 28) indicates that the first message request completes before the time-out interval elapses. The second message-request box (lines 30 to 37) indicates that execution of this message request exceeds the time-out interval and calls the callback function with a status value of 2. The callback function prompts the user to confirm the interrupt request (line 35).

Line 36 indicates that the **sqlbreak()** function has requested an interrupt. The message request then completes (line 37).

```

38
39
40      TIMEOUT INTERRUPT PROCESSED
41
42
43 *** Try another run? (y/n): y
44 Time-out interval for SQL requests is: 0.00333333 minutes
45
46 *** Are you ready to begin execution of the query? (y/n): y
47
48 Beginning execution of query...
49
50 +-----SQL Request begins-----+
51 |                                     |
52 +-----SQL Request ends-----+
53
54 +-----SQL Request begins-----+
55 |                                     |
56 |   An interrupt has been received by the application. |
57 |                                     |
58
59 *** Do you want to confirm this interrupt? (y/n): n
60 +-----SQL Request ends-----+
61
62 Displaying data...
63   sum(int_fld) = 25000000
64   char_fld1 = 4100 Bohannon Dr
65   char_fld2 = Menlo Park, CA
66
67   sum(int_fld) = 24995000
68   char_fld1 = Informix
69   char_fld2 = Software
70
71
72 Number of rows found: 2
73
74
75 *** Try another run? (y/n): n
76

```

```

77 Cleaning up...
78 Dropped table 'canceltst'
79
80 Disconnected stores7 connection
81
82 TIMEOUT Sample Program over.
83

```

Line 40

When the database server actually processes the interrupt request, it sets SQLCODE to -213. Line 40 indicates that the application program has responded to this status.

Line 43

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 50 to 56

The message-request box indicates that the first message request completes before the time-out interval elapses. The second message-request box (lines 54 to 60) indicates that execution of this message request again exceeds the time-out interval and calls the callback function (with **when_called** = 2). The callback function prompts the user to confirm the interrupt request (line 59). This time the user answers *n*.

Lines 62 to 72

Because the user has not interrupted the **canceltst** query, the program displays the row information that the query returns.

Lines 77 and 78

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 321 to 330” on page 12-85](#).

Using ESQL/C Connection Library Functions in a Windows Environment

To establish an *explicit* connection (sometimes called a *direct connection*), ESQL/C supports the SQL connection statements. For a complete description of the SQL connection statements, see the [Informix Guide to SQL: Syntax](#). ESQL/C also supports the connection library functions that Figure 12-11 lists for establishing an explicit connection from a Windows environment.

Figure 12-11
ESQL/C Connection Library Functions and Their SQL Equivalents

ESQL/C for Windows Library Function	Description	SQL Equivalent
GetConnect()	Requests an explicit connection and returns a pointer to the connection information	CONNECT TO '@DBSERVERNAME' WITH CONCURRENT TRANSACTION
SetConnect()	Switches the connection to an established (dormant) explicit connection	SET CONNECT TO (without the DEFAULT option)
ReleaseConnect()	Closes an established explicit connection	DISCONNECT (without the DEFAULT, CURRENT, or ALL options)



Important: ESQL/C supports the connection library functions for backward compatibility with Version 5.01 ESQL/C for Windows applications. When you write new ESQL/C applications for Windows environments, use the SQL connection statements (CONNECT, DISCONNECT, and SET CONNECTION) instead of the ESQL/C connection library functions.

ESQL/C uses an internal structure that contains the handle for the connection and other connection information. The 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:

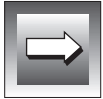
- If you open a cursor in one module (such as a shared DLL), and then use an explicit connection to use that cursor in another module, you must use a host variable for the name of the cursor when you declare the cursor.
- Make sure that your application uses the correct connection handle at all times.



Warning: *If an application uses the wrong connection handle, the application could modify the wrong database without the knowledge of the user.*

When you compile your ESQL/C program, the **esql** command processor automatically links the ESQL/C connection functions to your program. The following pages describe each of these connection functions.♦

WIN NT/95



GetConnect()

The **GetConnect()** function is available only in the Windows environment and establishes a new explicit connection to a database server.

Important: *ESQL/C supports the **GetConnect()** connection library function for backward compatibility with Version 5.01 ESQL/C for Windows applications. When you write new 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 on the default and specified database server, see [“Sources of Connection Information in a Windows Environment” on page 12-7.](#)

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, 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 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 <i>must</i> use DATABASE (or some other valid SQL statement) to open the database.	Can establish an explicit connection to a database server <i>and</i> 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 *ESQL/C* application does not need to ensure that the current transaction has been 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. ♦

WIN NT/95



ReleaseConnect()

The **ReleaseConnect()** function is available only in the Windows environment. It releases, or terminates, the explicit connection and clears all allocated memory.

Important: *ESQL/C supports the **ReleaseConnect()** connection library function for backward compatibility with Version 5.01 ESQL/C for Windows applications. When you write new ESQL/C applications for Windows environments, use the SQL DISCONNECT statement to terminate an established explicit connection.*

Syntax

```
void *ReleaseConnect ( void *CnctHndl )
```

CnctHndl is 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 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. ♦

WIN NT/95



SetConnect()

The **SetConnect()** function is available only in the Windows environment. It switches the connection to a specified explicit connection.

Important: *ESQL/C supports the **SetConnect()** connection library function for backward compatibility with Version 5.01 ESQL/C for Windows applications. When you write new ESQL/C applications for Windows environments, use the SQL SET CONNECTION statement to switch to another active connection.*

Syntax

```
void *SetConnect ( void *CnctHndl )
```

CnctHndl is 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 exists 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 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 ESQL/C application does not need to ensure that the current transaction has been committed or rolled back before it calls the **SetConnect()** function to switch to another explicit connection.

Return Codes

<i>CnctHndl</i>	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 <i>not</i> successful, indicating that no explicit connection has been established. ♦

Using Informix Libraries

Choosing a Version of the Informix General Libraries	13-5
The Informix General Libraries	13-5
The esql Command	13-6
Linking Static Informix General Libraries.	13-7
Linking Static Informix General Libraries into an ESQL/C Module	13-7
Linking Shared Informix General Libraries	13-8
Symbolic Names of Linked Shared Libraries	13-8
Choosing Between Shared and Static Versions	13-10
Compatibility of Pre-existing ESQL/C Applications with Current Library Versions	13-12
Using the ifx_getversion Utility	13-13
Checking the API Version of a Library	13-14
Creating Thread-Safe ESQL/C Applications	13-16
Characteristics of Thread-Safe ESQL/C Code	13-17
Programming a Thread-Safe ESQL/C Application	13-17
Concurrent Active Connections.	13-18
Connections Across Threads	13-20
The DISCONNECT ALL Statement	13-22
Prepared Statements Across Threads	13-22
Cursors Across Threads	13-24
Environment Variables Across Threads	13-25
Decimal Functions	13-25
DCE Restrictions	13-25

Linking Thread-Safe Libraries	13-26
Linking Thread-Safe Informix General Libraries to an ESQL/C Module on a UNIX Operating System	13-26
Defining Thread-Safe Variables	13-27
Linking Shared or Static Versions	13-28
Linking Thread-Safe Informix General Libraries to an ESQL/C Module in a Windows Environment	13-29
Using ESQL/C Thread-Safe Decimal Functions	13-29
ifx_dececv(), ifx_deccv()	13-31
A Sample Thread-Safe Program	13-33
Source Listing	13-33
Sample Output	13-39
Creating a Dynamic Thread Library on UNIX Operating Systems	13-40
Data Types	13-43
Registering the Dynamic Thread Functions	13-47
Setting the \$THREADLIB Environment Variable	13-50
Creating the Shared Library	13-50
Compiling with the -thread and -l Preprocessor Options.	13-51

Informix products use the *Informix general libraries* for interactions between the client SQL application programming interface (API) products (INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL) and the database server. You can choose between the following types of Informix general libraries to link with your ESQL/C application:

- **Static Informix general libraries**
To link a static library, the linker copies the library functions to the executable of your ESQL/C program. The static Informix general libraries allow an 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 of your 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 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.

Static versions of Informix general libraries are available only with ESQL/C on UNIX platforms. The Figure 13-1 shows the available options.

Figure 13-1
Different Version of the ESQL/C General Library Available for UNIX

Thread-Safe Options		
Linking Options	Thread-safe	Default
Static	Static, thread-safe general libraries	Static, default general libraries
Shared	Shared, thread-safe general libraries	Shared, default general libraries.



This chapter describes how to link the static, shared, and thread-safe Informix general libraries with your ESQL/C application.

For Windows environments, ESQL/C provides only one version of the Informix general libraries. These libraries are implemented as thread-safe dynamic link libraries (DLL)s. When you compile an ESQL/C program, the **esql** command processor automatically links these DLLs into your ESQL/C application.



Tip: A Windows environment DLL is a shared library. Windows environments do not support static libraries. Therefore, in Windows environments the **esql** command processor does not support the **-static** command-line option of **esql**. The option is available in the Version 9.1 and later UNIX versions of ESQL/C. ◆

UNIX

Choosing a Version of the Informix General Libraries

This section provides information on the following topics:

- What are the Informix general libraries?
- What command-line options of the **esql** command determine the version of the Informix general libraries to link with your ESQL/C program?
- How do you link the static Informix general libraries that are available on UNIX platforms with your ESQL/C program? ♦
- How do you link the shared Informix general libraries with your ESQL/C program?
- What are some factors that you need to consider to determine which type of Informix general libraries to use?

UNIX

The Informix General Libraries

The following table lists the Informix general libraries for ESQL/C on a UNIX platform.

Informix General Library	Purpose
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 Informix products
libasf	Contains functions that handle communication protocols between client application and database server

Windows

Informix general libraries reside in the **\$INFORMIXDIR/lib/esql** and **\$INFORMIXDIR/lib** directories on UNIX operating systems. ♦

The Informix general library for ESQL/C for Windows is just one DLL named **isqlt09a.dll**. It resides in the **%INFORMIXDIR%\lib** directories in Windows environments. ♦

The esql Command

To determine which type of Informix general libraries to link with your ESQL/C application, the **esql** command supports the command-line options in Figure 13-2.

Figure 13-2
esql Command-Line Options for Informix General Libraries

Version of Informix Libraries to Link	esql Command-Line Option	Reference and Comments
Shared libraries	<i>No option (default)</i>	“Linking Shared Informix General Libraries” on page 13-8
Static libraries	-static	“Linking Static Informix General Libraries” on page 13-7 . This option available for UNIX operating systems only.
Thread-safe shared libraries	-thread	“Linking Thread-Safe Informix General Libraries to an ESQL/C Module on a UNIX Operating System” on page 13-26 and “Linking Thread-Safe Informix General Libraries to an ESQL/C Module in a Windows Environment” on page 13-29
Thread-safe static libraries	-thread -static	“Creating a Dynamic Thread Library on UNIX Operating Systems” on page 13-40

For the complete syntax of the **esql** command, see [“Using the esql Command” on page 1-47](#).

UNIX

Linking Static Informix General Libraries

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 **libxxx.a**.
- A thread-safe static Informix general library has a name of the form **libthxxx.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.a** 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 ESQL/C application. At runtime, your ESQL/C program can access these Informix general-library functions directly from its executable file.

Linking Static Informix General Libraries into an ESQL/C Module

To link static Informix general libraries with an 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 ESQL/C application. For more information, see [“Linking Thread-Safe Libraries” on page 13-26](#).

Tip: The **esql** command for pre-Version 7.2 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 ESQL/C application. ♦



Linking Shared Informix General Libraries

ESQL/C can dynamically link a shared library, which places this library in shared memory. Once the shared library is in shared memory, other 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 ESQL/C application, your operating system must support shared libraries. Operating systems that support shared libraries include Sun and HP versions of UNIX as well as Windows NT and Windows 95. You should be familiar with the creation of shared libraries and with the compile options that your C compiler requires to build them.



UNIX

Symbolic Names of Linked Shared Libraries

When the **esql** command links shared or thread-safe shared Informix libraries with your 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 **libxxx.yyy**.
- A thread-safe shared Informix general library has a symbolic name of the form **libthxxx.yyy**.

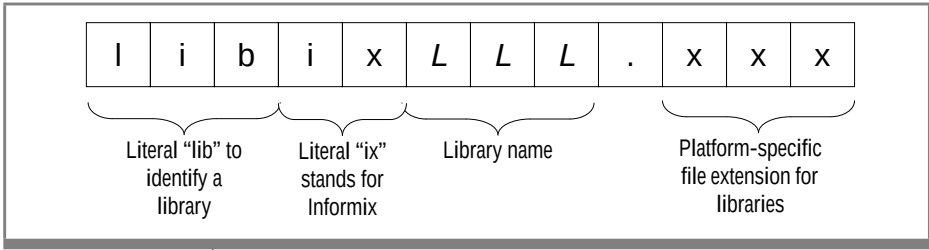
In these static-library names, xxx identifies the particular library and yyy is a platform-specific file extension that identifies shared library files.

Tip: To refer to a specific shared-library file, this manual 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, refer to your UNIX operating system documentation.

When you install the ESQL/C product, the installation script makes a symbolic link of the *actual* shared library name to the file with the symbolic name. [Figure 13-3 on page 13-9](#) shows the format for the actual names of shared and thread-safe shared versions of Informix libraries.



Figure 13-3
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 ESQL/C application. At runtime, 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 into 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 ESQL/C module, you do not need to specify a command-line option. 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 ESQL/C application. For more information, see [“Linking Thread-Safe Libraries” on page 13-26](#).

Choosing Between Shared and Static Versions

Pre-Version 7.2 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 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 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 ESQL/C executable are transparent to the end user.

Although shared libraries save both disk and memory space, when an 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 CPU 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 ESQL/C application. For more information, see [“Creating a Dynamic Thread Library on UNIX Operating Systems” on page 13-40](#). ♦

Compatibility of Pre-existing ESQL/C Applications with Current Library Versions

You specify the **esql** command-line options in [Figure 13-2 on page 13-6](#) to tell the **esql** command which version of the Informix libraries to link with the 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 ESQL/C, you receive new copies of the Informix general libraries. Whether you need to recompile and relink your existing ESQL/C applications to run with these new copies depends on the factors that the following table describes.

Important: *Static versions of Informix general libraries are available only for ESQL/C on a UNIX operating system.*



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 Thread-safe static	Only if the application needs to take advantage of a new feature in the new release
Informix general libraries in new release have a new major-version number	Shared Thread-safe shared	Only if the application needs to take advantage of a new feature in the new release
Informix general libraries in new release have a new API-version number	Shared Thread-safe shared	Must recompile and relink

UNIX

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” on page 13-13](#). ♦

WIN NT/95

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:

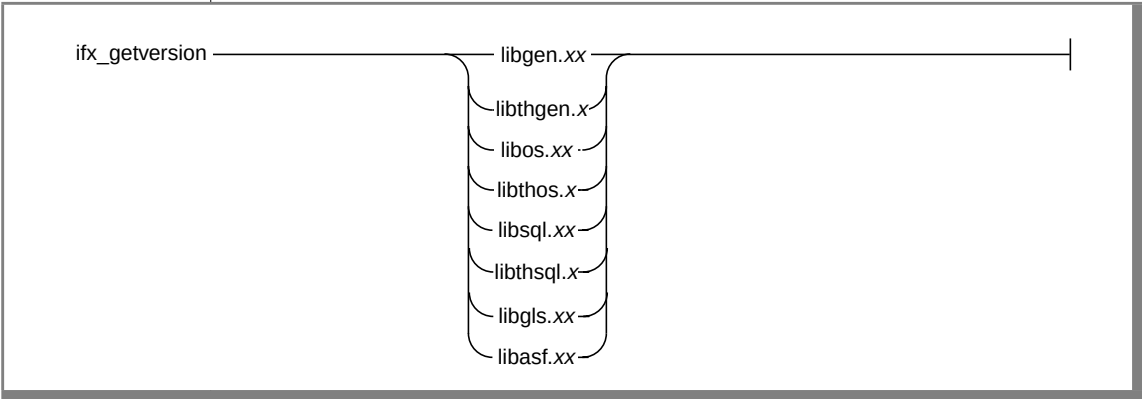
```
- - - - - ISQLT09A.DLL
INFORMIX-SQL Version 9.20T1N79
♦
```

UNIX

Using the ifx_getversion Utility

To obtain the complete version name of an Informix library, use the **ifx_getversion** utility. Before you run **ifx_getversion**, set the **INFORMIXDIR** environment variable to the directory in which your Informix product is installed.

The **ifx_getversion** utility has the following syntax.



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 sample output that the **ifx_getversion** utility generates for the **libgen** Informix library:

```
INFORMIX LIBGEN Version 9.10.U1N21  
Copyright (C) 1991-1997 Informix Software, Inc.
```

Output of **ifx_getversion** depends on the version of ESQL/C software that is installed on your system. ♦

Checking the API Version of a Library

When you invoke an ESQL/C application that is linked with shared 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.

WIN NT/95

In a Windows environment, a developer can easily verify the name of the shared library DLL, namely **isqltnx.dll**, where **nn** stand for the version number, and **x** stand for the API version. ♦

UNIX

In case of an 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, ESQL/C does this check for you. 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 ESQL/C product. [Figure 13-3 on page 13-9](#) shows where the API version appears in the shared library name.

ESQL/C uses an 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, ESQL/C checks the values of special macro definitions in the executable file. When the 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 ESQL/C preprocessor automatically includes the **sqlhdr.h** file in all ESQL/C executable files that it generates.

If the API version of the libraries in this executable file are not compatible, ESQL/C returns a runtime error that indicates which library is not compatible. You must recompile your 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 ESQL/C application, you must explicitly link the **checkapi.o** file with your application. Otherwise, ESQL/C might generate an error at link time of the form:

```
undefined ifx_checkAPI()
```



Creating Thread-Safe ESQL/C Applications

ESQL/C provides shared and static thread-safe as well as shared and static default versions of the Informix general libraries on a UNIX operating system, and only thread-safe shared libraries in a Windows environment.

A thread-safe 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.

UNIX

In the case of 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. ♦

WIN NT/95

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 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 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 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 ESQL/C preprocessor, **esqlc**. With the **-thread** option, the ESQL/C preprocessor generates thread-safe code that different threads can execute concurrently.

Characteristics of Thread-Safe ESQL/C Code

The thread-safe 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, ESQL/C allocates **sqllda** structures dynamically and binds host variables to these **sqllda** structures at runtime. For more information on **sqllda** structures to perform dynamic SQL, see [Chapter 12, “Working with the Database Server.”](#)
- 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” on page 13-27.](#)

Because of the preceding differences, the thread-safe C source file (.c) that the ESQL/C preprocessor generates is different from the non-threaded C source file. Therefore, you cannot link 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 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 ♦

Concurrent Active Connections

In a thread-safe 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 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 has been 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 could 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.

Figure 13-4 shows a thread-safe ESQL/C application that establishes three concurrent connections, each of which is active.

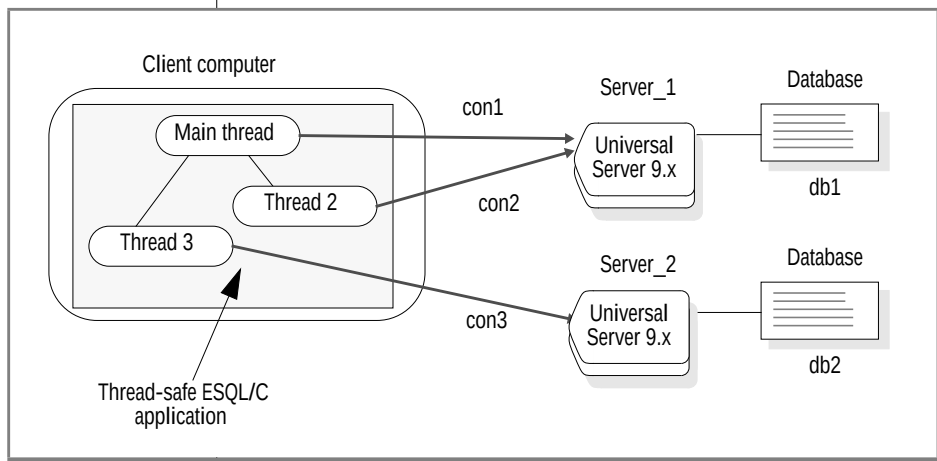


Figure 13-4
Concurrent
Connections in a
Thread-Safe
ESQL/C Application

In Figure 13-4, the 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 13-4 are concurrently active and can execute SQL statements. The following code fragment establishes the connections that Figure 13-4 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 12-30](#).

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 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 [Informix Guide to SQL: Syntax](#).) The code fragment in Figure 13-5 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 13-5
*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 CPU 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 13-4 on page 13-19](#) 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_10** uses a SET CONNECTION statement loop (see [Figure 13-7 on page 13-28](#)) to wait for **con2** to become available. Once **thread_20** makes **con2** dormant, other threads can use this connection. At this time, the SET CONNECTION statement in **thread_10** is successful and **thread_10** 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 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. Informix recommends 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, 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 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 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 13-6, the code fragment prepares SQL statements during the connection initialization and executes them later in the program.

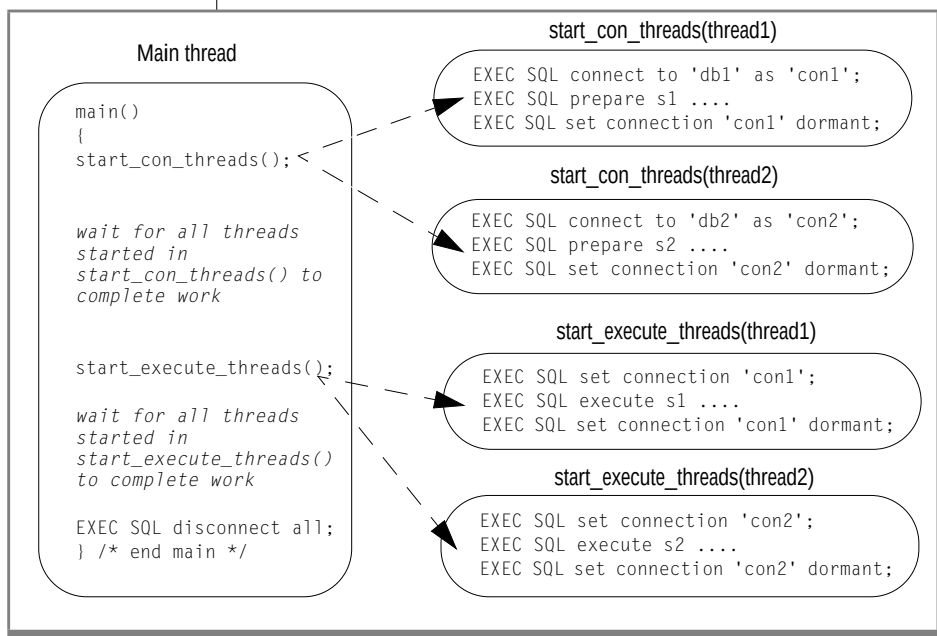


Figure 13-6
Using Prepared
SQL Statements
Across Threads

The code fragment in [Figure 13-6 on page 13-23](#) 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 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 13-22](#)). The following code fragment shows a modified version of the **start_con_thread()** function (in [Figure 13-6 on page 13-23](#)). 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 13-33](#).

Environment Variables Across Threads

Environment variables are not thread-scoped in a thread-safe 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.

Decimal Functions

The **dececv()** and **decfcvt()** functions of the 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_dececv()** and **ifx_decfcvt()**, respectively. For more information, see [“Using ESQL/C Thread-Safe Decimal Functions” on page 13-30](#).

DCE Restrictions

A thread-safe 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 ESQL/C application. For more information, refer to your DCE documentation.

Operating-System Calls

You must substitute DCE thread-jacket routines for all operating-system calls within the thread-safe 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()**, Informix recommends that the child process call **sqldetach()** before it executes any ESQL/C statements. For more information on **sqldetach()**, see [page 12-56](#).

Resource Allocation

Informix recommends 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 (available on UNIX operating systems only) or shared Informix general libraries when you specify the **-thread** command-line option.

UNIX

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 ESQL/C module on a UNIX operating system:

1. Install the DCE thread package on the same client computer as the ESQL/C product. For more information, refer to your DCE installation instructions.
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
```

Important: This version of ESQL/C supports only the DCE thread package.

For more information on **THREADLIB**, see the [Informix Guide to SQL: Reference](#).



3. 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 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

When you specify the **-thread** command-line option to **esql**, the 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 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 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 13-7 shows an excerpt from the **sqlca.h** file with the thread-scoped definitions for ESQL/C status variables.

```
⋮
extern struct sqlca_s sqlca;

extern long SQLCODE;

extern char SQLSTATE[];
#else /* IFX_THREAD */
extern long * 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 13-7
*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 ESQL/C application, you must perform the following steps:

1. In your 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 ESQL/C source file, specify the **-thread** command-line option of the **esql** command.

The **-thread** option tells the ESQL/C preprocessor to generate thread-safe C code when it translates SQL and ESQL/C statements. This thread-safe code includes calls to thread-safe functions in the Informix DLLs.

If you are not creating an 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 **dececv()** and **decfcvt()** functions of the 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
ifx_dececv()	Thread-safe version of the dececv() ESQL/C function
ifx_decfcvt()	Thread-safe version of the decfcvt() ESQL/C function

Both of these functions convert a decimal value to an ASCII string and return it in the user-defined buffer.

When you compile your 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 on the following pages.

ifx_dececv(), ifx_decfcvt()

The **ifx_dececv()** and **ifx_decfcvt()** functions are the thread-safe versions of the **dececv()** and **decfcvt()** ESQL/C library functions.

Syntax

```
int ifx_dececv(np, ndigit, decpt, sign, decstr, decstrlen)
    register dec_t *np;
    register int ndigit;
    int *decpt;
    int *sign;
    char *decstr;
    int decstrlen;
```

```
int ifx_decfcvt(np, ndigit, decpt, sign, decstr, decstrlen)
    register dec_t *np;
    register int ndigit;
    int *decpt;
    int *sign;
    char *decstr;
    int decstrlen;
```

<i>np</i>	is a pointer to a decimal structure that contains the decimal value to be converted.
<i>ndigit</i>	is the length of the ASCII string for ifx_dececv() . It is the number of digits to the right of the decimal point for ifx_decfcvt() .
<i>decpt</i>	is a pointer to an integer that is the position of the decimal point relative to the beginning of the string. A negative value for <i>*decpt</i> means the position is located to the left of the returned digits.
<i>sign</i>	is a pointer to the sign of the result. If the sign of the result is negative, <i>*sign</i> is nonzero; otherwise, it is zero.
<i>decstr</i>	is the user-defined buffer where the function returns the converted decimal value.
<i>decstrlen</i>	is the length, in bytes, of the <i>decstr</i> buffer that the user defines.

ifx_dececv(), *ifx_decfcvt()*

Usage

The **ifx_dececv()** function is the thread-safe version of the **dececv()** 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 on how to use **dececv()** and **decfcvt()**, see [page 4-26](#).

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.

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() */

/*****
 * 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() */

```

Sample 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 ESQL/C performs and you must register those functions with ESQL/C. The following list shows all of the functions that a multithreaded ESQL/C application requires and describes what each function must do.

1. **int 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);
```

2. **int 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.

3. **int 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()**.

4. **int 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.

5. **int 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.

6. **int 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.

7. **int 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.

8. **int 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.

9. **int 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.

10. **int 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.

11. **int 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.

12. **int ifxOS_th_condattr_delete(ifxOS_th_condattr_t *cond_attr)**

This routine deletes the condition variable attribute object **cond_attr**. The routine should have the same functionality as the DCE **pthread_condattr_delete()**, or POSIX **pthread_condattr_destroy()** routines.

13. **int 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.

14. **int 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.

15. **int 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. The routine should have the same functionality as the DCE **pthread_keycreate()**, or the POSIX **pthread_key_create()** routines.

16. **int ifxOS_th_getspecific(ifxOS_th_key_t key, ifxOS_th_addr_t *tcb)**

This routine obtains the thread-specific data associated with the key **key**. The routine should have the same functionality as the DCE **pthread_getspecific()**, or the POSIX **pthread_getspecific()** routines.

17. **int ifxOS_th_setspecific(ifxOS_th_key_t key, ifxOS_th_addr_t tcb)**

This routine sets the thread-specific data in **tcb** associated with the key **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. The 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 **int**.

- **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. The 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 **int**.

- **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**:

UNIX

```

/* Prototypes for the dynamic thread functions */

int ifx_th_once(mutex_t *pblock, void (*pfn)(void), int *init_data);
int ifx_th_mutexattr_create(int *mutex_attr);
int ifx_th_mutexattr_setkind_np(int *mutex_attr, int kind);
int ifx_th_mutexattr_delete(int *mutex_attr);
int ifx_th_mutex_init(mutex_t *mutexp, int mutex_attr);
int ifx_th_mutex_destroy(mutex_t *mutexp);
int ifx_th_mutex_lock(mutex_t *mutexp);
int ifx_th_mutex_trylock(mutex_t *mutexp);
int ifx_th_mutex_unlock(mutex_t *mutexp);
int ifx_th_condattr_create(int *cond_attr);
int ifx_th_cond_init(cond_t *condp, int cond_attr);
int ifx_th_condattr_delete(int *cond_attr);
int ifx_th_cond_destroy(cond_t *condp);
int ifx_th_cond_timedwait(cond_t *sleep_cond, mutex_t *sleep_mutex, timestruc_t
    *t);
int ifx_th_keycreate(thread_key_t *allkey, void (*AllDestructor)(void *));
int ifx_th_getspecific(thread_key_t key, void **tcb);
int ifx_th_setspecific(thread_key_t key, void *tcb);

/*
 * The functions . . . *
 */

int ifx_th_once(mutex_t *pblock, void (*pfn)(void), int *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.                                     */

int ifx_th_mutexattr_create(int *mutex_attr)
{
    *mutex_attr = 0;
    return(0);
}

/* Mutex attributes are not supported in solaris kernel threads */
int ifx_th_mutexattr_setkind_np(int *mutex_attr, int kind)
{
    *mutex_attr = 0;
    return(0);
}

/* Mutex attributes are not supported in solaris kernel threads */
int ifx_th_mutexattr_delete(int *mutex_attr)
{
    return(0);
}

```

```
}

int ifx_th_mutex_init(mutex_t *mutexp, int mutex_attr)
{
    return(mutex_init(mutexp, USYNC_THREAD, (void *)NULL));
}

int ifx_th_mutex_destroy(mutex_t *mutexp)
{
    return(mutex_destroy(mutexp));
}

int ifx_th_mutex_lock(mutex_t *mutexp)
{
    return(mutex_lock(mutexp));
}
/* Simulate mutex_trylock using mutex_lock */
int ifx_th_mutex_trylock(mutex_t *mutexp)
{
    int 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);
}

int ifx_th_mutex_unlock(mutex_t *mutexp)
{
    return(mutex_unlock(mutexp));
}

/* Condition attributes are not supported in solaris kernel threads */
int ifx_th_condattr_create(int *cond_attr)
{
    *cond_attr = 0;
    return(0);
}

int ifx_th_cond_init(cond_t *condp, int cond_attr)
{
    return(cond_init(condp, USYNC_THREAD, (void *)NULL));
}

int ifx_th_condattr_delete(int *cond_attr)
{
    return(0);
}

int ifx_th_cond_destroy(cond_t *condp)
{
    return(cond_destroy(condp));
}

int ifx_th_cond_timedwait(cond_t *sleep_cond, mutex_t *sleep_mutex, timestruc_t
*t)
{
    return(cond_timedwait(sleep_cond, sleep_mutex, t));
}
```

```

int ifx_th_keycreate(thread_key_t *allkey, void (*AllDestructor)(void *))
{
    return(thr_keycreate(allkey, AllDestructor));
}

int ifx_th_getspecific(thread_key_t key, void **tcb)
{
    return(thr_getspecific(key, tcb));
}

int ifx_th_setspecific(thread_key_t key, void *tcb)
{
    return(thr_setspecific(key, tcb));
}

```



Registering the Dynamic Thread Functions

Your ESQL/C application must use the **ifxOS_set_thrfunc()** function to register the dynamic thread functions with ESQL/C. The following declaration describes the **ifxOS_set_thrfunc()** function.

```
int ifxOS_set_thrfunc(int func, unsigned long (*funcptr)())
```

The first parameter, **func**, is an integer 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 step 1.

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, (unsigned long (*)( ))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 step 1. FUNCFAIL is defined to be -1.

```
dynthr_init()
{
    if (ifxOS_set_thrfunc(TH_ONCE, (unsigned long (*)())ifx_th_once) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEXATTR_CREATE,
        (unsigned long (*)())ifx_th_mutexattr_create) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEXATTR_SETKIND,
        (unsigned long (*)())ifx_th_mutexattr_setkind_np) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEXATTR_DELETE,
        (unsigned long (*)())ifx_th_mutexattr_delete) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_INIT,
        (unsigned long (*)())ifx_th_mutex_init) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_DESTROY,
        (unsigned long (*)()) ifx_th_mutex_destroy) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_LOCK,
        (unsigned long (*)()) ifx_th_mutex_lock) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_UNLOCK,
        (unsigned long (*)())ifx_th_mutex_unlock) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_TRYLOCK,
        (unsigned long (*)())ifx_th_mutex_trylock) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_CONDATTR_CREATE,
        (unsigned long (*)())ifx_th_condattr_create) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_CONDATTR_DELETE,
        (unsigned long (*)())ifx_th_condattr_delete) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_COND_INIT,
        (unsigned long (*)())ifx_th_cond_init) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_COND_DESTROY,
        (unsigned long (*)())ifx_th_cond_destroy) == FUNCFAIL)
        return FUNCFAIL;
}
```

```
if (ifx0S_set_thrfunc(TH_COND_TIMEDWAIT,
    (unsigned long (*)())ifx_th_cond_timedwait) == FUNCFAIL)
    return FUNCFAIL;

if (ifx0S_set_thrfunc(TH_KEYCREATE,
    (unsigned long (*)())ifx_th_keycreate) == FUNCFAIL)
    return FUNCFAIL;

if (ifx0S_set_thrfunc(TH_GETSPECIFIC,
    (unsigned long (*)())ifx_th_getspecific) == FUNCFAIL)
    return FUNCFAIL;

if (ifx0S_set_thrfunc(TH_SETSPECIFIC,
    (unsigned long (*)())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/include/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/include/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/include/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 -I Preprocessor Options

You must compile the application using the **-thread** and the **-I** preprocessor options. The **-thread** option indicates that you are linking thread-safe libraries instead of the default Informix shared libraries. The **-I** 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
```

Dynamic SQL

Section



Using Dynamic SQL

Using Dynamic SQL	14-4
Assembling and Preparing the SQL Statement	14-5
Assembling the Statement.	14-5
Preparing Statements That Have Collection Variables	14-8
Checking the Prepared Statement	14-9
Executing the SQL Statement	14-10
Freeing Resources	14-13
Using a Database Cursor	14-14
Receiving More Than One Row	14-14
A Select Cursor	14-16
A Function Cursor	14-16
Sending More Than One Row	14-18
Naming the Cursor	14-19
Optimizing Cursor Execution	14-20
Sizing the Cursor Buffer	14-20
Automatically Freeing a Cursor.	14-24
Deferring Execution of the PREPARE	14-30
Optimizing OPEN, FETCH, and CLOSE	14-35
Restrictions on OPTOFC	14-36
Enabling the OPTOFC Feature	14-37
Using OPTOFC and Deferred-PREPARE Together	14-38
SQL Statements That Are Known at Compile Time	14-40
Executing Non-SELECT Statements.	14-40
Using PREPARE and EXECUTE.	14-41
Using EXECUTE IMMEDIATE	14-42
Executing SELECT Statements	14-43
Using PREPARE and EXECUTE INTO	14-43
Using a Select Cursor	14-45

Executing User-Defined Routines in Universal Server	14-46
A User-Defined Procedure.	14-47
A User-Defined Function	14-48
Executing Statements with Input Parameters	14-51
Using an EXECUTE USING Statement	14-52
Using an OPEN USING Statement	14-54
SQL Statements That Are Not Known at Compile Time	14-58

A

static SQL statement is one for which all the information is known at compile time. For example, the 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;
```

To execute a static statement, you just embed the statement directly in your program. (See [“Embedding SQL Statements” on page 1-17.](#))

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. Such applications require *dynamic* SQL. Dynamic SQL allows an INFORMIX-ESQL/C program to build an SQL statement at runtime, so that the contents of the statement can be determined by user input.

This chapter discusses the following dynamic SQL topics:

- 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 on how to execute SQL statements when you do *not* know all the information at compile time, see [Chapter 15, “Determining SQL Statements.”](#)

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 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 just list it after the FROM keyword of the PREPARE statement, as shown in the following example:

```
EXEC SQL prepare slct_id from  
      'select company from customer where customer_num = 101';
```



Tip: The statement text in the preceding example is surrounded with single quotes. Double quotes are also valid, although single quotes are specified by the ANSI SQL standard.

Alternatively, you can copy the statement into a **char** variable as shown in Figure 14-1.

```
strcpy("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 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 has been 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 [page 14-10](#)). 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 on **-local**, see [Chapter 1](#) of this manual.

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 [“Executing Statements with Input Parameters” on page 14-51](#).
- You can use ESQL/C string library functions like **stcopy()** and **stcat()**. For more information, see [Chapter 3, “Working with Character and String Data Types.”](#)

Figure 14-2 shows the SELECT statement of [Figure 14-1 on page 14-6](#) 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 ones that cannot be prepared 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 [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 14-30](#).

Preparing Statements That Have Collection Variables

You use the Collection Derived Table clause with an INSERT or SELECT statement to access an ESQL/C **collection** variable. (For more information on how to use the Collection Derived Table clause and **collection** variables, see Chapter 9 of this manual.)

When you prepare a statement that manipulates an 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, 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 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 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 has been 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 on how to execute statements that require cursors, see [“Using a Database Cursor” on page 14-14](#).

- 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 14-40](#).

- 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.”](#)

Figure 14-4 summarizes how to execute the different types of prepared SQL statements.

Figure 14-4
Executing a Prepared SQL Statement

Type of SQL Statement	Input Parameters	Statement to Execute	Page Reference
Statements that do not return rows (except INSERT that is associated with a cursor)			
With no input parameters	No	EXECUTE	14-40
When number and data types of input parameters are known	Yes	EXECUTE...USING	14-52
When number and data types of input parameters are <i>not</i> known	Yes	EXECUTE...USING SQL DESCRIPTOR	16-46
		EXECUTE...USING DESCRIPTOR	17-43
INSERT that is associated with a cursor			
With no input parameters	No	OPEN	14-45
When number and data types of input parameters (insert columns) are known	Yes	OPEN...USING	14-54, 15-24
When number and data types of input parameters are <i>not</i> known	Yes	OPEN...USING SQL DESCRIPTOR	16-29
		OPEN...USING DESCRIPTOR	17-30
Statements that can return more than one row: non-singleton SELECT, SPL function			
With no input parameters	No	OPEN	14-45
When number and data types of select-list columns are <i>not</i> known	No	OPEN	16-15, 17-18

(1 of 2)

Type of SQL Statement	Input Parameters	Statement to Execute	Page Reference
When number and data types of return values are <i>not</i> known	No	OPEN	16-28, 17-28
When number and data types of input parameters are known	Yes	OPEN...USING	14-54
When number and data types of input parameters are <i>not</i> known	Yes	OPEN...USING SQL DESCRIPTOR	16-37
		OPEN...USING DESCRIPTOR	17-34
Statements that return <i>only one row</i>: singleton SELECT, any external function or an SPL function that returns only one group of values			
With no input parameters	No	EXECUTE...INTO	14-43
When number and data types of returned values are <i>not</i> known	No	EXECUTE...INTO DESCRIPTOR	16-13, 16-23
		EXECUTE...INTO SQL DESCRIPTOR	17-17, 17-28
When number and data types of input parameters are known	Yes	EXECUTE...INTO ...USING	14-52
When number and data types of input parameters are <i>not</i> known	Yes	EXECUTE...INTO ...USING SQL DESCRIPTOR	16-43
		EXECUTE...INTO ...USING DESCRIPTOR	17-41

(2 of 2)

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 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 14-24](#).

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 on how to deallocate a system-descriptor area, see [“Freeing Memory Allocated to a System-Descriptor Area” on page 16-12](#). For information on how to deallocate an **sqllda** structure, see [“Freeing Memory Allocated to an sqllda Structure” on page 17-15](#).

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 ESQL/C application. For information about how to use cursors, see the [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. Figure 14-5 summarizes the SQL statements that manage a select or function cursor.

Figure 14-5
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 on any of these statements, see their entries in the *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 14-20.

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 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 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 on how to write this type of SPL function, see the chapter on SPL in the [Informix Guide to SQL: Tutorial](#).

- An external function that is an iterator function

For information on how to write an iterator function, see the *DataBlade API Programmer's Manual*.

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 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 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 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 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. Figure 14-6 summarizes the SQL statements that manage an insert cursor.

Figure 14-6
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

(1 of 2)

Task	Insert Cursor
Clear the insert buffer and send the contents to the database server	FLUSH
Close the cursor	CLOSE
Free cursor resources	FREE

(2 of 2)

For more information on any of these statements, see their entries in the [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 14-20](#).

Naming the Cursor

In an 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 [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 1-32](#).
- 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 ESQL/C statement within the function, make sure you declare the argument as a host variable with the PARAMETER keyword. [Figure 14-7 on page 14-20](#) 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 14-7
*A Sample ESQL/C
Function That Uses
a Dynamic Cursor*

Optimizing Cursor Execution

INFORMIX-ESQL/C supports the following features that allow you to minimize network traffic when an 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 ESQL/C application uses to hold the data (except simple large-object data) in a cursor. 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, ESQL/C stores these rows in the fetch buffer.
- The *insert buffer* holds data for an insert cursor.
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, ESQL/C makes this request when the PREPARE statement executes. If the cursor does not have an associated statement, 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, ESQL/C can determine the size of a row of data. By default, 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, ESQL/C allocates the cursor buffer.

Changing Size of Cursor Buffer

An ESQL/C application might want to increase the size of the cursor buffer to increase the number of rows that can be passed at one time between the client application and the database server. When you increase the number of rows that can be passed, you reduce the number of round-trip message requests that need to be sent between the client application and the database server. The bigger the cursor buffer, the more data that can be sent in one round trip.

To increase the size of the cursor buffer, you can set either of the following variables in the client environment:

- At runtime, you can set the **FET_BUF_SIZE** environment variable to determine the size of the cursor buffer *for all select, fetch, and insert cursors in all client applications*.

For example, the following command sets the **FET_BUF_SIZE** environment variable in the C-shell environment to 20,000 bytes (20 kilobytes):

```
setenv FET_BUF_SIZE 20000
```

- At compile time, you can set the **FetBufSize** global variable in an ESQL/C application to set the size of the cursor buffer *from cursor to cursor in a particular client application*.

The **FetBufSize** global variable is defined in the ESQL/C **sqlhdr.h** header file. (The ESQL/C processor automatically includes the **sqlhdr.h** header file in all ESQL/C programs.) You set the size of the cursor buffer at the point that you make the assignment to the **FetBufSize** variable. This new size takes effect the next time ESQL/C checks the buffer size. ESQL/C checks the size of the cursor buffer when it allocates the cursor buffer. ESQL/C also checks the size of a fetch buffer when it tells the database server how many rows to send at one time.

The global variable overrides any value in the **FET_BUF_SIZE** environment variable. The following C code sets **FetBufSize** to 30,000 bytes (30 kilobytes):

```
FetBufSize = 30000;
```



Tip: Contrary to what the names might imply, the **FET_BUF_SIZE** environment variable and the **FetBufSize** global ESQL/C variable do not change the size of only the fetch buffer. They change the size of the cursor buffer, which is used for both the fetch buffer and the insert buffer.



In a client-server environment, you must set the cursor buffer size on the client side of the application because this buffer resides in the application process. Specify the size of the cursor buffer in bytes, up to the maximum size of a small integer (*short* data type) on your system. For most 32-bit platforms, this maximum is 32,767 bytes. 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.

Important: Informix extends ANSI-standard syntax to set the size of the cursor buffer.

If you increase the size of the cursor buffer, you reduce the overhead of refilling the buffer. The database server can buffer more data before it sends it to the application. A larger fetch buffer can enhance performance when you fetch a large number of rows or when a single row is large. The greater the size of the fetch buffer, the fewer number of times the application needs to wait while the database server retrieves a large number of rows or a few large rows. However, when you fetch only a few rows (unless the row is very large), the performance gain is not as apparent.

Automatically Freeing a Cursor

When an 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, 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
```

IUS

ODS

OWS

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 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 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 on the syntax and use of the SET AUTOFREE statement, see [“Using the SET AUTOFREE Statement” on page 14-26](#). For more information on the **IFX_AUTOFREE** environment variable, see the [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 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 has not been associated with any declared cursor
- If the cursor with the prepared statement has been freed but the prepared statement has 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 has been 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 on the syntax and use of the SET AUTOFREE statement, see the [Informix Guide to SQL: Syntax](#).



IUS

ODS

OVS

Deferring Execution of the PREPARE Statement

When an 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, 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 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 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 features 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 SETDEFERRED_PREPARE Statement” on page 14-32.](#)

Enabling the Deferred-PREPARE Feature

You can enable the Deferred-PREPARE feature for an 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 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 on the syntax and use of the SET DEFERRED PREPARE statement, see “Using the SETDEFERRED_PREPARE Statement.” For more information on the **IFX_DEFERRED_PREPARE** environment variable, see the [Informix Guide to SQL: Reference](#).

Using the SETDEFERRED_PREPARE Statement

From within an 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 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++;
}
```



Optimizing OPEN, FETCH, and CLOSE

When an 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:

- ESQL/C saves one round trip because it does not send the OPEN statement as a separate command.

When 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 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.

- 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, ESQL/C does not need to send the CLOSE statement to the database server.



Important: ESQL/C does not send the CLOSE statement to the database server. However, if you include the CLOSE statement, no error is generated.

IUS

ODS

OWS

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 has been 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 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.

UNIX

On UNIX operating systems, you can set **OPTOFC** from within 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");
```



WIN NT

WIN 95

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 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.
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, 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, 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, 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 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 **lvvarchar** host variable to hold the data because ESQL/C cannot convert the opaque-type data from its internal format (which it receives from the database server) to its external (**lvvarchar**) format.

- The FetArrSize feature does not work when both the Deferred-PREPARE and OPTOFC features are enabled. When these two features are enabled, 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 on the AUTOFREE feature, see [“Automatically Freeing a Cursor” on page 14-24](#). ♦

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 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 14-18](#).

For a list of SQL statements that cannot be prepared, see the entry for the PREPARE statement in the [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 **tablename** and **search_condition**. The **tablename** 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, **tablename**, 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",
        tablename, 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 14-52](#).

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 14-43](#). 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 14-18](#).

- 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 15-26](#).

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 14-52](#).) 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 [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 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 14-5](#).

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 14-52](#).



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 has been 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 ESQL/C program, as shown in the following example:

```
EXEC SQL select order_date from orders where order_num = 1004;
```

Figure 14-8 shows how to execute the **items_pct()** SPL function (which [Figure 16-3 on page 16-24](#) 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 14-8
*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 14-8 on page 14-44](#).

```
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 15-23](#).

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 ESQL/C application to handle multiple rows that a query returns.

Your 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 14-5](#).
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 on cursors and cursor-management statements, see [“Using a Database Cursor” on page 14-14](#).



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 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.

IUS

Executing User-Defined Routines in Universal Server

In Universal Server, 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.

Task	SQL Statement for User Defined Routines	
	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 on these statements, see the [Informix Guide to SQL: Syntax](#).

Universal Server 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 on how to write an external routine in C, see *Extending INFORMIX-Universal Server: Data Types*.
- 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 on how to write a stored routine, see the [Informix Guide to SQL: Tutorial](#).



Tip: In earlier versions of Informix products, the term “stored procedure” was used for both SPL procedures and SPL functions. That is, a stored procedure could include the `RETURN` statement to return values. For backward compatibility with earlier products, Universal Server continues to support the execution of SPL functions with the `EXECUTE PROCEDURE` statement. However, for new SPL routines, Informix recommends 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 [“Executing Statements with Input Parameters”](#) on page 14-51.

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 on the syntax of the `EXECUTE PROCEDURE` statement, see the [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 ([page 14-41](#))
- the `EXECUTE IMMEDIATE` statement ([page 14-42](#))

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 16-3 on page 16-24](#) defines):

```
EXEC SQL execute function items_pct(\"HSK\")
      into :manuf_percent;
```

For more information on the syntax of the EXECUTE FUNCTION statement, see the [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 ([page 15-23](#)). Both the SELECT and the user-defined function return values to the 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 [page 15-26](#).



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 ESQL/C program must take the following actions:

1. Declare host variables to receive the values that the database server returns.

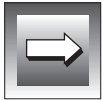
For a prepared EXECUTE FUNCTION statement, these values are the return values of the user-defined function.

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 14-5](#).

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, 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 [page 14-50](#).

EXT

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 on these data types, see [Chapter 9, “Working with INFORMIX-Universal Server Complex Data Types.”](#)) ♦

SPL

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 14-43](#).

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 ESQL/C application to handle the multiple rows that a user-defined function returns. Your 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 14-5](#).

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.

EXT

Only an external function that is an iterator function can return *more than one row* of data. For information on how to write an iterator function, see the *DataBlade API Programmer’s Manual*. ♦

SPL

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 on cursors and cursor-management statements, see [“Using a Database Cursor” on page 14-14](#). ♦

Executing Statements with Input Parameters

An *input parameter* is a placeholder in an SQL statement that indicates that the actual value will be 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 could 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 15-25](#).

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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).
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, see [“Determining Unknown Input Parameters” on page 15-25](#).

For more information on the USING clause, see the entry for EXECUTE in the [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 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 on the USING clause, see the entry for OPEN in the [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("DEM02 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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

```
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 terminal
25 * 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 (strcmp(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 (strcmp(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 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.”](#)

Determining SQL Statements

Using Dynamic-Management Structures	15-4
A System-Descriptor Area	15-5
Fixed-Length Portion	15-7
An Item Descriptor	15-7
An sqlda Structure.	15-10
Fixed-Length Portion	15-11
An sqlvar_struct Structure	15-11
Descriptive Information	15-14
Using the DESCRIBE Statement	15-14
Determining Statement Type	15-16
Determining the Data Type of a Column	15-19
Informix-Specific SQL Data Types	15-20
X/Open SQL Data Types	15-20
Constants for ESQL/C Data Types.	15-21
Checking for a WHERE Clause	15-22
Determining Statement Information at Runtime	15-22
Handling an Unknown Select List	15-23
Handling an Unknown Column List	15-24
Determining Unknown Input Parameters.	15-25
Determining Return Values Dynamically	15-26
Handling Statements That Contain User-Defined Data Types	15-27
SQL Statements with Opaque-Type Columns	15-27
SQL Statements with Distinct-Type Columns	15-30
Using a Fetch Array	15-33
Allocating Memory for the Fetch Arrays	15-37
Obtaining Values from Fetch Arrays	15-42
Freeing Memory for a Fetch Array	15-45

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.

This chapter contains the following topics 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 Structures

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 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 **sqllda** 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 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 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.

Figure 15-1 shows what a system-descriptor area looks like for two values.

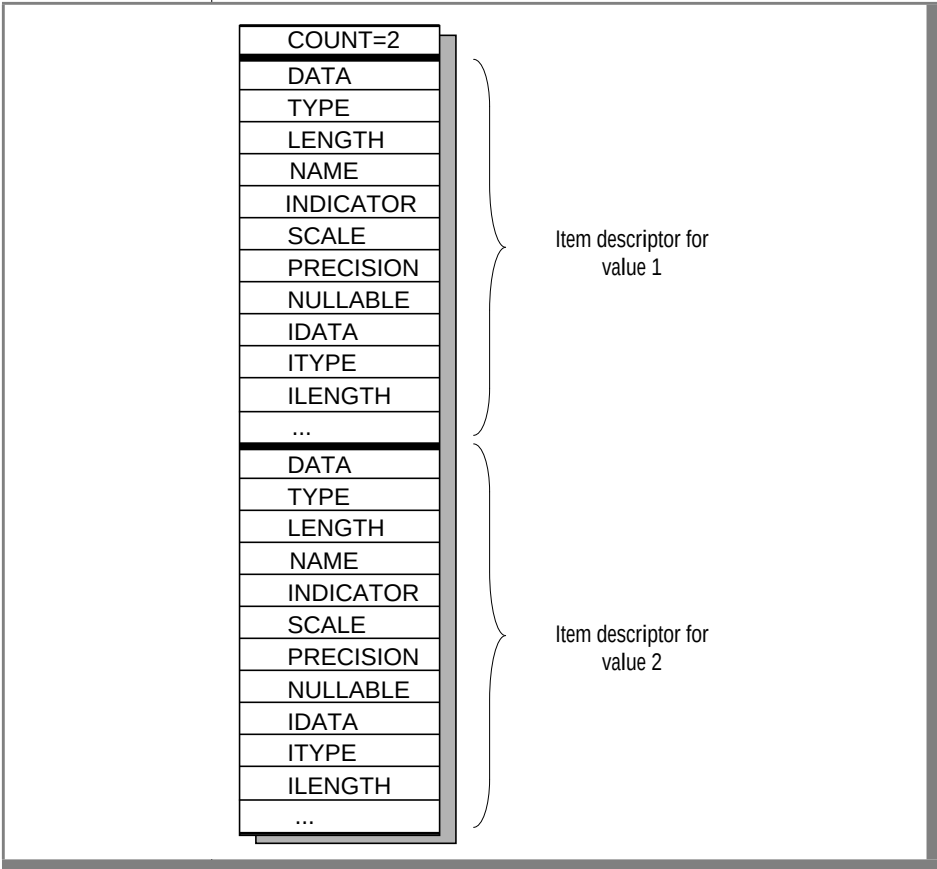


Figure 15-1
Schematic That Shows System-Descriptor Area for Two Values

Fixed-Length Portion

The fixed-size portion of the system-descriptor area consists of the single field, which Figure 15-2 shows.

Figure 15-2

Field in the Fixed-Size Portion of a System-Descriptor Area

Field	Data Type	Description
COUNT	short	The number of column values or <i>occurrences</i> 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 Figure 15-3 summarizes.

Figure 15-3

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.
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 page 15-19).
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.

(1 of 3)

Field	Data Type	Description
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: The DATA field contains non-null data. -1: The DATA field contains null data.
SCALE	short	Contains the scale of the column that is in the DATA field; defined <i>only</i> for the DECIMAL or MONEY data type.
PRECISION	short	Contains the precision of the column that is in the DATA field; defined <i>only</i> 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 <i>not</i> 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 <i>not</i> 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 page 15-19 .) The ITYPE field is <i>not</i> a standard X/Open field.
ILENGTH	short	The length, in bytes, of the user-defined indicator. The ILENGTH field is <i>not</i> a standard X/Open field.

(2 of 3)

Field	Data Type	Description
EXTYPEID	long	The extended identifier for the user-defined (opaque or distinct) or complex (collection or row) data type. Refer to the Informix Guide to SQL: Reference for a description of the extended_id column of the sysxdtypes system catalog table.
EXTYPENAME	char *	The name of the user-defined (opaque or distinct) or complex (collection or row) data type. Refer to the Informix Guide to SQL: Reference for a description of the name column of the sysxdtypes 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. Refer to the Informix Guide to SQL: Reference for a description of the owner column of the sysxdtypes system catalog.
EXTYPEOWNER-LENGTH	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. Refer to the Informix Guide to SQL: Reference for a description of the type column of the sysxdtypes system catalog.
SOURCEID	long	The extended identifier of the source data type for a distinct-type column. Refer to the Informix Guide to SQL: Reference for a description of the source column of the sysxdtypes system catalog.

(3 of 3)

For information on how to use a system-descriptor area to dynamically execute SQL statements, see [Chapter 16, “Using a System-Descriptor Area.”](#)

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 [Figure 15-8 on page 15-14](#).

Figure 15-4 shows what an **sqlda** structure looks like for two values.

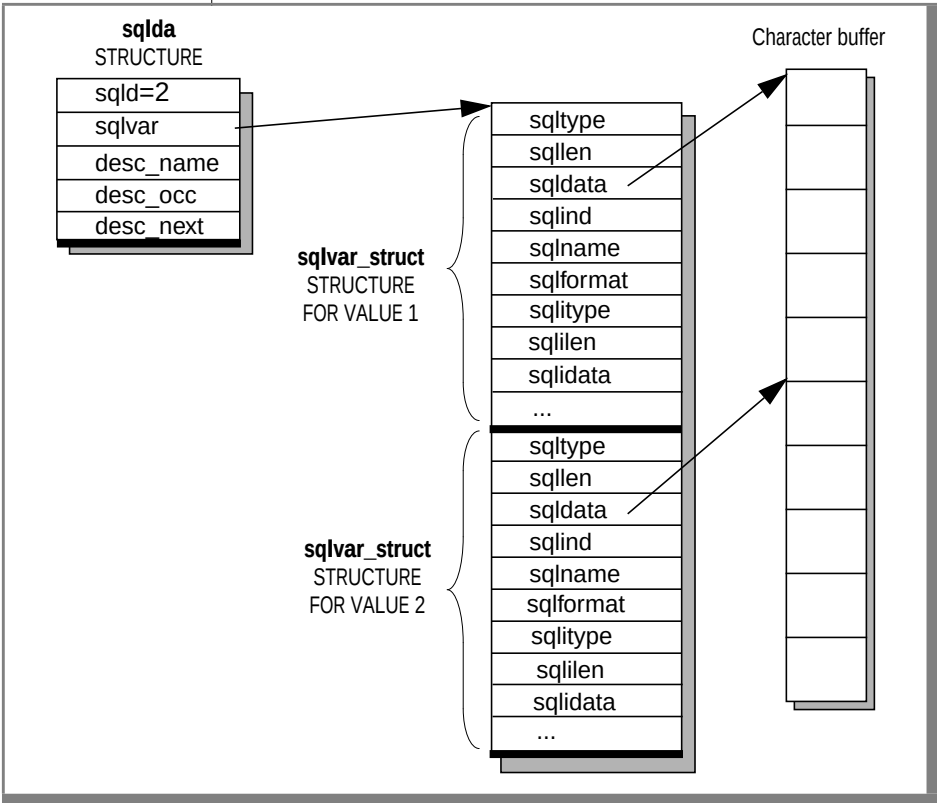


Figure 15-4
Schematic That Shows sqlda Structure for Two Values

Figure 15-4 on page 15-10 shows the column data in the **sql**da fields in a single data buffer. This data could also be stored in separate buffers. For more information, see “Allocating Memory for Column Data” on page 17-10.

Fixed-Length Portion

Figure 15-5 describes the fixed-size portion of the **sql**da structure, which consists of a single field.

Figure 15-5
*Field in the Fixed-Size Portion of an sql*da Structure

Field	Data Type	Description
sql d	short	The number of column values or <i>occurrences</i> in the sql da structure. This is the number of sql var_struct structures, one for each column. The DESCRIBE...INTO statement sets sql d to the number of described columns. You must set sql d 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 **sql**da structure points to the initial address of a sequence of **sql**var_struct structures that contain the necessary information for each variable in the set. Each **sql**var_struct structure holds a data value that can be sent to or received from the database server. Your program accesses these **sql**var_struct structures through the **sql**var field of **sql**da. Figure 15-6 and Figure 15-7 on page 15-12 summarize the variable-sized structure of **sql**da.

Figure 15-6
*Field to Access the Variable-Sized Portion of an sql*da Structure

Field	Data Type	Description
sql var	struct sql var_struct *	A pointer to the variable-sized portion of an sql da structure. There is one sql var_struct for each column value returned from or sent to the database server. The sql var field points to the <i>first</i> of the sql var_struct structures.

Figure 15-7 shows the fields in the **sqlvar_struct** structure.

Figure 15-7
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 page 15-19 .)
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 page 17-10 .)
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 *	<i>Reserved for future use.</i>
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 page 15-19 .)
sqlilen	short	The length, in bytes, of a user-defined indicator variable.
sqldata	char *	A pointer to the data of the user-defined indicator variable.
sqlxid	long	The extended identifier for the user-defined (opaque or distinct) or complex (collection or row) data type. Refer to the Informix Guide to SQL: Reference for a description of the extended_id column of the sysxdtypes system catalog table.

(1 of 2)

Field	Data Type	Description
sqltypename	char *	The name of the user-defined (opaque or distinct) or complex (collection or row) data type. Refer to the Informix Guide to SQL: Reference for a description of the name column of the sysxdtypes 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. Refer to the Informix Guide to SQL: Reference for a description of the owner column of the sysxdtypes system catalog.
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. Refer to the Informix Guide to SQL: Reference for a description of the type column of the sysxdtypes system catalog.
sqlsourceid	long	The extended identifier of the source data type for a distinct-type column. Refer to the Informix Guide to SQL: Reference for a description of the source column of the sysxdtypes system catalog.
sqlflags	long	<i>For internal use only</i>

(2 of 2)

Descriptive Information

Figure 15-8 summarizes the **sqlda** fields that describe the **sqlda** structure itself.

Figure 15-8
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 on how to use an **sqlda** structure to dynamically execute SQL statements, see [Chapter 17, “Using an sqlda Structure.”](#)

Using the DESCRIBE Statement

This section provides a brief introduction to the DESCRIBE statement. For more information about the DESCRIBE statement, refer to its entry in the [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 15-3](#).



- 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 **sqlid** 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 15-4 through 15-8](#).

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 14-30.*

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 Statement Type” on page 15-16](#).
- 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 on the column descriptions returned by DESCRIBE, see [“Handling an Unknown Select List” on page 15-23](#) and [“Handling an Unknown Column List” on page 15-24](#). For information on the data type values returned by DESCRIBE, see [“Determining the Data Type of a Column” on page 15-19](#).

- 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 15-22](#).

Determining Statement Type

The **sqltype.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 ESQL/C program that uses dynamic SQL statements, you can use the constants that Figure 15-9 shows to determine which SQL statement was prepared.

Figure 15-9
The Constants for SQL Statement Types That the sqltype.h File Defines

SQL Statement	Defined sqltype.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
CREATE DATABASE	SQ_CREATEDB	12
DROP DATABASE	SQ_DROPDB	13
CREATE TABLE	SQ_CREATAB	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

(1 of 3)

SQL Statement	Defined sqlstype.h Constant	Value
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
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_PROCASTATS	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

(2 of 3)

SQL Statement	Defined <code>sqlstype.h</code> Constant	Value
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

(3 of 3)



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 has been prepared dynamically, your 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 on [page 16-25](#) 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 **sqllda** structure, in the **sqltype** field of the **sqlvar_struct** structure for each column described

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 is used by the DESCRIBE statement.
- The **sqlxtype.h** header file contains defined constants for the X/Open SQL data types. DESCRIBE uses these values when you compile your 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 16-8](#). When you set the data type of a column in an **sqllda** 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 a sqllda Structure” on page 17-13](#).



X/O

Informix-Specific SQL Data Types

The Informix-specific SQL data types are available to a column in an Informix database. The [Informix Guide to SQL: Reference](#) describes these data types. If you do not include the **-xopen** option when you compile your 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 ESQL/C **sqltypes.h** header file.

Figure 15-10 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 15-10
*Some Informix SQL
Data-Type
Constants*

For a complete list of constants for SQL data types, see [Figure 2-5 on page 2-8](#). The integer values in Figure 15-10 are language-independent constants; they are the same in *all* Informix embedded products.

X/O

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 SQL data-type values that Figure 15-11 shows. The constants for these values are defined in the **sqlxtype.h** header file.

```
#define SQLCHAR      1
#define SQLDECIMAL   3
#define SQLINT       4
#define SQLSMINT     5
#define SQLFLOAT     6
```

Figure 15-11
*SQL Data-Type
Constants in an
X/Open
Environment*

The DESCRIBE statement uses these data types to specify the data type of a column (or a return value) when you include the **-xopen** option to compile your ESQL/C program. ♦

Constants for ESQL/C Data Types

The **sqltypes.h** header file also contains defined constants for the ESQL/C data types. The ESQL/C data types are assigned to host variables in an ESQL/C program. If your program initializes a column description, it usually obtains the column value from an ESQL/C host variable. To set the column data type for this value, the program must use the ESQL/C data types.

Figure 15-12 shows only some of the ESQL/C data type entries in the **sqltypes.h** header file. For a complete list of constants for ESQL/C data types, see [Figure 2-5 on page 2-8](#).

```
#define CCHARTYPE      100
#define CSHORTTYPE    101
#define CINTTYPE      102
#define CLONGTYPE     103
#define CFLOATTYPE    104
#define CDOUBLETTYPE  105
:
:
```

Figure 15-12
*Some ESQL/C Data-
Type Constants
from the sqltypes.h
Header File*

Within an ESQL/C program that uses dynamic SQL statements, you can use the constants that are shown in Figure 15-12 to set the data types of the associated host variables. Use the 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 on [page 16-31](#) stores a TEXT value into a database table.

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 "01107".

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 [page 16-46](#); for information on how to use an **sqllda** structure, see [page 17-43](#).

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 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 [page 1-116](#) of this manual), 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 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 **sqllda** 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.

X/O



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 14-43](#).

For information on how to identify columns in the select list of a *SELECT* statement with a system-descriptor area, see [page 16-13](#). To use an **sqlda** structure, see [page 17-17](#).

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 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 14-40](#).

For information on how to identify columns in the *VALUES* column list of an *INSERT* statement with a system-descriptor area, see [page 16-29](#). To use an **sqlda** structure, see [page 17-30](#).

X/O



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 [page 14-51](#)). 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 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 **sqllda** 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 [“Executing Statements with Input Parameters” on page 14-51](#).

For information on how to handle input parameters in the WHERE clause of a dynamic SELECT statement with a system-descriptor area, see [page 16-36](#); to use an **sqlda** structure, see [page 17-32](#). For information on how to handle input parameters as arguments of a user-defined function with a system-descriptor area, see [page 16-43](#); to use an **sqlda** structure, see [page 17-41](#). For information on how to handle input parameters in the WHERE clause of a dynamic UPDATE or DELETE statement with a system-descriptor area, see [page 16-46](#); to use an **sqlda** structure, see [page 17-43](#).

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, refer to [“Executing User-Defined Routines in Universal Server” on page 14-46](#) for more information.



X/O

For information on how to use a system-descriptor area to hold function return values, see [page 16-22](#). To use an **sqllda** structure to hold return values, see [page 17-26](#).

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 **sqllda** structure) match the data type of the data you insert into an opaque-type column.
- 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. Figure 15-13 shows the type and length fields for the dynamic-management structures.

Figure 15-13

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_length;
    int extrn_type;
EXEC SQL END DECLARE SECTION;
:
:
EXEC SQL allocate descriptor 'desc1' with max 100;
EXEC SQL prepare ins_stmt from
    'insert into tabl (opaque_col) values(?)';
EXEC SQL describe ins_stmt using sql descriptor 'desc1';

/* At this point the TYPE field of the item descriptor is
 * SQLUDTFIXED
 */

strcpy("(1, 2, 3, 4)", extrn_value);
extrn_length = 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_length;
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, 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
sqllda 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 ESQL/C handles truncation of opaque-type data that is larger than 32 kilobytes. When 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, ESQL/C truncates the data to 32 kilobytes. 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
sqllda 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 15-14.](#)) 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. ([Figure 15-13 on page 15-28](#) 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(<i>type_id</i>)
BOOLEAN	SQLDBOOLEAN	ISDISTINCTBOOLEAN(<i>type_id</i>)
Any other data type	SQLDISTINCT	ISDISTINCTTYPE(<i>type_id</i>)

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 + SQLDLVARIABLE
        *   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 **sqllda** 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 `ESQL/C` fetches a TEXT or BYTE column, the database server returns the descriptor for the column.
- `ESQL/C` subsequently requests the database server to obtain the column data.

When you use a fetch array, `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 on how to declare and use sqlda structures, see [“An sqlda Structure” on page 15-9](#).
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 15-37](#) below.
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 on the **sqlerrd** array, see [Chapter 11, “Exception Handling.”](#) For more information on obtaining the column values, see [“Obtaining Values from Fetch Arrays” on page 15-42](#).

6. Repeat Steps 4 and 5 until all rows are fetched.
7. Free the memory that the **sqllda** structure uses.

As with other uses of the **sqllda** structure, ESQL/C does not release resources for this structure. Your application must free memory allocated the **sqllda** structure when it no longer needs it. For more information, see [“Freeing Memory for a Fetch Array” on page 15-45](#).



Important: The **FetArrSize** feature does not work when both the *Deferred-PREPARE* and *OPTOFC* features are enabled. When these two features are enabled, 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 **sqllda** structure. These functions are described in the following sections.

```
EXEC SQL include locator.h;
EXEC SQL include sqltypes.h;

#define BLOBSIZE 32276;

main()
{
    int i = 0;
    int row_count;

    /*****
    * Step 1: declare an sqllda structure to hold the retrieved column
    *         values
    *****/
    struct sqllda *da_ptr;

    EXEC SQL connect to 'stores7';
    if ( SQLCODE < 0 )
    {
        printf("CONNECT failed: %d\n", SQLCODE)
        exit(0);
    }

    /* create a table on which to query */
    EXEC SQL create table blob_tab (int_col integer, blob_col byte);

    /* code to insert values into blob_tab would go here */
    load_db();

    /* Prepare the SELECT on the tab1 table */
    EXEC SQL prepare selct_id 'select * from tab1';
```

```
/* *****
 * 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;

/* *****
 * Step 3: initialize the sqlda structure to hold fetch arrays for
 *         columns
 * *****/

row_size = init_sqlda(da_ptr, 0);

/* declare and open a cursor for the prepared SELECT */
EXEC SQL declare curs for selct_id;
EXEC SQL open curs;

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);
cleanup_db();
}

```

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 ESQL/C can retrieve. It's value is 32767 bytes (32 kilobytes). If the size of the fetch array is greater than MAXSMINT, 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 global variable indicates. For information about the size of the fetch buffer, see “Optimizing Cursor Execution” on page 14-20 .
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 on the rtypmsize() function, see Chapter 2, “INFORMIX-ESQL/C Data Types.”

However, if you set **FetArrSize** so that the following relationship is true,

$$(\text{FetArrSize} * \text{row size}) > \text{FetBufSize}$$

ESQL/C automatically resizes the fetch buffer (**FetBufSize**) as follows to hold the size of the fetch array:

$$\text{FetBufSize} = \text{FetArrSize} * \text{row size}$$

If the result is greater than 32 kilobytes (MAXSMINT), ESQL/C sets **FetBufSize** to 32 kilobytes and **FetArrSize** as follows:

$$\text{FetArrSize} = \text{MAXSMINT} / (\text{row size})$$

Important: The **FetArrSize** global variable can be used in thread safe 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 **loc_t** structures.



4. For each simple large object column (TEXT or BYTE), initialize the **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 ESQL/C automatically allocate memory for the simple large object columns. For more information on how to initialize a **loc_t** structure to retrieve simple large objects in memory, see [“Selecting a Simple Large Object into Memory” on page 8-15](#).
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(in_da, print)
struct sqlda *in_da;
int print;
{
    int i, j, row_size=0, msglen=0, num_to_alloc;
    struct sqlvar_struct *col_ptr;
    loc_t *temp_loc;
    char *type;

    if (print)
        printf("columns: %d. ", in_da->sqld);

/* Step 1: determine row size */
    for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++, col_ptr++)
    {

```

Allocating Memory for the Fetch Arrays

```
/* 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->sqlen; /* get database sizes */

/* calculate size for C data: string columns get extra byte added
 * to hold null terminator */

    col_ptr->sqlen = rtypmsize(col_ptr->sqltype, col_ptr->sqlen);

/* 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->sqlen;
}
if (print)
    printf("Total row size = %d\n", row_size);

/* 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 == -1) /* 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;

/* set type in sqlvar_struct structure to corresponding C type */

for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqlld; i++, col_ptr++)
{
    switch(col_ptr->sqltype)
    {
        case SQLCHAR:
            type = "char ";
            col_ptr->sqltype = CCHARTYPE;
            break;

        case SQLINT:
            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 loc_t structures for TEXT or BYTE column */

        temp_loc = (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 loc_t structures to hold
blob values in a user-defined buffer in memory */

        byfill(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);
            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;
    }

```

Obtaining Values from Fetch Arrays

```
/* 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 on how to allocate memory for the **sqldata** field, see [“Allocating Memory for the sqlda Structure” on page 17-6](#).

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 be implemented to make this function complete.
*****/
void print_sqlda(sqlda, count)
struct sqlda *sqlda;
int count;
{
    void *data;
    int i, j;
    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;
            }
        }
    }
}
```

Obtaining Values from Fetch Arrays

```
        case CINTTYPE:
            type = "int";
            sprintf(buffer, " %d", *(int *) data);
            break;
        case CLOCATORTYPE:
            type = "byte";
            temp_loc = (loc_t *) (col_ptr->sqldata +
                                   (j * sizeof(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

ESQL/C does not release resources for the **sqllda** structure. When your application no longer needs the **sqllda** structure, it must free all memory that it uses. For more information, see [“Freeing Memory Allocated to an sqllda Structure” on page 17-15](#).

The sample program calls the **free_sqllda()** function to free the memory that the **sqllda** structure uses.

```

/*****
* Function: free_sqllda
* Purpose: Frees memory used by sqllda. This memory includes:
*          o loc_buffer memory (used by TEXT & BYTE)
*          o sqldata memory
*          o sqllda structure
*****/
void free_sqllda(sqllda)
struct sqllda *sqllda;
{
    int i,j, num_to_dealloc;
    struct sqlvar_struct *col_ptr;
    loc_t *temp_loc;

    for (i = 0, col_ptr = sqllda->sqlvar; i < sqllda->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 = (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 sqllda structure */
    free(sqllda);
}

```


Using a System-Descriptor Area

Managing a System-Descriptor Area	16-4
Allocating Memory for a System-Descriptor Area	16-5
Initializing the System-Descriptor Area	16-6
The DESCRIBE Statement and Input Parameters	16-6
The DESCRIBE Statement and Memory Allocation	16-7
Assigning and Obtaining Values from a System-Descriptor Area	16-8
Using the SET DESCRIPTOR Statement	16-9
Using the GET DESCRIPTOR Statement	16-10
Specifying Input Parameter Values	16-10
Putting Column Values into a System-Descriptor Area	16-11
Freeing Memory Allocated to a System-Descriptor Area	16-12
Using a System-Descriptor Area	16-13
Handling an Unknown Select List.	16-13
Executing a SELECT That Returns Multiple Rows.	16-15
Executing a Singleton SELECT	16-22
Handling Unknown Return Values	16-22
Executing a Noncursor Function.	16-23
Executing a Cursor Function	16-28
Handling an Unknown Column List.	16-29
Executing a Simple Insert	16-30
Executing an INSERT That Is Associated with a Cursor	16-35
Handling a Parameterized SELECT Statement	16-36
Executing a Parameterized SELECT That Returns Multiple Rows.	16-37
Executing a Parameterized Singleton SELECT	16-43

Handling a Parameterized User-Defined Routine	16-43
Executing a Parameterized Function	16-44
Executing a Parameterized Procedure	16-45
Handling a Parameterized UPDATE or DELETE Statement	16-46
The dyn_sql Program	16-47
Compiling the Program	16-47
Guide to the dyn_sql.ec File	16-48

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. ♦

The chapter contains the following topics 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 on the fields of the system-descriptor area, see [page 15-5](#).

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 ESQL / C program can manipulate a system-descriptor area with the SQL statements that Figure 16-1 summarizes.

Figure 16-1
SQL Statements That Can Be Used to Manipulate a System-Descriptor Area

SQL Statement	Purpose	See Page...
ALLOCATE DESCRIPTOR	Allocates memory for a system-descriptor area	16-5
DESCRIBE...USING SQL DESCRIPTOR	Initializes the system-descriptor area with information about column-list columns	16-6
GET DESCRIPTOR	Obtains information from the fields of the system-descriptor area	16-8
SET DESCRIPTOR	Places information into a system-descriptor area for the database server to access	16-8
For SELECT and EXECUTE FUNCTION statements that use cursors:		
OPEN...USING SQL DESCRIPTOR	Takes any input parameters from the specified system-descriptor area	16-10
FETCH...USING SQL DESCRIPTOR	Puts the contents of the row into the system-descriptor area	16-11
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	16-11
For non-SELECT statements:		
EXECUTE...USING SQL DESCRIPTOR	Takes any input parameters from the specified system-descriptor area	16-10

(1 of 2)

SQL Statement	Purpose	See Page...
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	16-29
DEALLOCATE DESCRIPTOR	Frees memory allocated for the system-descriptor area when your program is finished with it	16-12

(2 of 2)

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 [Figure 16-1 on page 16-4](#) 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, refer to 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 [Figure 15-3 on page 15-7](#).

- It returns the type of SQL statement prepared.
For more information, see [“Determining the Data Type of a Column” on page 15-19](#).

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 has been 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 16-10](#).

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 **lvvarchar**.

Before you fetch **lvvarchar** 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 **lvvarchar** 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.
- set the item-descriptor fields for each column value for which you provide a description.

```
EXEC SQL set descriptor sysdesc COUNT=:hostvar;
```

```
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 [Figure 15-3 on page 15-7](#).

Set field values to provide values for input parameters in a WHERE clause ([page 16-10](#)) 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 ([page 16-11](#)).

Informix 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 on the data-type constants, see “[Determining the Data Type of a Column](#)” on [page 15-19](#). For more information about how to set individual system-descriptor fields, see the entry for the SET DESCRIPTOR statement in the *Informix Guide to SQL: Syntax*.

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 [Figure 15-3 on page 15-7](#).

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 ([page 16-11](#)).

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 15-19](#).

For more information about how to get individual system-descriptor fields, see the entry for the GET DESCRIPTOR statement in the [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 on how to define input parameters dynamically, see [page 15-25](#).

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 16-13](#).
- 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 16-46](#).
- 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 16-29](#).

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 cursory 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, `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 `ESQL/C` data types that are needed for host variables that hold the return value.

For more information on how to execute `SELECT` statements dynamically, see [“Handling an Unknown Select List” on page 16-13](#). For more information on how to execute user-defined functions dynamically, see [“Handling Unknown Return Values” on page 16-22](#).

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 14-13](#). For more information about `DEALLOCATE DESCRIPTOR`, see the *Informix Guide to SQL: Syntax*.

Using a System-Descriptor Area

Use a system-descriptor area to execute SQL statements that contain unknown values. Figure 16-2 summarizes the types of dynamic statements that the remaining sections of this chapter cover.

Figure 16-2
Using a System-Descriptor Area to Execute Dynamic SQL Statements

Purpose of a System-Descriptor Area	See Page...
Holds select-list column values retrieved by a SELECT statement	16-13
Holds returned values from user-defined functions	16-22
Describes unknown columns in an INSERT statement	16-29
Describes input parameters in the WHERE clause of a SELECT statement	16-36
Describes input parameters in the WHERE clause of a DELETE or UPDATE statement	16-46

Handling an Unknown Select List

For an introduction to how to handle unknown columns in an unknown select list, see [page 15-23](#). 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 14-5](#).
2. Allocate a system-descriptor area with the ALLOCATE DESCRIPTOR statement. For more information, see [“Allocating Memory for a System-Descriptor Area” on page 16-5](#).

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 on DESCRIBE, see [“Initializing the System-Descriptor Area” on page 16-6](#).
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 16-11](#).
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 on GET DESCRIPTOR, see [“Assigning and Obtaining Values from a System-Descriptor Area” on page 16-8](#).
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 16-12](#).



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 16-36](#).

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 ([page 17-18](#)) that uses a system-descriptor area to hold select-list columns. The **demo4** program does *not* include exception handling.

Executing a *SELECT* That Returns Multiple Rows

```
1 #include <stdio.h>

2 EXEC SQL define NAME_LEN          15;

3 main()
4 {
5 EXEC SQL BEGIN DECLARE SECTION;
6   int i;
7   int 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("DEM04 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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

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 [page 16-36](#).)

Executing a SELECT That Returns Multiple Rows

```
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.



Executing a *SELECT* That Returns Multiple Rows

```
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("\nDEM04 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 16-12](#).


```
58 prsysdesc(index)
59 EXEC SQL BEGIN DECLARE SECTION;
60   PARAMETER int index;
61 EXEC SQL END DECLARE SECTION;
62 {
63   EXEC SQL BEGIN DECLARE SECTION;
64     int type;
65     int len;
66     int nullable;
67     char name[40];
68   EXEC SQL END DECLARE SECTION;
69
70   EXEC SQL get descriptor 'demo4desc' VALUE :index
71     :type = TYPE,
72     :len = LENGTH,
73     :nullable = NULLABLE,
74     :name = NAME;
75   printf("    Column %d: type = %d, len = %d, nullable=%d, name = %s\n",
76     index, type, len, nullable, name);
77 }
```

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 16-11](#).

Handling Unknown Return Values

For an introduction on how to handle unknown return values from a user-defined function, see [page 15-26](#). 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 14-5](#).
2. Allocate a system-descriptor area with the ALLOCATE DESCRIPTOR statement.
For more information, see [“Allocating Memory for a System-Descriptor Area” on page 16-5](#).
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 on DESCRIBE, see [“Initializing the System-Descriptor Area” on page 16-6](#).

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 **sqltype.h** header file. For more information, see [“Determining Statement Type” on page 15-16](#).

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 16-12](#).

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.

EXT

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. ♦

SPL

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, 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. Figure 16-3 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 16-3
*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 sqlstyp;
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 strcpy("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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

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 16-5](#).

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 **sqltype.h** file to verify that the EXECUTE FUNCTION statement has been prepared. (For more information, see [“Determining Statement Type” on page 15-16](#).)

```

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 save the number of return values in a host variable, the GET DESCRIPTOR statement obtains the value of the COUNT field. 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 16-48](#).

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 16-12](#).

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 16-11](#).
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 16-8](#).
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 16-12](#).

EXT

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. ♦

SPL

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 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 [page 15-24](#). 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 14-5](#).
2. Allocate a system-descriptor area with the ALLOCATE DESCRIPTOR statement. For more information, see [“Allocating Memory for a System-Descriptor Area” on page 16-5](#).
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 on DESCRIBE, see [“Initializing the System-Descriptor Area” on page 16-6](#).
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 insert a null value, set the appropriate INDICATOR field to -1. For more information on SET DESCRIPTOR, see [“Assigning and Obtaining Values from a System-Descriptor Area” on page 16-8](#).
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 16-12](#).

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         loc_t loc1;
9         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 could be created at runtime. For more information on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

Executing a Simple Insert

```
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 on [page 16-21](#).

```

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_RDONLY;
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 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 ESQL/C **locator.h** header file (line 1) to define the **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 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 8-28](#).

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 ESQL/C **sqltypes.h** header file) to indicate that the column value is stored in an 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.

Executing a Simple Insert

```
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 %ld\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 has been 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 16-12](#).

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.”](#)

Executing an INSERT That Is Associated with a Cursor

Your ESQL/C program must still use the DESCRIBE and SET DESCRIPTOR statements ([page 16-29](#)) 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 ([page 16-13](#)). For more information on how to use an insert cursor, refer to the PUT statement in the [Informix Guide to SQL: Syntax](#).

Handling a Parameterized SELECT Statement

For an introduction on how to determine input parameters, see [page 15-25](#). 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 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 15-25](#).
2. Allocate a system-descriptor area and assign it a name with the ALLOCATE DESCRIPTOR statement. For more information on ALLOCATE DESCRIPTOR, see [“Allocating Memory for a System-Descriptor Area” on page 16-5](#).
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 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 [page 15-19](#).
 - ❑ 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 15-19](#).

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 on SET DESCRIPTOR, see [“Assigning and Obtaining Values from a System-Descriptor Area” on page 16-8](#).

5. 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.

6. Deallocate the system-descriptor area with the DEALLOCATE DESCRIPTOR statement. For more information, see [“Freeing Memory Allocated to a System-Descriptor Area”](#) on page 16-12.



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 16-13.*

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.

Executing a Parameterized SELECT That Returns Multiple Rows

```
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 DEM04 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.

Executing a Parameterized SELECT That Returns Multiple Rows

```
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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

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 on ALLOCATE DESCRIPTOR, see [“Allocating Memory for a System-Descriptor Area” on page 16-5](#).

Executing a Parameterized SELECT That Returns Multiple Rows

```
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.

Executing a Parameterized SELECT That Returns Multiple Rows

```
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 on how to use a system-descriptor area to receive column values, see [page 16-13](#).

Executing a Parameterized SELECT That Returns Multiple Rows

```
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 DEM04 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 16-12](#).

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 16-10](#).

Handling a Parameterized User-Defined Routine

For an introduction on how to determine input parameters, see [page 15-25](#). 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 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 16-43](#).

- 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 16-37](#).

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 16-44](#).

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 ([page 16-36](#)). 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 15-22](#).

The dyn_sql Program

The **dyn_sql.ec** program is an 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 has been 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 on the **esql** preprocessor command, see [“Using the esql Command” on page 1-47](#).

Guide to the dyn_sql.ec File

```
1
2 /*
3   This program prompts the user to enter a SELECT statement
4   for the stores7 database. It processes the statement using dynamic sql
5   and system descriptor areas and displays the rows returned by the
6   database server.
7 */

8 #include <stdio.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];
```

Continued on page [16-50](#)

Lines 8 to 13

These lines specify C and ESQL/C files to include in the program. The **stdio.h** file enables **dyn_sql** to use the standard C I/O library. 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 (**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 11-59](#).

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” on page 16-65](#).)

```
19 EXEC SQL BEGIN DECLARE SECTION;
20     loc_t lcat_descr;
21     loc_t lcat_picture;
22 EXEC SQL END DECLARE SECTION;

23 int whenexp_chk();

24 main(argc, argv)
25 int argc;
26 char *argv[];
27 {
28     long 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         int 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;
```

Continued on page 16-52

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.

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 a **long integer** data type (**ret**) to receive a return value from the **getrow()** function. Line 28 also declares that the **getrow()** function returns a **long integer** data type. Line 37 executes the **WHENEVER** statement to transfer control to **whenexp_chk()** if any errors occur in SQL statements. For more information on the **whenexp_chk()** function, see [“Guide to the exp_chk.ec File” on page 11-59](#).

Lines 30 to 35 define the host variables that are local to the **main()** program block. Lines 45 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     }
```

Continued on page 16-54

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 **sqltypes.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 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 simplelarge-object
107  * data into memory and sets the address of the locator structure
108  * 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 && !strcmp(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 && !strcmp(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  }

```

Continued on page 16-56

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 on how to work with the TEXT and BYTE data types, see [Chapter 8, “Working with Simple Large Objects.”](#)) 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 */
158long getrow(sysdesc)
159EXEC SQL BEGIN DECLARE SECTION;
160 PARAMETER char *sysdesc;
161EXEC SQL END DECLARE SECTION;
162{
163 long 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}
```

Continued on page 16-58

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 on the **getrow()** function, see [“Lines 157 to 167” on page 16-57](#).

Line 152

The **free_stuff()** function frees resources that were allocated when the dynamic SELECT statement was processed. See [“Lines 382 to 388” on page 16-65](#).

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 int col_cnt;
175 EXEC SQL BEGIN DECLARE SECTION;
176 PARAMETER char *sysdesc;
177 EXEC SQL END DECLARE SECTION;
178
179 EXEC SQL BEGIN DECLARE SECTION;
180     int int_data, i;
181     char *char_data;
182     long date_data;
183     datetime dt_data;
184     interval intvl_data;
185     decimal dec_data;
186     short short_data;
187     char name[40];
188     short char_len, type, ind;
189 EXEC SQL END DECLARE SECTION;

190 long size;
191 unsigned amount;
192 int x;
193 char shdesc[81], str[40], *p;

194 printf("\n\n");

195 /* For each column described in the system descriptor area,
196  * determine its data type. Then retrieve the column name and its
197  * value, storing the value in a host variable defined for the
198  * particular data type. If the column is not NULL, display the
199  * name and value.
200  */
201 for(i = 1; i <= col_cnt; i++)
202 {
203     strcpy(statement, "GET DESCRIPTOR: TYPE field");
204     EXEC SQL get descriptor :sysdesc VALUE :i
205         :type = TYPE;
206     switch(type)
207     {
208     case SQLSERIAL:
209     case SQLINT:
210         strcpy(statement,
211             "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
212         EXEC SQL get descriptor :sysdesc VALUE :i
213             :name = NAME,
214             ind = INDICATOR,
215             :int_data = DATA;
216         if(ind == -1)
217             printf("\n%.20s: NULL", name);
```

```

218         else
219             printf("\n%.20s: %d", name, int_data);
220         break;
221     case SQLSMINT:
222         strcpy(statement,
223             "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
224         EXEC SQL get descriptor :sysdesc VALUE :i
225             :name = NAME,
226             :ind = INDICATOR,
227             :short_data = DATA;
228         if(ind == -1)
229             printf("\n%.20s: NULL", name);
230         else
231             printf("\n%.20s: %d", name, short_data);
232         break;
233     case SQLDECIMAL:
234     case SQLMONEY:
235         strcpy(statement,
236             "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
237         EXEC SQL get descriptor :sysdesc VALUE :i
238             :name = NAME,
239             :ind = INDICATOR,
240             :dec_data = DATA;
241         if(ind == -1)
242             printf("\n%.20s: NULL", name);
243         else
244             {
245                 if(type == SQLDECIMAL)
246                     rfmtdec(&dec_data, "###,###,###.##", str);
247                 else
248                     rfmtdec(&dec_data, "$$$,,$$,,$$,,$$", str);
249                 printf("\n%.20s: %s", name, str);
250             }
251         break;
252     case SQLDATE:
253         strcpy(statement,
254             "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
255         EXEC SQL get descriptor :sysdesc VALUE :i
256             :name = NAME,
257             :ind = INDICATOR,
258             :date_data = DATA;
259         if(ind == -1)
260             printf("\n%.20s: NULL", name);
261         else
262             {
263                 if((x = rfmtdate(date_data, "mmm. dd, yyyy",
264                     str)) < 0)
265                     printf("\ndisp_data() - DATE - fmt error");
266                 else
267                     printf("\n%.20s: %s", name, str);
268             }
269         break;

```

```

270     case SQLDTIME:
271         strcpy(statement,
272             "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
273         EXEC SQL get descriptor :sysdesc VALUE :i
274             :name = NAME,
275             :ind = INDICATOR,
276             :dt_data = DATA;
277         if(ind == -1)
278             printf("\n%.20s: NULL", name);
279         else
280             {
281                 x = dttofmtasc(&dt_data, str, sizeof(str), 0);
282                 printf("\n%.20s: %s", name, str);
283             }
284         break;
285     case SQLINTERVAL:
286         strcpy(statement,
287             "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
288         EXEC SQL get descriptor :sysdesc VALUE :i
289             :name = NAME,
290             :ind = INDICATOR,
291             :intvl_data = DATA;
292         if(ind == -1)
293             printf("\n%.20s: NULL", name);
294         else
295             {
296                 if((x = intofmtasc(&intvl_data, str,
297                     sizeof(str),
298                     "%3d days, %2H hours, %2M minutes"))
299                     < 0)
300                     printf("\nINTRVL - fmt error %d", x);
301                 else
302                     printf("\n%.20s: %s", name, str);
303             }
304         break;
305     case SQLVCHAR:
306     case SQLCHAR:
307         strcpy(statement,
308             "GET DESCRIPTOR: LENGTH, NAME fields");
309         EXEC SQL get descriptor :sysdesc VALUE :i
310             :char_len = LENGTH,
311             :name = NAME;
312         amount = char_len;
313         if(char_data = (char *) (malloc(amount + 1)))
314             {
315                 strcpy(statement,
316                     "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
317                 EXEC SQL get descriptor :sysdesc VALUE :i
318                     :char_data = DATA,
319                     :ind = INDICATOR;
320                 if(ind == -1)
321                     printf("\n%.20s: NULL", name);

```



```

322         else
323             printf("\n%.20s: %s", name, char_data);
324     }
325     else
326     {
327         printf("\n%.20s: ", name);
328         printf("Can't display: out of memory");
329     }
330     break;
331 case SQLTEXT:
332     strcpy (statement,
333            "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
334     EXEC SQL get descriptor :sysdesc VALUE :i
335         :name = NAME,
336         :ind = INDICATOR,
337         :lcat_descr = DATA;
338     size = lcat_descr.loc_size; /* get size of data */
339     printf("\n%.20s: ", name);
340     if(ind == -1)
341     {
342         printf("NULL");
343         break;
344     }
345     p = lcat_descr.loc_buffer; /* set p to buf addr */

346     /* print buffer 80 characters at a time */
347     while(size >= 80)
348     {
349         /* mv from buffer to shdesc */
350         ldchar(p, 80, shdesc);
351         printf("\n%80s", shdesc); /* display it */
352         size -= 80; /* decrement length */
353         p += 80; /* bump p by 80 */
354     }
355     strncpy(shdesc, p, size);
356     shdesc[size] = '\0';
357     printf("%-s\n", shdesc); /* dsply last segment */
358     break;
359 case SQLBYTES:
360     strcpy (statement,
361            "GET DESCRIPTOR: NAME, INDICATOR fields");
362     EXEC SQL get descriptor :sysdesc VALUE :i
363         :name = NAME,
364         :ind = INDICATOR;
365     if(ind == -1)
366         printf("%.20s: NULL", name);
367     else
368     {
369         printf("%.20s: ", name);
370         printf("Can't display BYTE type value");
371     }
372     break;

```

```

373         default:
374             printf("\nUnexpected data type: %d", type);
375             EXEC SQL disconnect current;
376             printf("\nDYN_SQL Sample Program over.\n\n");
377             exit(1);
378         }
379     }
380     printf("\n");
381 }

```

Continued on page 16-64

Lines 168 to 381

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 179 to 189), 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 ESQL/C library functions **rfmtdec()**, **rfmtdate()**, **dttofmtasc()**, and **intofmtosc()** to format the **DECIMAL** and **MONEY**, **DATE**, **DATETIME**, and **INTERVAL** data types, respectively, 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:

1. The GET DESCRIPTOR statement (lines 334 and 362) retrieves the locator structure from the system-descriptor area and moves it to the **loc_t** host variable.
2. The **disp_data()** function obtains the address of the data buffer from the locator structure, in **loc_buffer** (lines 345), 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 8-50](#).

```
382 free_stuff()
383 {
    384 EXEC SQL free sel_id; /* free resources for statement */
    385 EXEC SQL free sel_curs; /* free resources for cursor */

    386 /* free system descriptor area */
    387 EXEC SQL deallocate descriptor 'selcat';
    388 }

389 /*
390 * The inpfuns.c file contains the following functions used in this
391 * program:
392 *     more_to_do() - asks the user to enter 'y' or 'n' to indicate
393 *                   whether to run the main program loop again.
394 *
395 *     getans(ans, len) - accepts user input, up to 'len' number of
396 *                       characters and puts it in 'ans'
397 */
398 #include "inpfuns.c"

399 /*
400 * The exp_chk.ec file contains the exception handling functions to
401 * check the SQLSTATE status variable to see if an error has occurred
402 * following an SQL statement. If a warning or an error has
403 * occurred, exp_chk() executes the GET DIAGNOSTICS statement and
404 * displays the detail for each exception that is returned.
405 */
406 EXEC SQL include exp_chk.ec;
```

Lines 382 to 388

The **free_stuff()** function frees resources that were allocated to process the dynamic statement. Line 384 frees resources that were allocated by the application when it prepared the dynamic SELECT statement. Line 385 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 389 to 398

Several of the 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 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 inpfuns.c File” on page 8-68](#).

Lines 399 to 406

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 ESQL/C **include** directive because the exception-handling functions use ESQL/C statements. The **exp_chk.ec** source file is described in [Chapter 11, “Exception Handling.”](#)

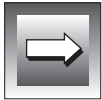


Tip: In a production environment, you would put functions such as **more_to_do()**, **getans()**, and **whenexp_chk()** into a library and include them on the command line when you compile the ESQL/C program.

Using an sqlda Structure

Managing an sqlda Structure	17-4
Defining an sqlda Structure	17-5
Allocating Memory for the sqlda Structure	17-6
Initializing the sqlda Structure	17-7
Allocating Memory for Column Data	17-10
Assigning and Obtaining Values from an sqlda Structure	17-13
Assigning Values	17-13
Obtaining Values	17-13
Specifying Input Parameter Values	17-14
Putting Column Values into an sqlda Structure	17-14
Freeing Memory Allocated to an sqlda Structure	17-15
Using an sqlda Structure	17-16
Handling an Unknown Select List.	17-17
Executing a SELECT That Returns Multiple Rows.	17-18
Executing a Singleton SELECT	17-26
Handling Unknown Return Values	17-26
Executing a Noncursor Function.	17-28
Executing a Cursor Function	17-28
Handling an Unknown Column List.	17-30
Executing a Simple Insert	17-31
Executing an INSERT That Is Associated with a Cursor	17-31
Handling a Parameterized SELECT Statement	17-32
Executing a Parameterized SELECT That Returns Multiple Rows.	17-34
Executing a Parameterized Singleton SELECT	17-41

Handling a Parameterized User-Defined Routine	17-41
Executing a Parameterized Function	17-42
Executing a Parameterized Procedure	17-42
Handling a Parameterized UPDATE or DELETE Statement	17-43



An **sqllda** 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 **sqllda.h** header file.

Important: *The **sqllda** structure does not conform to the X/Open standards. It is an Informix extension to ESQL/C.*

This chapter describes the following topics about how to use an **sqllda** structure:

- Using an **sqllda** structure to hold unknown values
- Managing an **sqllda** structure
- Using an **sqllda** structure to handle unknown values in dynamic SQL statements

For information on the fields of the **sqllda** structure, see [page 15-10](#).

Managing an sqlda Structure

Your ESQL/C program can manipulate an **sqlda** structure with the SQL statements that Figure 17-1 summarizes.

Figure 17-1
SQL Statements That Can Be Used to Manipulate an sqlda Structure

SQL Statement	Purpose	See Page...
DESCRIBE...INTO	Allocates an sqlda structure and initializes this structure with information about column-list columns	17-6 17-7
For SELECT and EXECUTE FUNCTION statements that use cursors:		
OPEN...USING DESCRIPTOR	Takes any input parameters from the specified sqlda structure	17-14
FETCH...USING DESCRIPTOR	Puts the contents of the row into the sqlda structure	17-14
For SELECT and EXECUTE FUNCTION statements that return <i>only one row</i> :		
EXECUTE...INTO DESCRIPTOR	Puts the contents of the singleton row into the sqlda structure	17-14
For non-SELECT statements:		
EXECUTE...USING DESCRIPTOR	Takes any input parameters from the specified sqlda structure	17-14
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	17-30

In addition, your 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 ESQL/C **sqlda.h** header file defines the **sqlda** structure. To define an **sqlda** structure, the ESQL/C program must take the following actions:

- Include the **sqlda.h** header file to provide the declaration for **sqlda** in your program

The ESQL/C preprocessor automatically includes the **sqlhdr.h** file, which includes the **sqlda.h** header file. You can explicitly include **sqlda.h** to improve the readability of your programs, as follows:

```
EXEC SQL include sqlda.h;
```

- 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 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 has been 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 17-7](#).

- 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 17-10](#).

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 within it. (See [page 17-14](#).)

Initializing the sqlda Structure

To send or receive column values in the database, your 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 [page 17-6](#)), 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 17-6](#).
- 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, MULTISSET, 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 [Figure 15-7 on page 15-12](#).



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 17-10](#).

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 has been 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 Statement Type” on page 15-16](#).

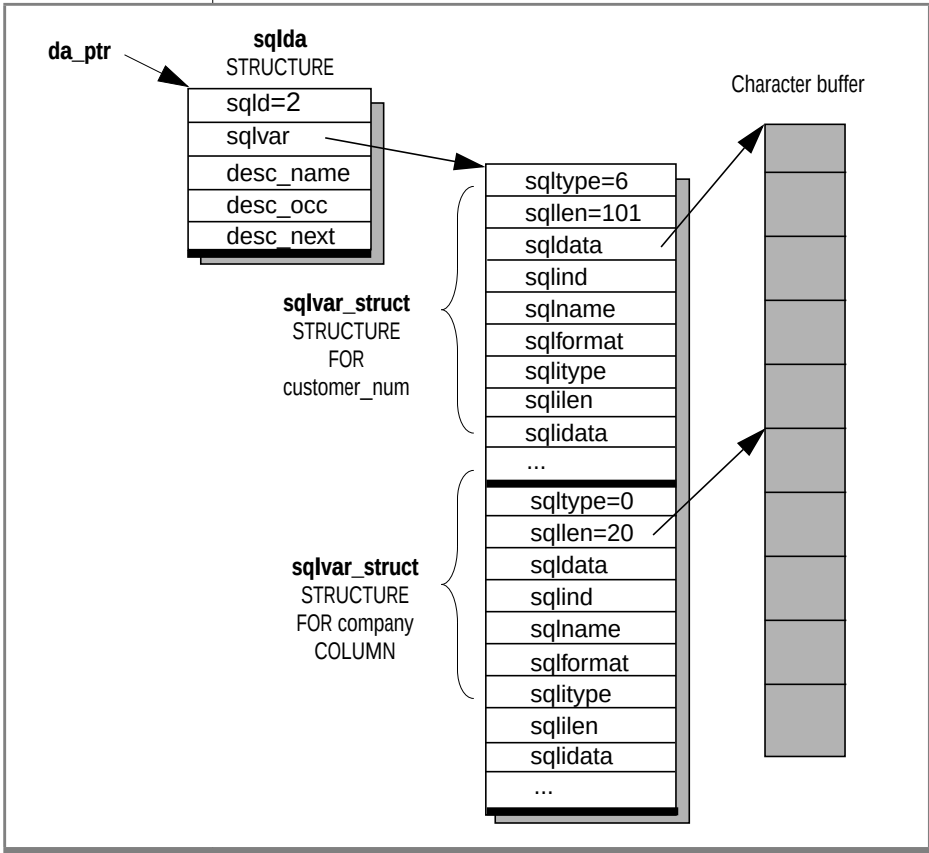
The following DESCRIBE statement also allocates memory for an **sqlda** structure and for two **sqlvar_struct** data structures (one for the **customer_num** column and another for the **company** column) and then initializes the pointer **da_ptr->sqlvar** with the initial address of the memory that 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 17-2 shows a sample **sqlda** structure that this DESCRIBE...INTO statement might initialize.

Figure 17-2
Sample sqlda
Structure for Two
Columns



If some other SQL statement has been 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 [page 17-14](#).)

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 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 17-3 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->sqlld;
    cnt++, col_ptr++)
{
    pos = rtypalign(pos, col_ptr->sqltype);
    col_ptr->sqldata = &data_buf[pos];
    size = rtypmsize(col_ptr->sqltype, col_ptr->sqlllen);
    pos += size;
}
```

Figure 17-3
*Allocating Column
Data from a Static
Character Buffer*

You could replace the code fragment in Figure 17-3 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 ESQL/C or SQL data types defined in the **sqltypes.h** header file. (See [page 15-19](#).) 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, 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 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 ESQL/C or SQL data type.

This function accepts two arguments: the integer ESQL/C or SQL data type (in **sqltype**) and the length (in **sqllen**) for each column value.

These ESQL/C library functions are described in more detail in [Chapter 2, “INFORMIX-ESQL/C Data Types.”](#)

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 **datetime.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** ([page 17-18](#)) and **unload.ec** demonstration programs that are supplied with ESQL/C.

Assigning and Obtaining Values from an `sqllda` Structure

When you use the **`sqllda`** 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 **`sqllda`** 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 **`sqllda`** fields to provide values for input parameters in a WHERE clause ([page 17-14](#)) or to modify the contents of a field after you use the DESCRIBE...INTO statement to fill the **`sqllda`** structure ([page 17-10](#)).

Obtaining Values

To obtain values from the **`sqllda`** 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 **`sqllda`** 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 **`sqllda`** structure into a host variable ([page 17-14](#)).

The data type of the host variable must be compatible with the type of the associated field in the **`sqllda`** 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 15-19](#).

Specifying Input Parameter Values

Since the DESCRIBE...INTO statement does not analyze the WHERE clause, your program must explicitly allocate an **sqllda** structure and the **sqlvar_struct** structures (see [page 17-6](#)). To describe the input parameters you must determine the number of input parameters and their data types and store this information in the allocated **sqllda** structure. For general information on how to define input parameters dynamically, see [page 15-25](#).

When you execute a parameterized statement, you must include the USING DESCRIPTOR clause to specify the **sqllda** 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 17-32](#).
- 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 17-43](#).
- 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 17-30](#).

Putting Column Values into an sqllda 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 **sqllda** 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 cursory 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 `sqllda` 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, `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 `sqllda` structure.*

Once the column values are in the `sqllda` structure, you can transfer the values from the `sqldata` fields to the appropriate host variables. You must use the `sqlllen` 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 `ESQL/C` data types that are needed for host variables that hold the returned value.

For more information on how to execute `SELECT` statements dynamically, see [“Handling an Unknown Select List” on page 17-17](#). For more information on how to execute a user-defined function dynamically, see [page 17-26](#).

Freeing Memory Allocated to an `sqllda` Structure

Once you finish with an `sqllda` 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.

To release the memory that the `sqllda` structure occupies, use the standard C library `free()` function, as shown in the following example:

```
free(sqllda_ptr);
```

If your 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 17-4 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;
:
free(sqldaptr1);
free(sqldaptr2);
free(sqldaptr3);
```

Figure 17-4
*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 17-10](#)), you must also deallocate the memory allocated to the **sqldata** fields. For information on how to free other program resources, see [“Freeing Resources” on page 14-13](#).

Using an sqlda Structure

Use an SQL descriptor-area (**sqlda**) structure to execute SQL statements that contain unknown values. Figure 17-5 summarizes the types of dynamic statements that the remaining sections of this chapter cover.

Figure 17-5
Using an sqlda Structure to Execute Dynamic SQL Statements

Purpose of the sqlda Structure	See Page...
Holds select-list column values retrieved by a SELECT	17-17
Holds returned values from user-defined functions	17-26
Describes unknown columns in an INSERT	17-30
Describes input parameters in the WHERE clause of a SELECT	17-32
Describes input parameters in the WHERE clause of a DELETE or UPDATE	17-43

Handling an Unknown Select List

For an introduction on how to handle unknown columns in an unknown select list, see [page 15-23](#). This section describes how to use an **sqllda** structure to handle a SELECT statement.

To use an **sqllda** structure to handle unknown select-list columns

1. Declare a variable to hold the address of an **sqllda** structure. For more information, see [“Defining an sqllda Structure” on page 17-5](#).
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 14-5](#).
3. Use the DESCRIBE...INTO statement to perform two tasks:
 - a. Allocate an **sqllda** structure. The address of the allocated structure is stored in the **sqllda** pointer that you declare. For more information, see [“Allocating Memory for the sqllda Structure” on page 17-6](#).
 - 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 sqllda Structure” on page 17-7](#).
4. Examine the **sqltype** and **sqllen** fields of **sqllda** 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 17-10](#).
5. Save the number of select-list columns stored in the **sqlc** 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 **sqllda** structure. See [“Putting Column Values into an sqllda Structure” on page 17-14](#).



7. Retrieve the row data from the **sqllda** 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 sqllda Structure”](#) on page 17-13.
8. Release memory allocated to the **sqldata** fields and the **sqllda** structure. For more information, see [“Freeing Memory Allocated to an sqllda Structure”](#) on page 17-15.

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 **sqllda** structure. For more information, see [“Handling a Parameterized SELECT Statement”](#) on page 17-32.

The following sections demonstrate how to use an **sqllda** 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 **sqllda** structure.

The **demo4** sample program ([page 16-15](#)) assumes these same conditions. While **demo4** uses a system-descriptor area to define the select-list columns, **demo3** uses an **sqllda** 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     short int i, desc_count;
12     char demoquery[80];
13 EXEC SQL END DECLARE SECTION;

14     printf("DEM03 Sample ESQL program running.\n\n");
15     EXEC SQL connect to 'stores7';
```

Line 2

The program must include the 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 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.

Executing a SELECT That Returns Multiple Rows

```
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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

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 **sqllda** structure to which **demo3_ptr** points as the location for these column descriptions. The DESCRIBE...INTO statement also allocates memory for an **sqllda** 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 on 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 on the **lname** column of the **customer** table.

Executing a *SELECT* That Returns Multiple Rows

```
27 desc_count = demo3_ptr->sqlc;
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->sqlllen += 1;
40     /* Get next word boundary for column data and
41        assign buffer position to sqldata */
42     pos = 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->sqlllen);
47     pos += size;
48 }

49 EXEC SQL open demo3cursor;

```

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 **sqlllen** 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 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 ESQL/C **rtypmsize()** function (line 46) returns the number of bytes required for the SQL data type that is specified by the **sqltype** and **sqlllen** 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 17-10](#).

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 [page 17-30](#)).

Executing a SELECT That Returns Multiple Rows

```
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 }
61 if (strncmp(SQLSTATE, "02", 2) != 0)
62     printf("SQLSTATE after fetch is %s\n", SQLSTATE);

63 EXEC SQL close demo3cursor;
```

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 **sqllda** structure to which **demo3_ptr** points as the location of the column values. After this **FETCH**, the column values are stored in the specified **sqllda** 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.

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("\nDEM03 Sample Program Over.\n\n");
74 }

75 prsqllda(index, sp)
76 short int 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 17-15](#).

The **free()** system call (line 71) frees the **sqlda** structure to which **demo3_ptr** points.

Lines 75 to 81

The **prsqllda()** 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 ESQL/C demonstration programs **unload.ec** and **dyn_sql.ec** (described on [page 16-47](#)) also use **sqlda** to describe columns of a select list. Also refer to the PREPARE statement in the “[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 17-14](#).

Handling Unknown Return Values

For an introduction on how to handle unknown return values from a user-defined function, see [page 15-26](#). 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 17-5](#).
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 14-5](#).

3. Use the DESCRIBE...INTO statement to perform two tasks:
 - a. Allocate an **sqllda** structure. The address of the allocated structure is stored in the **sqllda** pointer that you declare. For more information, see [“Allocating Memory for the sqllda Structure” on page 17-6](#).
 - 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 sqllda Structure” on page 17-7](#).
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 **sqltype.h** header file. For more information, see [“Determining Statement Type” on page 15-16](#).
5. Examine the **sqltype** and **sqllen** fields of **sqllda** 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 17-10](#).
6. Execute the EXECUTE FUNCTION statement and store the return values in the **sqllda** 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 **sqllda** structure.

See [“Freeing Memory Allocated to an sqllda Structure” on page 17-15](#).

EXT

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. ♦

SPL

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, 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 **sqllda** structure.

For more information, see [“Putting Column Values into an sqllda Structure” on page 17-14.](#)

4. Retrieve the row data from the **sqllda** 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 sqllda Structure” on page 17-13.](#)

5. Release memory allocated to the **sqldata** fields and the **sqllda** structure.

For more information, see [“Freeing Memory Allocated to an sqllda Structure” on page 17-15.](#)

EXT

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 **sqllda** structure contains only one **sqlvar_struct** structure with the single return value. ♦

SPL

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 **sqllda** 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 [page 15-24](#). 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 17-5](#).
2. Prepare the INSERT statement (with the PREPARE statement) and give it a statement identifier. See [“Assembling and Preparing the SQL Statement” on page 14-5](#).
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 17-6](#).
 - 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 17-7](#).
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 17-10](#).
5. Save the number of columns stored in the **sqlc** 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 **sqllda** structure. For more information, see [“Freeing Memory Allocated to an sqllda Structure” on page 17-15](#).

Executing a Simple Insert

The following steps outline how to execute a simple INSERT statement with an **sqllda** 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 **sqllda**.
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 ([page 17-26](#)). 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 **sqllda** 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 ([page 17-18](#)). For more information on how to use an insert cursor, refer to the PUT statement in the *Informix Guide to SQL: Syntax*.

Handling a Parameterized SELECT Statement

For an introduction on how to determine input parameters, see [page 15-25](#). 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 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 17-5](#).
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 15-25](#).
3. Allocate an **sqlda** structure with a system memory-allocation function such as **malloc()**. For more information, see [“Specifying Input Parameter Values” on page 17-14](#) and [“Allocating Memory for the sqlda Structure” on page 17-6](#).
4. Indicate the number of input parameters in the WHERE clause with C-language statements that set the **sqlid** 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 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 15-19](#).
 - b. For a CHAR or VARCHAR value, **sqlen** 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 **sqlen** 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 17-10](#).

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 17-13](#).

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 17-15](#).



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 17-17](#).

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 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 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 on these steps, see [“Assembling and Preparing the SQL Statement” on page 14-5](#).

Line 21

This line declares the **slct_cursor** cursor for the prepared statement identifier, **sql_id**.

Executing a Parameterized *SELECT* That Returns Multiple Rows

```
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” on page 17-39](#).

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 on **selectListMem()**, see [“Lines 85 to 102” on page 17-41](#).

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 on how to handle unknown select-list columns, see [page 17-17](#).

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.

Executing a Parameterized *SELECT* That Returns Multiple Rows

```
37     for(i=0; i<sqlda_ptr->sqlld; 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” on page 17-25](#)) 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 17-15](#).

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.

Executing a Parameterized *SELECT* That Returns Multiple Rows

```
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", descsp->sqld);

93     for(col_ptr=descp->sqlvar, cnt=pos=0; cnt < descsp->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 **sqllda** structure for the unknown select-list columns of the parameterized SELECT statement. For more information on how to use an **sqllda** structure for select-list columns, see [“Handling a Parameterized SELECT Statement” on page 17-32](#).

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 **sqllda** structure. For more information, see [“Specifying Input Parameter Values” on page 17-14](#).

Handling a Parameterized User-Defined Routine

For an introduction on how to determine input parameters, see [page 15-25](#). This section describes how to handle a parameterized user-defined routine with an **sqllda** 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 ESQL/C program can use an **sqllda** 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 **sqllda** structure and to execute the function. For more information, see [“Executing a Parameterized Singleton SELECT” on page 17-41](#).

- 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 **sqllda** structure. For more information, see [“Executing a Parameterized SELECT That Returns Multiple Rows” on page 17-34](#).

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 **sqllda** 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 ([page 17-32](#)). 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 **sqllda** 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 15-22](#).

List of ESQL/C Library Functions

A

All the library functions provided with INFORMIX-ESQL/C appear alphabetically in the following table.

Function Name	Description	Page
bympr()	Compares two groups of contiguous bytes	3-21
bycopy()	Copies bytes from one area to another	3-24
byfill()	Fills the specified area with a character	3-26
byleng()	Counts the number of bytes in a string	3-28
decadd()	Adds two decimal numbers	4-9
deccmp()	Compares two decimal numbers	4-11
deccopy()	Copies a decimal number	4-13
deccvasc ()	Converts a C char type to a decimal type	4-15
deccvdbl()	Converts a C double type to a decimal type	4-18
deccvint()	Converts a C int type to a decimal type	4-20
deccvlong()	Converts a C long type to a decimal type	4-22
decdiv()	Divides two decimal numbers	4-24
dececv()	Converts a decimal value to an ASCII string	4-26
decfcvt()	Converts a decimal value to an ASCII string	4-26
decmul()	Multiplies two decimal numbers	4-32
decround()	Rounds a decimal number	4-34
decsub()	Subtracts two decimal numbers	4-37
dectoasc()	Converts a decimal type to an ASCII string	4-39
dectodbl()	Converts a decimal type to a C double type	4-42
dectoint()	Converts a decimal type to a C int type	4-44
dectolong()	Converts a decimal type to a C long type	4-47
detrunc()	Truncates a decimal number	4-49
dtaddinv()	Adds an interval value to a datetime value	6-40

(1 of 8)

Function Name	Description	Page
dtcurrent()	Gets current date and time	6-42
dtcvasc()	Converts an ANSI-compliant character string to a datetime	6-44
dtcvfmtasc()	Converts a character string to a datetime value	6-47
dtextend()	Changes the qualifier of a datetime	6-51
dtsub()	Subtracts one datetime value from another	6-54
dtsubinv()	Subtracts an interval value from a datetime value	6-57
dttoasc()	Converts a datetime value to an ANSI-compliant character string	6-59
dttofmtasc()	Converts a datetime value to a character string	6-62
ifx_cl_card()	Returns the cardinality of the specified collection type host variable	9-45
ifx_dececvt()	Converts a decimal value to an ASCII string (thread-safe version)	13-31
ifx_decfcvt()	Converts a decimal value to an ASCII string (thread-safe version)	13-31
ifx_getcur_conn_name()	Returns the name of the current connection	12-47
ifx_getserial8()	Returns an inserted SERIAL8 value	5-10
ifx_int8add()	Adds two int8 numbers	5-12
ifx_int8cmp()	Compares two int8 numbers	5-13
ifx_int8copy()	Copies a int8 number	5-14
ifx_int8cvasc()	Converts a C char type value to a int8 type value	5-15
ifx_int8cvdbl()	Converts a C double type value to a int8 type value	5-16

(2 of 8)

Function Name	Description	Page
<code>ifx_int8cvtint()</code>	Converts a C int type value to a int8 type value	5-18
<code>ifx_int8cvlong()</code>	Converts a C long type value to a int8 type value	5-19
<code>ifx_int8div()</code>	Divides two int8 numbers	5-20
<code>ifx_int8mul()</code>	Multiplies two int8 numbers	5-21
<code>ifx_int8sub()</code>	Subtracts two int8 numbers	5-22
<code>ifx_int8toasc()</code>	Converts a int8 type value to a text string	5-23
<code>ifx_int8todbl()</code>	Converts a int8 type value to a C double type value	5-24
<code>ifx_int8toint()</code>	Converts a int8 type value to a C int type value	5-26
<code>ifx_int8tolong()</code>	Converts a int8 type value to a C long type value	5-27
<code>ifx_lo_alter()</code>	Alters the storage characteristics of an existing smart large object	7-36
<code>ifx_lo_close()</code>	Closes an open smart large object	7-38
<code>ifx_lo_col_info()</code>	Obtains column-level storage characteristics into an LO-specification structure	7-39
<code>ifx_lo_copy_to_file()</code>	Copies a smart large object into an operating-system file	7-41
<code>ifx_lo_copy_to_lo()</code>	Copies an operating-system file into an open smart large object	7-44
<code>ifx_lo_create()</code>	Creates an LO descriptor for a smart large object	7-46
<code>ifx_lo_def_create_spec()</code>	Allocates an LO-specification structure and initializes its fields to null values	7-49
<code>ifx_lo_filename()</code>	Returns the generated file name, given an LO descriptor and a file specification	7-50

(3 of 8)

Function Name	Description	Page
<code>ifx_lo_from_buffer()</code>	Copies bytes from a user-defined buffer into smart large object	7-52
<code>ifx_lo_open()</code>	Opens an existing smart large object	7-53
<code>ifx_lo_read()</code>	Reads a specified number of bytes from an open smart large object	7-55
<code>ifx_lo_readwithseek()</code>	Seeks to a specified position in an open smart large object and reads a specified number of bytes	7-57
<code>ifx_lo_release()</code>	Releases resources associated with a temporary smart large object	7-59
<code>ifx_lo_seek()</code>	Sets the seek position for the next read or write on an open smart large object	7-61
<code>ifx_lo_spec_free()</code>	Frees the resources allocated to an LO-specification structure	7-63
<code>ifx_lo_specget_estbytes()</code>	Gets the estimated number of bytes from the LO-specification structure	7-64
<code>ifx_lo_specget_extsz()</code>	Gets the allocation extent size from the LO-specification structure	7-66
<code>ifx_lo_specget_flags()</code>	Gets the create-time flags from the LO-specification structure	7-67
<code>ifx_lo_specget_maxbytes()</code>	Gets the maximum number of bytes from the LO-specification structure	7-69
<code>ifx_lo_specget_sbospace()</code>	Gets the name of the sbospace from the LO-specification structure	7-71
<code>ifx_lo_specset_estbytes()</code>	Sets the estimated number of bytes from the LO-specification structure	7-73
<code>ifx_lo_specset_extsz()</code>	Sets the allocation extent size in the LO-specification structure	7-75
<code>ifx_lo_specset_flags()</code>	Sets the create-time flags in the LO-specification structure	7-77

(4 of 8)

Function Name	Description	Page
<code>ifx_lo_specset_maxbytes()</code>	Sets the maximum number of bytes in the LO-specification structure	7-79
<code>ifx_lo_specset_sbospace()</code>	Sets the name of the sbospace in the LO-specification structure	7-80
<code>ifx_lo_stat()</code>	Returns status information about an open smart large object	7-81
<code>ifx_lo_stat_atime()</code>	Returns the last access time for a smart large object	7-83
<code>ifx_lo_stat_cspect()</code>	Returns the storage characteristics into the LO-specification structure for a specified smart large object	7-84
<code>ifx_lo_stat_ctime()</code>	Returns the last change-in-status time for the smart large object	7-86
<code>ifx_lo_stat_free()</code>	Frees the resources allocated to an LO-status structure	7-87
<code>ifx_lo_stat_mtime_sec()</code>	Returns the last modification time, in seconds, for the smart large object	7-88
<code>ifx_lo_stat_refcnt()</code>	Returns the reference count for the smart large object	7-89
<code>ifx_lo_stat_size()</code>	Returns the size of the smart large object	7-91
<code>ifx_lo_tell()</code>	Returns the current seek position of an open smart large object	7-92
<code>ifx_lo_to_buffer</code>	Copies bytes from smart large object into a user-defined buffer	7-93
<code>ifx_lo_truncate()</code>	Truncates a smart large object to a specific offset	7-95
<code>ifx_lo_write()</code>	Writes a specified number of bytes to an open smart large object	7-96
<code>ifx_lo_writewithseek()</code>	Seeks to a specified position in an open smart large object and writes a specified number of bytes	7-98

(5 of 8)

Function Name	Description	Page
incvasc()	Converts an ANSI-compliant character string to an interval VALUE	6-65
incvfmtasc()	Converts a character string to an interval value	6-68
intoasc()	Converts an interval value to an ANSI-compliant character string	6-71
intofmtasc()	Converts an interval value to a string	6-74
invdivdbl()	Divides an interval value by a numeric value	6-77
invdivinv()	Divides an interval value by an interval value	6-80
invextend()	Copies an interval value under a different qualifier	6-82
invmuldbl()	Multiplies an interval value by a numeric value	6-85
ldchar()	Copies a fixed-length string to a null-terminated string	3-30
rdatestr()	Converts an internal format to string	6-6
rdayofweek()	Returns the day of the week	6-8
rdefmtdate()	Converts a string to an internal format	6-10
rdownshift()	Converts all letters to lowercase	3-32
rfmtdate()	Converts an internal format to a string	6-14
rfmtdec()	Converts a decimal type to a formatted string	5-35
rfmtdouble()	Converts a double type to a string	5-38
rfmtlong()	Converts a long integer to a formatted string	5-41
rgetlmsg()	Retrieves the error message for a large error number	11-48

(6 of 8)

Function Name	Description	Page
rgetmsg()	Retrieves the error message for an error number	11-51
risnull()	Checks whether a C variable is null	2-24
rjulmdy()	Returns month, day, and year from an internal format	6-17
rleapyear()	Determines whether a specified year is a leap year	6-19
rmidyjul()	Returns an internal format from month, day, and year	6-21
rsetnull()	Sets a C variable to null	2-27
rstod()	Converts a string to a double type	3-34
rstoi()	Converts a null-terminated string to a short integer	3-36
rstol()	Converts a string to a long integer	3-38
rstrdate()	Converts a string to an internal format	6-23
rtoday()	Returns a system date in internal format	6-26
rtypalign()	Aligns data on a proper type boundary	2-30
rtypmsize()	Gives byte size of SQL data types	2-33
rtypname()	Returns the name of a specified SQL type	2-36
rtypwidth()	Gives minimum conversion byte size	2-39
rupshift()	Converts all letters to uppercase	3-40
sqgetdbcs()	Returns the names of the databases that a database server can access	12-48
sqlbreak()	Sends the database server a request to stop processing	12-51
sqlbreakcallback()	Provides a method of returning control to the application while it is waiting for the database server to process an SQL request	12-53

(7 of 8)

Function Name	Description	Page
sqldetach()	Detaches child process from parent process	12-56
sqldone()	Determines whether the database server is currently processing an SQL request	12-63
sqlexit()	Terminates a database server process	12-64
sqlsignal()	Performs signal handling and child-processes cleanup	12-65
sqlstart()	Starts a database server process	12-67
stcat()	Concatenates one string to another	3-42
stchar()	Copies a null-terminated string to a fixed-length string	3-42
stcmp()	Compares two strings	3-46
stcopy()	Copies one string to another string	3-48
stleng()	Counts the number of bytes in a string	3-50

(8 of 8)

The Demonstration Database

Your Informix software includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. Many of the examples in this manual are based on the **stores7** database. This appendix describes how to create it.

WIN NT/95

On Windows NT and Windows 95 systems, you must create the **stores7** demonstration database using the **dbaccessdemo7** script. For more information on using **dbaccessdemo7** to create the **stores7** database, refer to the [DB-Access User Manual](#). ♦

UNIX

On UNIX systems, Informix also includes source files for the INFORMIX-ESQL/C demonstration programs that access and manipulate the data stored in **stores7**. Informix also includes source files for many of the examples in this manual.

Preparing to Create the Demonstration Database

To create or use the demonstration database, you or your database administrator must configure the environment, as follows:

- Set the **INFORMIXDIR** environment variable so that it contains the name of the directory in which your Informix products are installed.
- Set the **INFORMIXSERVER** environment variable to the name of the default database server.

The name of the default database server must exist in the **sqlhosts** file or the file that the **INFORMIXSQLHOSTS** environment variable specifies. For a full description of environment variables, see the [Informix Guide to SQL: Reference](#). For information about **sqlhosts**, see the [Administrator's Guide](#) for your database server.

- To use the relative pathnames as described in the following sections, set your **PATH** environment variable to include **\$INFORMIXDIR/bin**.

Creating the Demonstration Database

The script that you use to install the demonstration database is called **esqldemo7** and is located in the **\$INFORMIXDIR/bin** directory on a UNIX server.

If you choose to name your database, you must use the following naming conventions:

- A name can have a minimum of 1 character and a maximum of 18 characters.
- The first character of a name must be a letter; other characters can be letters, numbers, or underscores (_).
- The name must be unique among the database names on the same database server. (SQL makes no distinction between uppercase and lowercase letters; therefore **stores7** and **STORES7** refer to the same database.)

For example, you might want to name the demonstration database **mystores**.

Important: *If you create the demonstration database with a name other than **stores7**, you must modify the example programs to access the database you create.*



To create and populate the stores7 demonstration database

1. Create a new directory for the demonstration programs by entering the following command:

```
mkdir dirname
```

You can create this directory wherever you want the demonstration programs to be. The *dirname* can be any name that you choose. For example, you might want to name the directory **demo_db**.

2. Make the new directory the current directory by entering the following command:

```
cd dirname
```

3. Create the demonstration database by entering any one of the following commands:

- To create the database and specify a name that you choose, enter:

```
esqldemo7 dbname
```

If you do not specify a database name, the database name defaults to **stores7**.

- To create the database with logging, enter:

```
esqldemo7 -log dbname
```

If you do not specify the **-log** option, your demonstration database will not support transaction logging.

- To specify a dbspace for the database, enter:

```
esqldemo7 dbname -dbspace dbspacename
```

Make sure that the dbspace that you specify exists. If it does not exist, you can create the dbspace with the **onspaces** utility. The **onspaces** utility is described in the [Administrator's Guide](#) for your database server.

If you do not specify a dbspace name, by default, the data for the database is put into the root dbspace.

- To specify all of the options in one command, enter:

```
esqldemo7 -log dbname -dbspace dbspacename
```

After you enter one of the previous commands, a series of messages appears on the screen as each table in the database is created. If you use the default database name, the following messages appear:

```
Dropping existing stores7 database ...
Creating stores7 database ...
:
The creation of the demonstration database is now
complete. The remainder of this script copies the
examples into your current directory.
Press "Y" to continue, or "N" to abort.
```

4. Press Y to copy the demonstration programs into the directory that you have created or press N to specify that the demonstration programs should not be copied.

You can now use the demonstration database and demonstration programs.

If you make changes to your database that you do not want to keep, you can run the **esqldemo7** script to create a fresh demonstration database. When the script asks if you would like to copy the demonstration programs to the current directory, press N if you have made changes to the demonstration programs and do not want them replaced with the original versions. ♦

Index

A

- .a file extension 1-76
- Access mode flags, locks on smart large objects 7-27
- Aggregate function 1-42, 1-43, 11-15, 11-31
- ALLOCATE COLLECTION statement 9-12
- ALLOCATE DESCRIPTOR statement 16-4, 16-5
- ALLOCATE ROW statement 9-57
- Allocating memory
 - for fetch arrays 15-37
- Allocation extent size 7-9, 7-66, 7-75
- ALTER INDEX statement 12-33, 12-51, 15-17
- ALTER TABLE statement 12-33, 12-51, 15-16
- Ampersand (&) symbol 5-28
- ANSI C standards 13-25
- ANSI compliance, level Intro-15
- ansi preprocessor option 1-51, 1-55, 12-17
 - See also* ANSI SQL standards.
- ANSI SQL standards
 - checking for Informix extensions 11-15, 11-23, 11-31, 12-17
 - connecting to a database 12-17
 - declaring host variables 1-27
 - defining ESQL/C function prototypes 1-23
 - delimiting identifiers 1-32
 - delimiting strings 1-32
 - escape character 1-19
 - for datetime and interval values 6-36, 6-59, 6-71
 - getting diagnostic information 11-6
 - preparing SQL statements 14-5
 - specifying host variables 1-21
 - SQLSTATE class values 11-14
 - using EXEC SQL keywords 1-13
 - using GOTO in
 - WHENEVER 11-46
 - using INDICATOR keyword 1-45, 1-46
 - warning values 11-22
- See also* ansi preprocessor option; ANSI-compliant database.
- ANSI-compliant database
 - determining 11-15, 11-22, 11-30, 12-26
 - indicating NOT FOUND condition 11-20, 11-35
 - inserting character data 3-18
 - nonstandard syntax in
 - sqlwarn 11-31
- ANSI-style parameters as host variables 1-40
- Arithmetic operations
 - datetime data 6-36
 - decimal data 4-7
 - description of 2-19
 - int8 data 5-8
 - interval data 6-36
- Array
 - and truncated SQL value 1-44
 - dimension limit 1-36
 - in a host-variable typedef 1-38
 - of host variables 1-36

ASKPASSATCONNECT network parameter 1-107, 12-13, 12-14

Asterisk (*) symbol

- as formatting character 5-28
- as overflow character 2-17, 6-35

AUTOFREE feature 14-24

- enabling 14-25
- for a particular cursor 14-25
- with Deferred PREPARE and OPTOFC features 14-39

B

Backslash (\) character 1-19

- bc processor option 1-78

Binary large object (blob). *See* Simple large-object data type.

BLOB data type

- corresponding ESQL/C data type 2-5, 2-11
- declaring host variable for 7-7
- implementation of 10-48
- on optical disc 7-32
- role of locator.h 2-11

See also ifx_lo_t data type; Smart-large-object data type.

BOOLEAN data type

- corresponding ESQL/C data type 2-5, 2-13
- data conversion 2-14, 2-18
- defined constant 2-9
- distinct-bit constant 15-31
- distinct-bit macro 15-31

See also boolean data type.

boolean data type

- corresponding SQL data type 2-13
- declaration 2-13
- defined constant 2-11

See also BOOLEAN data type.

Borland compiler, invoking 1-78

Buffer. *See* Cursor buffer.

Build file 1-82

Built-in data type

- as element type of collection 9-6
- as field type in row 9-50

bycmp() library function 3-21

bycopy() library function 3-24

byfill() library function 3-26

byleng() library function 3-28

BYTE data type

- corresponding ESQL/C data type 2-4, 2-11
- declaring host variable for 8-5
- defined constant 2-8
- inserting 8-17, 8-23, 8-28, 8-34
- locator structure shown 8-7
- on optical disc 8-40
- role of locator.h 1-23, 2-11
- selecting 8-15, 8-22, 8-27, 8-33
- subscripting 8-6

See also loc_t data type; Simple large-object data type.

Bytes

- comparing 3-21
- copying 3-24
- determining number of 3-28
- filling with a character 3-26

C

C compiler

- ANSI C 1-25, 1-26
- bc option 1-78
- Borland 1-78
- c option 1-78, 1-83
- called by esql 1-11
- generating thread-safe code 13-27
- #include preprocessor statement 1-26
- initializer expressions 1-29
- linking in other files 1-86
- mc option 1-79
- Microsoft 1-79
- naming restrictions 1-28
- options invoked implicitly 1-82
- passing arguments to 1-75, 1-78, 1-80, 1-90
- passing options to 1-76
- passing source files to 1-78
- role in compiling ESQL/C programs 1-10
- specifying 1-77, 1-80
- #define preprocessor statement 1-15
- #include preprocessor statement 1-14, 1-62

- .c file extension 1-9, 1-58, 1-76, 1-78, 1-80, 1-86, 11-33, 13-15

C header files

- for conditional compilation of ESQL/C programs 1-66
- for defining host variables 1-66
- including in EXEC SQL declare section 1-70
- shared between ESQL/C and C programs 1-69

C preprocessor

- role in compiling ESQL/C programs 1-10
- running first 1-10, 1-66

C preprocessor directives, using to define ESQL/C host variables 1-10

- c processor option 1-78, 1-80, 1-83

C programs, compiling 1-86

Callback function

- declaring 12-37, 12-39
- defining 12-38, 12-69
- definition of 12-36
- determining current connection 12-31
- disassociating 12-38, 12-55
- registering 12-39, 12-53
- calloc() system call 17-10
- Cardinality, ifx_cl_card() 9-45
- Case sensitivity 1-18, 1-33
- CBOOLTYPE data-type constant 2-11, 2-14

See also boolean data type.

- cc processor option 1-78, 1-80

CC8BITLEVEL environment variable 1-107

- ccargs preprocessor option 1-50

CCHARTYPE data-type

- constant 2-10, 2-33

See also char data type.

CCOLTYPE data-type

- constant 2-10

See also collection data type (ESQL/C).

CDATETYPE data-type

- constant 2-10

See also long int data type.

CDECIMALTYPE data-type
 constant 2-10
See also decimal data type.

CDOUBLETYPE data-type
 constant 2-10
See also double data type.

CDTIMETYPE data-type
 constant 2-10
See also datetime data type.

CFILETYPE data-type
 constant 2-10
See also char data type.

CFIXBINTYPE data-type
 constant 2-11
See also fixed binary data type.

CFIXCHARTYPE data-type
 constant 2-10
See also fixchar data type.

CFLOATTYPE data-type
 constant 2-10
See also float data type.

CHAR data type
 corresponding ESQL/C data
 type 2-4, 2-11, 3-3
 data conversion 3-11
 defined constant 2-8, 2-9, 15-20
 fetching 3-11
 inserting 3-12, 3-18
See also char data type; fixchar
 data type; string data type;
 varchar data type.

char data type
 converting from decimal 4-26,
 4-39, 5-35
 converting from double 5-38
 converting from int8 5-23
 converting from long int 5-41
 converting to decimal 4-15
 converting to int8 5-15
 defined constant 2-10
 definition of 3-4
 fetching into 2-18, 3-11, 3-13, 3-17,
 5-6, 6-35
 inserting from 3-12, 3-14, 3-17,
 3-18
 with ANSI-compliant
 database 3-18
See also CHAR data type.

CHARACTER data type. *See* CHAR
 data type.

Child process
 detaching from database
 server 12-56
 handling defunct 12-66

CINT8TYPE data-type
 constant 2-10
See also int8 data type.

CINTTYPE data-type constant 2-10
See also int data type.

CINVTTYPE data-type
 constant 2-10
See also interval data type.

Class value. *See* SQLSTATE
 variable, class and subclass
 codes.

Client-server environment
 and the cursor buffer 14-23
 architecture of 12-3
 connecting to a database 12-17
 local connection 12-4
 locating simple large objects 8-10
 optimized message
 transfers 12-41
 remote connection 12-4
 setting the cursor-buffer
 size 14-24

Client-side collection variable 9-11

CLIENT_GEN_VER version
 macro 13-15

CLIENT_GLS_VER version
 macro 13-15

CLIENT_LOCALE environment
 variable, in InetLogin
 structure 1-105

CLIENT_OS_VER version
 macro 13-15

CLIENT_SQLI_VER version
 macro 13-15

CLOB data type
 corresponding ESQL/C data
 type 2-5, 2-13
 declaring host variable for 7-7
 implementation of 10-48
 on optical disc 7-32
 role of locator.h 2-13
See also ifx_lo_t data type; Smart
 large-object data type.

CLOCATORTYPE data-type
 constant 2-10
See also loc_t data type.

CLONGTYPE data-type
 constant 2-10
See also long int data type.

CLOSE DATABASE
 statement 12-17, 12-18, 12-40,
 12-64, 12-98, 15-17

CLOSE statement 9-24, 14-13,
 14-15, 14-19
 optimizing 14-35

CLVCHARTYPE data-type
 constant 2-11
See also lvarchar data type.

CMONEYTYPE data-type
 constant 2-10
See also decimal data type.

COLLCHAR environment variable
 in InetLogin structure 1-105

collect.h header file, definition
 of 1-24

Collection data type
 cardinality, returning 9-45
 selecting entire row from 9-26

collection data type (ESQL/C)
 allocating memory for 9-12
 client-side 9-11

Collection Derived Table
 clause 9-14
 corresponding SQL data type 2-4,
 2-5, 2-12, 2-13
 deallocating memory for 9-12
 declaration 9-5
 defined constant 2-10
 fetching from 9-24
 fetching into 9-16
 initializing 9-16
 insert cursor for 9-21
 inserting into 9-18
 literal values 9-32
 operating on 9-13
 preparing statements that
 contain 14-8
 select cursor for 9-27
 typed collection variable 9-6
 untyped collection variable 9-9
 updating 9-29
See also Collection data type
 (SQL).

- Collection data type (SQL)
 - accessing 9-4
 - as element type of collection 9-6
 - as field type in row 9-50
 - declaring host variables for 9-5
 - definition of 9-4
 - deleting 9-44
 - extended identifier 15-9, 15-12
 - fetching 9-16, 9-24
 - in dynamic SQL 15-12
 - inserting into 9-18, 9-40, 9-41
 - literal values 9-32, 9-42, 9-43, 9-75
 - nested collection 9-26, 9-37, 9-43, 9-75
 - owner of 15-9, 15-13
 - selecting from 9-41
 - simple collection 9-13
 - updating 9-29, 9-40, 9-41
 - See also* collection data type (ESQL/C); LIST data type; MULTISET data type; SET data type.
- Collection data type (SQL), selecting a row from 9-26
- Collection derived table
 - for collection variables 9-13
 - for row variables 9-59
 - in DELETE 9-34, 9-68
 - in INSERT 9-18, 9-20, 9-64
 - in SELECT 9-24, 9-26, 9-64
 - in UPDATE 9-29, 9-66
- Colon (:)
 - between main variable and indicator variable 1-45
 - preceding host variables 1-21
 - specifying indicator variable 1-45
- Column (database)
 - determining data type 2-8
 - determining if truncated 11-15, 11-31
 - using data conversion 2-15
- Comma (,) symbol 5-28
- Command-line conventions
 - elements of Intro-10
 - example diagram Intro-12
 - how to read Intro-12
- Comment icons Intro-8
- Comments in ESQL/C program 1-21
- COMMIT WORK statement 12-29, 15-17
- Compiler
 - for ESQL/C programs 1-47
 - options. *See* Preprocessor option.
 - preprocessing 1-47
 - redirecting errors 1-64
 - syntax 1-48
 - version information 1-58
 - See also* Preprocessor.
- Compiler version
 - independence 1-89
- Compiling an ESQL/C program
 - default order, overview of 1-11
 - overview 1-9
 - the esql command 1-8, 1-9
 - the ESQL/C preprocessor 1-9
- Compiling dynamic thread application 13-51
- Compiling ESQL/C programs
 - default order 1-66
 - non-default order 1-68
- Complex data type. *See* Collection data type (SQL); Row type (SQL).
- Compliance, with industry standards Intro-15
- Conditional compilation
 - directives 1-12, 8-48
- Conditional compilation directives, ESQL/C
 - processing of 1-9
- Configuration information
 - in InetLogin structure 1-103
 - locations of 1-111
 - reading in 1-111
- CONNECT statement 12-11, 12-92
 - and explicit connections 12-17
 - determining database server features 12-26
 - determining name of a connection 11-11
 - determining name of a database server 11-11, 12-30
 - establishing a connection 12-19, 13-18
 - with an active transaction 12-29
- WITH CONCURRENT TRANSACTION clause 12-29
- Connection
 - using InetLogin structure 1-103, 12-7
 - using Registry 12-7
- Connection authentication 1-88
- Connection handle 12-93
- Connection. *See* Database server connection.
- Constants
 - for distinct bit 15-31
 - for ESQL/C data types 2-10
 - for smart large objects 7-10, 7-23, 7-41, 7-44, 7-58
 - for SQL data types 2-8, 15-19, 16-9
 - for SQL statements 11-33, 15-16
 - for varchar data 3-7
 - for X/Open SQL data types 2-9
 - with dynamic-management structures 15-16, 15-19
- Conversion. *See* Data conversion.
- COUNT descriptor field
 - after a DESCRIBE 15-14, 16-6
 - definition of 15-7
 - determining number of return values 16-27
 - initializing 16-5
 - saving 16-10, 16-14, 16-18, 16-29
 - setting 16-9, 16-36, 16-40
- cp preprocessor option 1-49
- CPFIRST environment variable 1-67
- cpu processor option 1-91
- CREATE DATABASE
 - statement 12-17, 12-18, 12-19, 12-26, 15-16
- CREATE FUNCTION
 - statement 14-46
- CREATE INDEX statement 12-33, 12-51, 15-16
- CREATE OPAQUE TYPE
 - statement, maxlength value and lvarchar 10-6
- CREATE PROCEDURE
 - statement 14-46, 15-17
- CREATE TABLE statement 7-14, 9-72, 12-33, 12-51, 15-16
- Create-time flags 7-67, 7-77
- create_clob sample program 7-100

CROWTYPE data-type
 constant 2-11
See also row data type (ESQL/C).

CSHORTTYPE data-type
 constant 2-10
See also short data type.

CSTRINGTYPE data-type
 constant 2-10, 2-33
See also string data type.

Cursor buffer
 changing size of 14-22
 default size 14-22
 description 14-20
 fetch buffer 14-20
 insert buffer 14-20
 sizing 14-20

Cursor function
 definition of 14-48
 known at compile time 14-50
 not known at compile time 16-28, 17-28
 parameterized 14-54
 with sqllda structure 17-28
 with system-descriptor area 16-28
See also SPL function.

Cursor names
 case sensitivity 1-18
 scope rules 1-64, 13-24
 specifying 14-19
 using delimited identifiers 1-33, 14-19
 using host variables 14-19

Cursor (database)
 and sqlca.sqlerrd 11-30
 deferring PREPARE 14-30
 definition of 14-14
 dynamic 13-17, 14-19
 for receiving rows 14-14
 for sending rows 14-18
 freeing 14-13, 14-24
 hold 14-15, 14-18
 identifying variable mismatch 11-15, 11-23, 11-31
 in thread-safe application 13-24
 insert 14-18
 interrupting the database server 12-51
 naming 14-19

optimizing 14-20
 scroll 14-15
 sequential 14-15, 14-18
 sizing the cursor buffer 14-20
 update 14-15
 using 14-14
See also Fetching; SELECT statement.

CVARBINTYPE data-type
 constant 2-11
See also var binary data type.

CVCHARTYPE data-type
 constant 2-10
See also varchar data type.

D

Data conversion
 arithmetic operations 2-19
 definition of 2-15
 for boolean values 2-14, 2-18
 for CHAR data type 3-11
 for char values 4-15, 4-26, 4-39, 5-15, 5-23
 for character data types 2-16, 2-18, 2-19, 3-11, 5-6
 for DATE values 6-37
 for DATETIME values 2-18, 6-33, 6-37, 6-51
 for DECIMAL values 2-20, 4-7, 5-17, 5-35
 for decimal values 4-15, 4-18, 4-20, 4-22, 4-26, 4-39, 4-42, 4-44, 4-47, 5-25
 for double values 4-18, 4-42, 5-24, 5-38
 for floating-point data types 2-18, 5-6
 for int values 4-20, 4-44, 5-26
 for INT8 values 5-8
 for INTERVAL values 2-18, 6-33, 6-37
 for long int values 4-22, 4-47, 5-19, 5-27, 5-41
 for MONEY values 5-35, 5-38, 5-41
 for numeric data types 2-16, 2-19, 2-20

for NVARCHAR data type 3-12
 for VARCHAR values 2-19, 3-12
 when fetching DATETIME 6-35
 when fetching INTERVAL 6-35
 when inserting DATETIME 6-35
 when inserting INTERVAL 6-35

DATA descriptor field
 after a DESCRIBE 16-6, 16-7
 after a FETCH 16-12, 16-14, 16-19, 16-28, 16-42
 allocating memory for 16-5
 definition of 15-7
 freeing memory for 16-12
 setting column value 16-30, 16-33
 setting input parameter 16-36, 16-41
 setting simple large-object column 16-55
See also Item descriptor.

Data type conversion. *See* Data conversion.

Data types
 array of host variables 1-36
 defined constants for 15-19, 16-9
 for dynamic thread library 13-43
 locale-specific 3-3
 locator structure 8-6
 pointers 1-39
 relationship between C and SQL types 2-3, 2-4
 structures 1-36, 1-37
 typedef expressions 1-38
 X/Open defined constants for 2-9, 15-20
See also ESQL/C data types; SQL data types.

Database
 closing 12-40
 determining available 12-32
 determining if ANSI-compliant 11-15, 11-22, 11-30, 12-26
 determining transaction logging 11-15, 11-22, 11-30, 12-26
 environment 12-5
 fetching CHAR data 3-11
 fetching DATETIME data 6-33
 fetching INTERVAL data 6-34

- fetching VARCHAR data 3-13
- inserting CHAR data 3-12
- inserting NCHAR data 3-12
- inserting NVARCHAR data 3-15
- inserting VARCHAR data 3-14
- opening 12-19
- Database cursor
 - in explicit connection 12-93
 - structure definitions 1-23
- Database server
 - connecting to 12-20
 - current 12-10
 - default 1-104, 12-7, 12-10, 12-25
 - determining available
 - databases 12-32
 - determining features of 12-26
 - determining name of 11-11, 12-30
 - determining type of 11-15, 11-23
 - in InetLogin structure 1-104
 - interrupting 12-51
 - message request 12-33, 12-38, 12-89, 14-22, 14-24, 14-30, 14-36
 - optimized message
 - transfers 12-41
 - optimizing OPEN, FETCH, and CLOSE 14-35
 - receiving configuration
 - information 12-8
 - reducing messages 14-30
 - specified 12-10
 - See also* DATABASE statement;
 - INFORMIX-Universal server.
- Database server connection
 - active 12-30, 12-47, 13-18
 - checking status of 12-32
 - current 7-53, 12-16, 12-28, 12-30, 12-46, 12-47
 - detaching from 12-32
 - determining features 12-26
 - determining name of 11-11, 12-30
 - dormant 12-16, 12-28, 13-18
 - explicit. *See* Explicit connection.
 - freeing resources of 12-64
 - implicit. *See* Implicit connection.
 - in thread-safe application 13-18
 - interrupting 12-33
 - switching between 12-28
 - terminating 12-40, 13-22
 - types of 12-15
 - using across threads 13-20
- DATABASE statement 12-95
- and implicit connections 12-17, 12-18, 12-67
- defined statement constant 15-16
- determining name of a
 - connection 11-11
- determining name of a database
 - server 11-11, 12-30
- opening a database 12-26
- starting a database server 12-19
- DATE data type
 - corresponding ESQ/L/C variable
 - type 2-4, 2-11
 - data conversion 6-37
 - declaring host variables for 6-3
 - defined constant 2-8
 - rdatestr() 6-6
 - rdayofweek() 6-8
 - rdefmtdate() 6-10
 - rfmtdate() 6-14
 - rjulmdy() 6-17
 - rleapyear() 6-19
 - rmidyjul() 6-21
 - rstodate() 6-23
 - rtoday() 6-26
 - See also* date data type.
- date data type
 - corresponding SQL data type 2-4, 2-11
 - data conversion 6-37
 - declaration 6-3
 - defined constant 2-10
 - See also* DATE data type.
- Date expressions
 - formatting 6-4
 - valid characters 6-4
- DATETIME data type
 - ANSI-standard qualifiers 6-36
 - corresponding ESQ/L/C data
 - type 2-4, 2-11
 - data conversion 2-18, 6-33, 6-35, 6-37, 6-51
 - declaring host variables for 6-29
 - defined constant 2-8
 - dtaddinv() 6-40
 - dtcurrent() 6-42
 - dtcvasc() 6-44
 - dtcvfntasc() 6-47
 - dtextend() 6-51
 - dtsubinv() 6-57
 - dtsub() 6-54
 - dttoasc() 6-59
 - dttofmntasc() 6-62
 - dynamically allocating structures
 - for 17-12
 - extending 6-33, 6-51
 - fetching 2-18, 6-33
 - inserting 2-18, 6-33
 - macros 6-31
 - precision of underlying decimal
 - value 6-29
 - qualifiers 6-29, 6-36
 - role of datetime.h 1-22, 2-11
 - See also* datetime data type.
- datetime data type
 - arithmetic operations 6-36
 - corresponding SQL data type 2-4
 - data conversion 6-37
 - declaration 6-29
 - defined constant 2-10
 - definition of 6-28
 - fetching into 6-33
 - inserting from 6-33
 - role of datetime.h 2-11
 - See also* DATETIME data type;
 - dttime_t typedef.
- datetime.h header file
 - contents and use 6-28, 6-40
 - data types defined 2-11, 2-12
 - definition of 1-22
 - macros defined 6-31, 17-12
- DBALSBC environment
 - variable 1-107
- DBANSIWARN environment
 - variable 1-59, 1-105, 11-23, 11-31, 12-17
- DBAPICODE environment
 - variable 1-107
- DBASCIIBC environment
 - variable 1-108
- DBCENTURY environment
 - variable 1-108, 6-11, 6-23, 6-44, 6-48, 6-63
- DBCODASET environment
 - variable 1-108

DBCONNECT environment variable 1-108
 DBCSConv environment variable 1-108
 DBCSOVERRIDE environment variable 1-108
 DBCSWIDTH environment variable 1-108
 DBDATE environment variable 1-104, 6-6, 6-23, 6-48, 6-63
 DBFLTMASK environment variable 1-108
 DBLANG environment variable 1-105
 DBMONEY environment variable 1-104, 5-28, 5-29, 5-35, 5-38, 5-41
 DBMONEYSCALE environment variable 1-109
 DBNLS environment variable 1-105
 DBPATH environment variable 1-104, 12-25, 12-49
 DBSS2 environment variable 1-109
 DBSS3 environment variable 1-109
 DBTEMP environment variable 1-104
 DBTIME environment variable 1-104, 6-47, 6-62, 6-68, 6-74
 DB_LOCALE environment variable 1-105
 DCE thread package. *See* Distributed Computing Environment (DCE).
 DEALLOCATE COLLECTION statement 9-12
 DEALLOCATE DESCRIPTOR statement 16-5, 16-12
 DEALLOCATE ROW statement 9-57
 DEC data type. *See* DECIMAL data type.
 decadd() library function 4-7, 4-9
 deccmp() library function 4-7, 4-11
 deccopy() library function 4-7, 4-13
 deccvasc() library function 4-7, 4-15

decvdbl() library function 4-7, 4-18
 deccvint() library function 4-7, 4-20
 deccvlong() library function 4-7, 4-22
 decdiv() library function 4-7, 4-24
 dececvt() library function 4-7, 4-26, 13-25, 13-30, 13-31
 decfcvt() library function 4-26, 13-25, 13-30, 13-31
 decfvct() library function 4-7
 Decimal arithmetic
 addition 4-9
 division 4-24
 multiplication 4-32
 subtraction 4-37
 DECIMAL data type
 corresponding ESQl/C variable type 2-4, 2-12
 data conversion 2-19, 2-20, 2-21, 4-7, 5-17, 5-35
 decadd() 4-9
 deccmp() 4-11
 deccopy() 4-13
 deccvasc() 4-15
 deccvbl() 4-18
 deccvint() 4-20
 deccvlong() 4-22
 decdiv() 4-24
 dececvt() 4-26
 decfcvt() 4-26
 decimal structure shown 4-4
 declaring host variables for 4-3
 decmul() 4-32
 decrround() 4-34
 decsub() 4-37
 dectoasc() 4-39
 dectodbl() 4-42
 dectoint() 4-44
 dectolong() 4-47
 dectrunc() 4-49
 defined constant 2-8, 2-9, 15-20
 fixed-point decimals 2-20, 2-22
 number of decimal digits 2-18, 5-6
 role of decimal.h 1-22, 2-12
 scale and precision 2-22, 15-8
See also decimal data type.

decimal data type
 addition 4-9
 arithmetic operations 4-7
 comparing 4-11
 converting from double 4-18
 converting from int8 5-25
 converting from integer 4-20
 converting from long int 4-22
 converting from text 4-15
 converting to double 4-42
 converting to int 4-44
 converting to int8 5-17
 converting to long int 4-47
 converting to text 4-26, 4-39, 5-35
 copying 4-13
 corresponding SQL data type 2-4
 data conversion 2-20, 4-7
 declaration 4-3
 defined constant for DECIMAL data type 2-10
 defined constant for MONEY data type 2-10
 division 4-24
 floating-point decimals 2-20, 2-22
 in thread-safe application 13-25
 multiplication 4-32
 role of decimal.h 2-12
 rounding 4-34
 subtraction 4-37
 truncating 4-49
See also DECIMAL data type; MONEY data type; dec_t typedef.
 decimal structure 4-4
 decimal.h header file
 data types defined 2-12, 4-4
 definition of 1-22
 DECLARE SECTION 1-27
 excluding C header file statements from 1-71
 including C declaration syntax in 1-71
 DECLARE statement
 and sqlca structure 11-30, 11-39
 in thread-safe application 13-24
 insert cursor for collection variable 9-21
 select cursor for collection variable 9-27

- with a SELECT statement 14-15
- with an EXECUTE FUNCTION statement 14-15, 16-28, 17-28
- with an INSERT statement 14-18
- with deferred PREPARE 14-30
- with OPTOFC and Deferred PREPARE features 14-38
- decmul() library function 4-7, 4-32
- decround() library function 4-7, 4-34
- decsub() library function 4-7, 4-37
- dectoasc() library function 4-7, 4-39
- dectodbl() library function 4-7, 4-42
- dectoint() library function 4-7, 4-44
- dectolong() library function 4-7, 4-47
- detrunc() library function 4-7, 4-49
- dec_t typedef
 - defined constant for DECIMAL data type 2-10
 - defined constant for MONEY data type 2-10
 - definition of 4-4
- .def file extension 1-79, 1-84, 1-86
- Default locale Intro-4
- Default order of compilation of
 - ESQL/C programs, overview of 1-11
- Deferred PREPARE feature 14-30
 - enabling 14-32
 - restrictions on 14-31
- SET DEFERRED_PREPARE statement 14-32
 - with AUTOFREE and OPTOFC features 14-39
 - with OPTOFC feature 14-38
- define directives, ESQL/C
 - processing of 1-9
- define preprocessor directive 1-12, 1-14, 1-60
- DELETE statement
 - and NOT FOUND condition 11-20, 11-35
 - collection columns 9-44
 - Collection Derived Table clause 9-34, 9-68
 - defined statement constant 15-16, 15-17
- determining estimated cost of 11-30
- determining number of rows deleted 11-8, 11-30
- determining rowid 11-30
- dynamic 14-40, 14-52, 15-25
- failing to access rows 11-29
- in ANSI-compliant database 3-18
- interrupting 12-33, 12-51
- known at compile time 14-40, 14-52
- not known at compile time 15-25
- parameterized 14-52, 15-26, 16-46, 17-43
- row variables 9-68
- WHERE CURRENT OF clause 14-19
 - with DESCRIBE 15-15, 15-22
 - without WHERE clause 11-15, 11-23, 11-31, 15-17, 15-22
- DELIMIDENT environment variable 1-33, 1-106, 9-68
- Delimited identifier 1-32, 9-68, 14-19
 - See also* SQL identifier.
- demo1 sample program 1-116
- demo2 sample program 14-54
- demo3 sample program 16-15, 16-37, 17-18
- demo4 sample program 14-54, 16-15, 17-34
- Demonstration database Intro-5
 - creating and populating B-3
- Demonstration programs
 - location 1-116
 - source files for B-1
- DESCRIBE statement
 - allocating memory for data 2-30, 16-6, 16-7, 17-10
 - allocating memory for sqlda 17-6
 - and deferred PREPARE 14-34
 - and lvarchar host variables 16-7
 - and sqlca structure 15-22
 - determining column data type 2-8, 3-7, 15-20, 15-21
 - determining return-value data type 15-20
 - determining SQL statement type 11-29, 11-33, 15-16
- initializing sqlda structure 17-7
- initializing system-descriptor area 16-6
- INTO clause 15-15, 17-4, 17-6, 17-7, 17-43
- role in dynamic SQL 14-10
- setting COUNT field 16-6
- SQLCODE value 11-33, 15-16
- SQLSTATE value. *See* DESCRIBE statement, SQLCODE value.
- USING SQL DESCRIPTOR
 - clause 15-14, 16-4, 16-6, 16-14, 16-29, 16-46
- warnings after 11-15, 11-23, 11-31
 - with an item descriptor 16-14, 16-29
 - with deferred PREPARE 14-31
 - with input parameters 16-6
 - with sqlvar_struct 17-17, 17-27
 - X/Open data type values 15-20
- Diagnostic information
 - definition of 11-4
 - with GET DIAGNOSTICS statement 11-7
 - with the sqlca structure 11-27
- Diagnostics area
 - CLASS_ORIGIN field 11-10, 11-22, 11-24, 11-25
 - CONNECTION_NAME field 11-11, 12-30
 - definition of 11-7
 - INFORMIX_SQLCODE field 11-10, 11-13
 - MESSAGE_LENGTH field 11-11
 - MESSAGE_TEXT field 11-11, 11-24
 - MORE field 11-8
 - NUMBER field 11-8
 - RETURNED_SQLSTATE field 11-10, 11-12
 - ROW_COUNT field 11-8
 - SERVER_NAME field 11-11, 12-30
 - SUBCLASS_ORIGIN field 11-10, 11-22, 11-24, 11-25
 - undefined fields 11-13
 - See also* GET DIAGNOSTICS statement.

- Directive. *See* Preprocessor directive.
- DISCONNECT statement 12-92, 12-97
 - and explicit connections 12-17
 - and open transactions 12-64
 - in thread-safe application 13-22
 - terminating a database server connection 12-40
- dispcat_pic sample program 8-44, 11-59
- Distinct bit 15-31
- Distinct data type
 - algorithm for determining 15-32
 - distinct bit 15-31
 - dynamically executing 15-30
 - extended identifier 15-9, 15-12
 - in dynamic SQL 15-12
 - name of 15-9, 15-13
 - owner of 15-9, 15-13
 - source data type 15-9, 15-13, 15-31
- Distributed Computing
 - Environment (DCE) 13-16, 13-25, 13-26
- .dll file extension 1-79, 1-93, 1-94, 12-20
- DLL Registry 1-88
- Documentation conventions
 - command-line Intro-10
 - icon Intro-8
 - typographical Intro-7
- Documentation notes Intro-14
- Documentation, types of
 - error message files Intro-13
 - on-line manuals Intro-13
 - printed manuals Intro-13
- Dollar (\$) sign
 - as formatting character 5-29
 - between main variable and indicator variable 1-45
 - for function parameters 1-40
 - relation to SQL keyword protection 1-72
 - to declare host variables 1-21, 1-27
 - with embedded SQL statements 1-17
 - with preprocessor directives 1-12

- Dot notation 9-66, 9-74
- Double dash (--) 1-21
- double data type
 - converting from decimal 4-42
 - converting from int8 5-24
 - converting to decimal 4-18
 - converting to int8 5-16
 - converting to text 5-38
 - corresponding SQL data type 2-4, 2-12, 5-5
 - data conversion 2-20
 - defined constant 2-10
 - See also* FLOAT data type.
- DOUBLE PRECISION data type.
 - See* FLOAT data type.
- Double quotes (")
 - delimiting identifiers 1-32, 9-68
 - escaping 1-19
 - in a literal collection 9-43
 - in a literal row 9-76
 - in a prepared statement string 14-5
 - in a quoted string 1-20
- DROP DATABASE
 - statement 12-17, 12-18, 12-19, 15-16
- DROP FUNCTION
 - statement 14-46
- DROP PROCEDURE
 - statement 14-46
- dtaddinv() library function 6-36, 6-40
- dtcurrent() library function 6-36, 6-42
- dtcvasc() library function 6-37, 6-44
- dtcvfmtasc() library function 6-37, 6-47
- dtextend() library function 6-33, 6-37, 6-51
- dtime structure 6-29
- dtime_t typedef
 - defined constant 2-10
 - definition 6-29, 6-33
 - setting qualifiers for DATETIME 17-12
- dtsubinv() library function 6-36, 6-57
- dtsub() library function 6-36, 6-54
- dttoasc() library function 6-37, 6-59

- dttofmtasc() library function 6-37, 6-62, 16-62
- Dynamic link library (DLL)
 - and import library 1-95
 - building 1-96
 - definition 1-94
 - ESQL client-interface 12-13
 - esqlauth.dll 1-88, 12-15
 - locating 1-95
 - Registry 1-88
 - sharing 12-20
 - with WHENEVER 11-45
- Dynamic SQL
 - assembling the statement string 14-5
 - definition of 14-3, 14-4
 - describing the statement 14-10, 15-14
 - executing the statement 14-10
 - freeing resources 14-13
 - memory management 15-4
 - non-SELECT statements known at compile time 14-40
 - non-SELECT statements not known at compile time 15-24, 15-25
 - preparing the statement 14-5
 - SELECT statements known at compile time 14-43
 - SELECT statements not known at compile time 15-23, 15-25
 - statements not known at compile time 15-27
 - statements used 14-11, 14-15, 14-18, 16-4, 17-4
 - statements with user-defined data types 15-27
 - user-defined functions known at compile time 14-46
 - user-defined functions not known at compile time 15-26
- Dynamic thread functions,
 - registering 13-47
- Dynamic thread library
 - creating 13-40
 - data types for 13-43
 - registering functions 13-47

Dynamic-management structure
 sqlda structure 15-10, 17-3
 system-descriptor area 15-5, 16-3
See also System-descriptor area;
 sqlda structure.
 dynthr_init() function 13-49
 dyn_sql sample program 11-59

E

-e preprocessor option 1-49, 1-53,
 1-58
 .ec file extension 1-10, 1-48, 1-80,
 1-81
 .ecp file extension 1-10, 1-48
 -ED preprocessor option 1-51, 1-55,
 1-60
 elif preprocessor directive 1-12,
 1-16
 else preprocessor directive 1-12,
 1-16
 Empty string 3-6
 END OF DATA condition. *See* NOT
 FOUND condition.
 endif preprocessor directive 1-12,
 1-16
 Environment variable
 CC8BITLEVEL 1-107
 CLIENT_LOCALE 1-105
 COLLCHAR 1-105
 CPFIRST 1-67
 DBALSBC 1-107
 DBANSIWARN 1-59, 1-105,
 11-23, 11-31, 12-17
 DBASCIIBC 1-108
 DBCENTURY 1-108, 6-11, 6-23,
 6-44, 6-48, 6-63
 DBCODESET 1-108
 DBCONNECT 1-108
 DBCSCONV 1-108
 DBCSOVERRIDE 1-108
 DBCSWIDTH 1-108
 DBDATE 1-104, 6-6, 6-23, 6-48,
 6-63
 DBFLTMASK 1-108
 DBLANG 1-105
 DBMONEY 1-104, 5-28, 5-29,
 5-35, 5-38, 5-41

DBMONEYSCALE 1-109
 DBNLS 1-105
 DBPATH 1-104, 12-25, 12-49
 DBSS2 1-109
 DBSS3 1-109
 DBTEMP 1-104
 DBTIME 1-104, 6-47, 6-62, 6-68,
 6-74
 DB_LOCALE 1-105
 DELIMIDENT 1-33, 1-106
 ESQLMF 1-107
 FET_BUF_SIZE 1-107, 14-23
 GL_DATE 1-107, 6-6, 6-23, 6-48,
 6-63
 GL_DATETIME 1-107
 IFX_AUTOFREE 14-25
 IFX_DEFERRED 14-32
 in thread-safe application 13-25
 INFORMIXC 1-77
 INFORMIXCONRETRY 1-106
 INFORMIXCONTIME 1-106
 INFORMIXDIR 1-48, 1-63,
 1-105, B-2
 INFORMIXSERVER 1-104, 1-118,
 12-7, 12-49, 12-67, 16-51, B-2
 INFORMIXSQLHOSTS 12-7,
 12-12, B-2
 LANG 1-105
 LC_COLLATE 1-105
 LC_CTYPE 1-106
 LC_MONETARY 1-106
 LC_NUMERIC 1-106
 LC_TIME 1-106
 LD_LIBRARY_PATH 13-9, 13-50
 OPTMSG 12-42
 OPTOFC 14-37
 PATH 1-96, B-2
 precedence 1-111
 retrieving 1-97
 setting 1-97
 THREADLIB 13-26, 13-27, 13-50
 en_us.8859-1 locale Intro-4
 eprotect utility
 protecting SQL keywords 1-72
 -u mode 1-68, 1-72
 Error code
 INFORMIX FIND ERROR
 utility 1-6
 Error code, finderr utility 1-6

Error handling
 retrieving an error message 11-48
 role of sqlca.h 1-23
 using in-line code 11-41
 with optimized message
 transfers 12-44
See also Exception handling;
 Warnings.
 Error message files Intro-13
 Error messages
 determining length of 11-11
 Informix-specific 11-40, 11-54
 obtaining parameters 11-29
 redirecting 1-64
 retrieving text of 11-11, 11-47,
 11-48, 11-51
 Errors. *See* Exceptions; Runtime
 errors; Warnings.
 Escape character 1-19, 9-43, 9-76
 Escape characters, multibyte filter
 for 1-6
 ESQL client-interface DLL 12-13
 contents of 1-94
 description 1-88
 ESQL client-interface library 1-84,
 1-95
 esql command
 calling C preprocessor and
 compiler 1-10
 compatibility issues 13-12
 compile-only options 1-78
 library options 13-6
 linking options 1-86
 options passed implicitly 1-82
 preprocessing options 1-58
 requirements for using 1-10, 1-48
 steps it performs 1-47
 syntax 1-48
 syntax on UNIX systems 1-49
 version information 1-54, 1-57
 Windows options 1-90, 1-91
See also Compiler; Preprocessor.
 ESQL preprocessor
 stage1 1-9
 ESQLAUTH sample
 program 12-15
 esqlauth.c authorization file 12-14
 esqlauth.dll ESQL client-interface
 DLL 1-88

- esqlauth.dll Informix DLL 12-15
- ESQLMF environment
 - variable 1-107
- ESQLMF.EXE multibyte filter 1-6
- ESQL/C compiler. *See* Compiler; C compiler.
- ESQL/C conditional compilation
 - directives
 - processing of 1-9
- ESQL/C data types
 - boolean 2-13
 - char 3-4
 - character data types 3-3
 - collection 2-4, 2-5, 2-12, 2-13, 9-4
 - date 2-4, 2-11, 6-3
 - datetime 2-4, 2-11, 6-28
 - decimal 2-4, 2-12, 4-3
 - defined constants for 2-7, 2-10, 15-21
 - double 2-4, 2-12, 5-5
 - fixchar 2-4, 2-11, 2-12, 3-4, 3-5
 - fixed binary 2-5, 2-13
 - float 2-4, 2-12, 5-5
 - floating-point data types 5-5
 - ifx_lo_t 2-5, 2-11, 2-13, 7-6
 - int8 2-5, 5-7
 - integer data types 5-4
 - interval 2-4, 2-12, 6-28
 - loc_t 2-4, 2-5, 2-11, 2-12, 8-5
 - long int 2-12, 5-4
 - lvvarchar 2-5, 2-12, 2-13, 3-4, 3-9
 - row 2-5, 2-13, 9-48
 - short int 2-12, 5-4
 - string 2-4, 2-12, 3-4, 3-5
 - trailing blanks 3-5
 - var binary 2-5, 2-13
 - varchar 2-4, 2-12, 3-4, 3-6
- ESQL/C define directives,
 - processing of 1-9
- ESQL/C host variables
 - using C preprocessor directives to
 - define 1-10
- ESQL/C include directives,
 - preprocessing of 1-9
- ESQL/C library functions
 - character and string
 - functions 3-19
 - connection functions 12-92

- database server control
 - functions 12-34, 12-38, 12-46
- DATE type functions 6-5
- DATETIME type functions 6-38
- DECIMAL type functions 4-8, 13-30
- environment variable
 - functions 1-97
- error message functions 11-47
- formatting functions 5-34
- function prototypes 1-25
- INT8 type functions 5-9
- INTERVAL type functions 6-38
- numeric formatting
 - functions 5-34
- size and alignment functions 2-23
- smart large-object functions 7-33
 - See also* Function library; Macro.
- ESQL/C preprocessor, stage2 1-9
- ESQL/C preprocessor. *See* Preprocessor.
- ESQL/C program. *See* Program.
- EU preprocessor option 1-51, 1-55, 1-60
- Exception handling
 - definition of 11-4
 - determining number of
 - exceptions 11-8
 - displaying error text 11-40, 11-54
 - NOT FOUND condition 11-20, 11-35, 11-36, 11-45
 - retrieving error message
 - text 11-11, 11-47
 - runtime errors 11-24, 11-38, 11-45
 - success condition 11-20, 11-34
 - using the WHENEVER
 - statement 11-45
 - warning conditions 11-22, 11-37, 11-45
 - with sqlca structure 11-34
 - with SQLSTATE variable 11-19
 - See also* Error handling; NOT FOUND condition; Runtime errors; Warnings.
- Exclamation point (!), wildcard in
 - smart-large-object
 - filenames 7-42
- .exe file extension 1-79, 1-93, 1-94, 12-20

- EXEC SQL keywords
 - BEGIN DECLARE SECTION. *See* DECLARE SECTION.
 - END DECLARE SECTION. *See* DECLARE SECTION.
 - to declare host variables 1-27
 - with embedded SQL
 - statements 1-17
 - with preprocessor directives 1-12
- EXECUTE FUNCTION statement
 - associated with a cursor 14-14, 14-50, 14-51
 - defined statement constant 15-17, 16-23, 16-26, 17-27
 - executing a cursor function 14-50
 - executing a noncursor
 - function 14-49
 - for user-defined functions 14-46, 14-48, 15-26
- INTO host_var clause 14-48
- known at compile time 14-45, 14-48
- not known at compile time 15-23, 15-25
- parameterized 14-52, 14-54, 15-25
- with DESCRIBE 15-15, 15-26, 16-6, 17-7, 17-8
- with dynamic-management
 - structures 15-26
- with sqlda structure 17-26
- with system-descriptor
 - area 16-22
- See also* User-defined function.
- EXECUTE IMMEDIATE
 - statement 11-39, 14-42, 14-47
- EXECUTE PROCEDURE
 - statement 12-33, 14-40, 14-46, 14-47, 14-52, 15-17, 15-25
 - See also* User-defined procedure.
- EXECUTE statement
 - INTO DESCRIPTOR clause 17-4, 17-15, 17-28
 - INTO host_var clause 11-31, 14-43, 14-49
 - INTO SQL DESCRIPTOR
 - clause 16-4, 16-11, 16-22, 16-23
 - role in dynamic SQL 14-10
 - SQLCODE values 11-40

- USING DESCRIPTOR
 - clause 16-4, 17-4, 17-14, 17-31, 17-41, 17-42, 17-43
- USING host_var clause 14-52
- USING SQL DESCRIPTOR
 - clause 16-11, 16-30, 16-43, 16-44, 16-45, 16-46
- with noncursor functions known at compile time 14-49
- with non-SELECT statements known at compile time 14-41
- with singleton SELECT statements known at compile time 14-43
- with user-defined procedures 14-47
- exec() system call 12-57, 13-25
- exit() system call 11-33, 11-44
- Explicit connection
 - connection handle 12-93
 - default 12-25
 - definition of 12-16, 12-20
 - establishing 12-19, 12-94
 - identifying 12-30
 - limits of 12-93
 - starting 12-19
 - switching to 12-99
 - terminating 12-40, 12-97
 - when to use 12-20
 - with sqlexit() 12-64
- Exporting runtime routines 1-89
- Extended identifier 15-9, 15-12, 15-30
- External function
 - definition 14-47
 - executing dynamically 14-49, 14-50, 16-23, 16-29, 17-28, 17-29
 - iterator function 14-50
 - See also* User-defined function.
- External procedure 14-47
 - See also* User-defined procedure.
- External routine 14-47
- EXTYPEID descriptor field 15-9
- EXTYPELENGTH descriptor field 15-9
- EXTYPENAME descriptor field 15-9

- EXTYPEOWNERLENGTH
 - descriptor field 15-9
- EXTYPEOWNERNAME descriptor field 15-9

F

- f processor option 1-54, 1-81
- fclose() system call 1-62
- Feature icons Intro-9
- Features, product Intro-5
- FetArrSize global variable 15-37
 - and FetBufSize 15-38
 - with a fetch array 15-34
- FetBufSize global variable 14-23
 - and FetArrSize 15-38
- Fetch array
 - allocating memory for 15-37
 - allocating memory, example 15-39
 - and simple large objects 15-33
 - description of 15-33
 - FetArrSize global variable 15-34
 - freeing memory 15-45
 - obtaining values from 15-42
 - sample program 15-35
 - use of loc_t structure 15-38
 - using 15-33
- USING DESCRIPTOR
 - clause 15-34
 - using sqllda structure with 15-34
 - with Deferred PREPARE and OPTOFC features 15-35
- Fetch buffer 14-15, 14-20
- FETCH statement
 - and NOT FOUND condition 11-20, 11-35
 - checking for truncation 11-31
 - fetching into a collection variable 9-28
 - getting values from a system-descriptor area 16-10
- INTO DESCRIPTOR clause 17-28
- INTO host_var clause 1-38, 11-31
- INTO SQL DESCRIPTOR
 - clause 16-28
 - optimizing 14-35
 - retrieving a row 14-15

- USING DESCRIPTOR
 - clause 17-4, 17-14, 17-18, 17-29, 17-34
- USING SQL DESCRIPTOR
 - clause 16-4, 16-11, 16-14, 16-15, 16-28, 16-37
- warnings 11-31
- with aggregate functions 11-31
- with fetch array 15-33
- with OPTOFC and Deferred PREPARE features 14-38
- See also* Fetch buffer; Fetching.
- Fetching
 - CHAR values 3-11
 - character data 3-11
 - collection data 9-16
 - data conversion 2-15
 - DATETIME values 2-18, 6-33
 - determining rowid 11-30
 - INTERVAL values 2-18, 6-34
 - into char host variable 2-18, 3-11, 3-13, 3-17, 5-6, 6-35
 - into collection host variable 9-37, 9-41
 - into datetime host variable 6-33
 - into fixchar host variable 2-18, 3-11, 3-13, 3-17, 5-6, 6-35
 - into fixed binary host variable 10-23
 - into ifx_lo_t host variable 7-22
 - into interval host variable 6-34
 - into lvarchar host variable 10-14
 - into row host variable 9-73
 - into string host variable 2-18, 3-11, 3-13, 3-17, 5-6, 6-35
 - into var binary host variable 10-30
 - into varchar host variable 2-18, 3-11, 3-17, 5-6
- NCHAR values 3-12
- NVARCHAR values 3-15
- row-type data 9-62
- VARCHAR values 2-19, 3-13
- See also* SELECT statement; Cursor (database).
- FET_BUF_SIZE environment variable 1-107, 14-23

- File
 - copying a smart large object to 7-41
 - copying to a smart large object 7-44
 - getting name for a smart large object 7-50
 - named file as a simple large-object location 8-26, 16-33, 16-34
 - open file as a simple large-object location 8-21

- File extension
 - .a 1-76
 - .c 1-9, 1-58, 1-76, 1-78, 1-80, 1-86, 13-15
 - .def 1-79, 1-84, 1-86
 - .dll 1-79, 1-93, 1-94, 12-20
 - .ec 1-10, 1-48, 1-80, 1-81
 - .ecp 1-48
 - .exe 1-79, 1-93, 1-94, 12-20
 - .h 1-14
 - .icp 1-50
 - .lib 1-79
 - .o 1-75, 1-86
 - .obj 1-79, 1-80
 - .rc 1-79, 1-86, 1-88
 - .res 1-79, 1-86, 1-88
 - .sl 13-13
 - .so 13-8, 13-13

- Filename
 - C object file 1-79
 - C source file 1-78
 - compiled resource file 1-79, 1-86
 - ESQL/C executable 1-54, 1-79
 - ESQL/C libraries 1-54, 1-86
 - include file 1-56
 - log file 1-56
 - module-definition file 1-79, 1-86
 - options for 1-58
 - project file 1-54, 1-81
 - resource file 1-79, 1-86
 - response file 1-54, 1-82
 - runtime library 1-92
 - system library 1-79
 - user-defined library 1-79

- File-open mode flags 8-20
- finderr utility Intro-13, 1-6
- fixchar data type

- corresponding SQL data type 2-4, 2-11, 2-12
- defined constant 2-10
- definition of 3-4, 3-5
- fetching into 2-18, 3-11, 3-13, 3-17, 5-6, 6-35
- for boolean values 2-14
- inserting from 3-12, 3-14, 3-17, 3-18
- with ANSI-compliant database 3-18
- See also* CHAR data type.
- fixed binary data type
 - checking for null 10-23
 - corresponding SQL data type 2-5, 2-13
 - declaration 10-18
 - defined constant 2-11
 - fetching into 10-23
 - for smart large objects. *See* ifx_lo_t data type.
 - inserting from 10-22
 - setting to null 10-22
 - use with smart large objects 7-7
 - See also* Fixed-length opaque data type; Smart large-object data type.
- Fixed-length opaque data type
 - declaring host variable for 10-18
 - inserting 10-13, 10-22
 - selecting 10-14, 10-23
 - See also* fixed binary data type; Varying-length opaque data type.
- Fixed-point decimals 2-20, 2-22
- FLOAT data type
 - corresponding ESQL/C data type 2-4, 2-12, 5-5
 - data conversion 2-19, 2-20, 2-21
 - defined constant 2-8, 2-9, 15-20
 - determining how stored 11-15, 11-23, 11-30, 12-27
 - number of decimal digits 2-18, 5-6
 - See also* double data type.
- float data type
 - corresponding SQL data type 2-4, 2-12, 5-5
 - data conversion 2-20

- defined constant 2-10
- passed as double 5-5
- See also* SMALLFLOAT data type.
- Floating-point decimals 2-18, 2-20, 2-22, 4-42, 5-6, 5-24
- FLUSH statement 9-24, 14-19, 16-35, 17-32
- fopen() system call 1-62
- for 1-6
- fork() system call 8-62, 12-56, 13-25
- Formatting function
 - rdefmtdate() 6-10
 - rfmtdate() 6-14
 - rfmtdec() 5-35
 - rfmtdouble() 5-38
 - rfmtlong() 5-41
- fread() system call 1-62
- FREE statement
 - freeing cursor resources 14-13, 14-15, 14-19
 - freeing statement-identifier resources 14-13
 - role in dynamic SQL 14-13
- Freeing a cursor
 - automatically 14-24
- Freeing memory, fetch array 15-45
- free() system call
 - freeing a simple large-object memory buffer 8-14
 - freeing an sqlda structure 17-25
 - freeing column data buffer 17-25
- Function
 - callback 12-36
 - cursor 14-48, 14-50
 - iterator 14-50
 - noncursor 14-48, 14-49
 - parameters 14-19, 16-62
 - signal handler 12-34
- Function cursor
 - definition of 14-16
 - fetch buffer 14-20
 - statements that manage 14-15
 - using 14-50
 - with sqlda structure 17-28
 - with system-descriptor area 16-28
- Function libraries, for ESQL/C 1-7

Function library

bycmp() 3-21
bycopy() 3-24
byfill() 3-26
bylen() 3-28
decadd() 4-9
deccmp() 4-11
deccopy() 4-13
deccvasc() 4-15
deccvdbl() 4-18
deccvint() 4-20
deccvlong() 4-22
decdiv() 4-24
dececv() 4-26, 13-25, 13-30, 13-31
decfcvt() 4-26, 13-25, 13-30, 13-31
decmul() 4-32
decround() 4-34
decsub() 4-37
dectoasc() 4-39
dectodbl() 4-42
dectoint() 4-44
dectolong() 4-47
dectrunc() 4-49
dtaddinv() 6-40
dtcurrent() 6-42
dtcvasc() 6-44
dtcvfntasc() 6-47
dtextend() 6-51
dtsubinv() 6-57
dtsub() 6-54
dttoasc() 6-59
dttofmntasc() 6-62, 16-62
GetConnect() 12-92, 12-94
ifx_cl_card() 9-45
ifx_dececv() 13-25, 13-31
ifx_decfcvt() 13-25, 13-31
ifx_getcur_conn_name() 12-47
ifx_getenv() 1-102
ifx_getserial() 5-10
ifx_int8add() 5-12
ifx_int8cmp() 5-13
ifx_int8copy() 5-14
ifx_int8cvasc() 5-15
ifx_int8cvdbl() 5-16
ifx_int8cvdec() 5-17
ifx_int8cvint() 5-18
ifx_int8cvlong() 5-19
ifx_int8div() 5-20
ifx_int8mul() 5-21

ifx_int8sub() 5-22
ifx_int8toasc() 5-23
ifx_int8todbl() 5-24
ifx_int8todec() 5-25
ifx_int8toint() 5-26
ifx_int8tolong() 5-27
ifx_lo_alter() 7-36
ifx_lo_close() 7-38
ifx_lo_col_info() 7-39
ifx_lo_copy_to_file() 7-41, 9-45
ifx_lo_copy_to_lo() 7-44
ifx_lo_create() 7-46
ifx_lo_def_create_spec() 7-49
ifx_lo_filename() 7-50
ifx_lo_from_buffer() 7-52
ifx_lo_open() 7-52, 7-53
ifx_lo_readwithseek() 7-57
ifx_lo_read() 7-55
ifx_lo_release() 7-59
ifx_lo_seek() 7-61
ifx_lo_specget_estbytes() 7-64
ifx_lo_specget_extsz() 7-66
ifx_lo_specget_flags() 7-67
ifx_lo_specget_maxbytes() 7-69
ifx_lo_specget_sbspace() 7-71
ifx_lo_specset_estbytes() 7-73
ifx_lo_specset_extsz() 7-75
ifx_lo_specset_flags() 7-77
ifx_lo_specset_maxbytes() 7-79
ifx_lo_specset_sbspace() 7-80
ifx_lo_spec_free() 7-63
ifx_lo_stat() 7-81
ifx_lo_stat_atime() 7-83
ifx_lo_stat_cspec() 7-84
ifx_lo_stat_ctime() 7-86
ifx_lo_stat_free() 7-87
ifx_lo_stat_mtime_sec() 7-88
ifx_lo_stat_refcnt() 7-89
ifx_lo_stat_size() 7-91
ifx_lo_tell() 7-92
ifx_lo_to_buffer() 7-93
ifx_lo_truncate() 7-95
ifx_lo_writewithseek() 7-98
ifx_lo_write() 7-96
ifx_putenv() 1-100
incvasc() 6-65
incvfntasc() 6-68
intoasc() 6-71
intofmtasc() 6-74, 16-62

invdivdbl() 6-77
invdivinv() 6-80
invextend() 6-82
invmuldbl() 6-85
ldchar() 3-30, 8-67
rdatestr() 6-6
rdayofweek() 6-8
rdefmtdate() 6-10
rdownshift() 3-32
ReleaseConnect() 12-92, 12-97
rfmtdate() 6-14, 16-62
rfmtdec() 5-35, 16-62
rfmtdouble() 5-38
rfmtlong() 5-41
rgetlmsg() 11-48
rgetmsg() 11-51
risnull() 1-43, 2-24
rjulmdy() 6-17
rleapyear() 6-19
rmidyjul() 6-21
rsetnull() 2-27
rstod() 3-34
rstoi() 3-36
rstol() 3-38
rstrdate() 6-23
rtoday() 6-26
rtypalign() 2-30, 17-12, 17-23
rtypmsize() 2-33, 17-12, 17-23
rtypname() 2-36
rtypwidth() 2-39
rupshift() 3-40
SetConnect() 12-92, 12-99
sqgetdbs() 12-48
sqlbreakcallback() 12-36, 12-53
sqlbreak() 12-33, 12-51, 12-69
sqldetach() 12-32, 12-40, 12-56, 13-25
sqldone() 12-32, 12-63, 12-69
sqlxexit() 12-40, 12-64
sqlsignal() 12-65
sqlstart() 12-19, 12-64, 12-67
stcat() 3-42, 14-7
stchar() 3-44
stcmp() 3-46
stcopy() 3-48, 14-7
stleng() 3-50
See also Macro.

Functions

- dynamic thread library 13-40
- dynamic thread, registering 13-40

G

- G preprocessor option 1-51, 1-56, 1-64
- g preprocessor option 1-51, 1-56, 1-64
- GET DESCRIPTOR statement
 - getting COUNT field 16-14, 16-29
 - getting fields 16-4, 16-10
 - getting row values 16-14
 - setting COUNT field 16-10
 - with OPTOFC and Deferred PREPARE features 14-38*See also* System-descriptor area.
- GET DIAGNOSTICS statement and OPTOFC feature 14-36
- ANSI SQL compliance 11-6
- description 11-7
- examples of use 11-9, 11-12, 11-26, 11-55
- exception information 11-9
- retrieving multiple exceptions 11-25
- RETURNED_SQLSTATE field 11-13
- SQLCODE variable 11-13
- SQLSTATE variable 11-12
- statement information 11-7
- X/Open compliance 11-6
- See also* Diagnostics area.
- getcd_me sample program 8-15
- getcd_nf sample program 8-27
- getcd_of sample program 8-22
- GetConnect() library
 - function 12-92, 12-94
- Global ESQL/C variables 1-115
- Global Language Support (GLS) Intro-4
- Global Language Support (GLS) environment
 - character data types for host variables 3-3
 - inserting character data 3-12, 3-15
 - naming host variables 1-28

- naming indicator variables 1-45
- transferring character data 3-11
- Global variable
 - FetBufSize 14-23
 - OptMsg 12-42
- GLS. *See* Global Language Support (GLS) environment.
- gls.h header file, definition of 1-22
- GL_DATE environment
 - variable 1-107, 6-6, 6-23, 6-48, 6-63
- GL_DATETIME environment
 - variable 1-107

H

- .h file extension 1-14
- Header file
 - automatic inclusion 1-26
 - collect.h 1-24
 - datetime.h 1-22
 - decimal.h 1-22
 - gls.h 1-22
 - ifxcexp.h 1-25
 - ifxgls.h 1-24
 - int8.h 1-24, 5-7
 - list of 2-11
 - location of 1-22
 - locator.h 1-23, 7-7
 - login.h 1-25, 1-104, 12-13
 - pthread.h 13-25
 - sqlca.h 1-23
 - sqlda.h 1-23, 1-26
 - sqlhdr.h 1-23, 1-26, 13-15
 - sqlapi.h 1-23
 - sqlproto.h 1-23, 1-25
 - sqlstype.h 1-23
 - sqltypes.h 1-23
 - sqlxtype.h 1-23
 - syntax for including 1-14, 1-26
 - system 1-14, 1-117
 - value.h 1-24
 - varchar.h 1-24
 - xa.h 1-24*See also* individual header file names.
- Hold cursor 9-21, 14-15, 14-18
- HOST network parameter 1-106, 12-13

Host variable

- array of 1-36
- arrays of 1-36
- as ANSI-style parameter 1-40
- as C structure 1-37
- as cursor name 14-19
- as function parameter 14-19, 16-62
- as input parameter 14-52
- as pointer 1-39
- as routine argument 14-45
- as SQL identifier 1-32
- assigning a value to 1-32
- based on definitions in C header files 1-66
- boolean data type 2-13
- case sensitivity 1-18
- char data type 2-19, 3-4, 6-35
- choosing data type for 1-28, 2-3, 2-10, 15-21, 17-15
- collection data type 9-4
- date data type 6-3
- datetime data type 6-29
- decimal data type 4-3
- declaring 1-27
- fetching DATETIME value 2-18
- fetching INTERVAL value 2-18
- fetching VARCHAR value 2-19
- fixchar data type 2-19, 3-5, 6-35
- fixed binary data type 10-18
- ifx_lo_t data type 7-7
- in embedded SQL 1-21
- in EXECUTE FUNCTION 14-49, 14-50
- in GLS environment 1-28
- in nonparameterized SELECT 14-43, 14-45
- in parameterized DELETE or UPDATE 14-52
- in parameterized EXECUTE FUNCTION 14-54
- in parameterized SELECT 14-54
- in typedef expressions 1-38
- indicator variable. *See* Indicator variable.
- initializing 1-29
- inserting DATETIME value 2-18
- inserting INTERVAL value 2-18
- inserting VARCHAR 2-19

- int8 data type 5-7
- interval data type 6-30, 6-33, 6-34
- loc_t data type 8-5
- lvarchar data type 3-9, 10-6
- naming 1-28
- row data type 9-48
- scope of 1-29
- string data type 2-19, 3-5, 6-35
- truncation of 1-44
- using data conversion 2-15
- var binary data type 10-23
- varchar data type 3-6
- with float values 5-6
- with null values 1-35, 2-24, 2-27, 10-41, 10-46
- HostInfoStruct structure
 - AskPassAtConnect field 12-13
 - definition of 12-13
 - Host field 12-13
 - InfxServer field 12-13
 - Options field 12-14
 - Pass field 12-13
 - Protocol field 12-14
 - Service field 12-14
 - User field 12-13
- Hyphen
 - as formatting character 5-28
 - double(-) 1-21

I

- I preprocessor option 1-52, 1-56, 1-62
- icheck preprocessor option 1-43, 1-51, 1-56, 1-62
- Icons
 - comment Intro-8
 - feature Intro-9
 - platform Intro-9
- .icp file extension 1-50
- IDATA descriptor field 15-8, 16-36
- ifdef preprocessor directive 1-12, 1-16
- ifndef preprocessor directive 1-12, 1-16
- ifxcexp.h header file, definition of 1-25
- ifxgls.h header file, definition of 1-24
- IFX_AUTOFREE environment variable 14-25
 - client only 14-37
- ifx_cl_card() library function 9-45
- ifx_dececv() library function 13-25, 13-31
- ifx_decfcvt() library function 13-25, 13-31
- IFX_DEFERRED_PREPARE environment variable 14-32
 - client only 14-37
- ifx_getcur_conn_name() library function 12-47
- ifx_getenv() library function 1-102
- ifx_getserial8() library function 5-10
- ifx_getversion utility 13-13
- ifx_int8add() library function 5-8, 5-12
- ifx_int8cmp() library function 5-8, 5-13
- ifx_int8copy() library function 5-8, 5-14
- ifx_int8cvasc() library function 5-8, 5-15
- ifx_int8cvdbl() library function 5-8, 5-16
- ifx_int8cvdec() library function 5-8, 5-17
- ifx_int8cvflt() library function 5-8
- ifx_int8cvint() library function 5-8, 5-18
- ifx_int8cvlong() library function 5-8, 5-19
- ifx_int8div() library function 5-8, 5-20
- ifx_int8mul() library function 5-8, 5-21
- ifx_int8sub() library function 5-8, 5-22
- ifx_int8toasc() library function 5-8, 5-23
- ifx_int8todbl() library function 5-8, 5-24
- ifx_int8todec() library function 5-8, 5-25
- ifx_int8toflt() library function 5-8
- ifx_int8toint() library function 5-8, 5-26
- ifx_int8tolong() library function 5-8, 5-27
- ifx_int8_t structure. *See* int8 data type.
- ifx_lo_alter() library function 7-36
- lightweight I/O 7-26
- ifx_lo_close() library function 7-29, 7-38
- lightweight I/O 7-26
- ifx_lo_col_info() library function 7-14, 7-39
- ifx_lo_copy_to_file() library function 7-41, 7-50, 9-45
- ifx_lo_copy_to_lo() library function 7-44
- ifx_lo_create() library function 7-14, 7-16, 7-24, 7-46
- duration of open 7-27
- lightweight I/O 7-25
- locks on smart large objects 7-27
- ifx_lo_create_spec_t structure. *See* LO-specification structure.
- ifx_lo_def_create_spec() library function 7-7, 7-14, 7-49
- ifx_lo_filename() library function 7-50
- ifx_lo_from_buffer() library function 7-19, 7-33, 7-52
- ifx_lo_open() library function 7-24, 7-52, 7-53
- duration of open 7-27
- lightweight I/O 7-25
- locks on smart large objects 7-27
- ifx_lo_readwithseek() library function 7-28, 7-57
- ifx_lo_read() library function 7-28, 7-55
- ifx_lo_release 7-52
- ifx_lo_release() library function 7-19, 7-34, 7-59
- ifx_lo_seek() library function 7-28, 7-29, 7-61
- ifx_lo_specget_estbytes() library function 7-9, 7-64
- ifx_lo_specget_extsz() library function 7-9, 7-66

ifx_lo_specget_flags() library function 7-11, 7-67
 ifx_lo_specget_maxbytes() library function 7-9, 7-69
 ifx_lo_specget_sbspace() library function 7-9, 7-71
 ifx_lo_specset_estbytes() library function 7-9, 7-73
 ifx_lo_specset_extsz() library function 7-9, 7-75
 ifx_lo_specset_flags() library function 7-11, 7-77
 ifx_lo_specset_maxbytes() library function 7-9, 7-79
 ifx_lo_specset_sbspace() library function 7-9, 7-80
 ifx_lo_spec_free() library function 7-16, 7-63
 ifx_lo_stat() library function 7-30, 7-81
 ifx_lo_stat_atime() library function 7-31, 7-83
 ifx_lo_stat_cspec() library function 7-31, 7-84
 ifx_lo_stat_ctime() library function 7-31, 7-86
 ifx_lo_stat_free() library function 7-32, 7-87
 ifx_lo_stat_mtime_sec() library function 7-31, 7-88
 ifx_lo_stat_refcnt() library function 7-31, 7-89
 ifx_lo_stat_size() library function 7-31, 7-91
 ifx_lo_stat_t structure. *See* LO-status structure.
 ifx_lo_t data type
 corresponding SQL data type 2-5, 2-11, 2-13
 declaration 7-7
 definition of 7-6
 fetching into 7-22
 inserting from 7-17
 use of fixed binary data type 7-7, 10-48
 See also BLOB data type; CLOB data type.
 ifx_lo_t structure. *See* LO-pointer structure.

ifx_lo_tell() library function 7-28, 7-29, 7-92
 ifx_lo_to_buffer() function 7-35
 ifx_lo_to_buffer() library function 7-19, 7-93
 ifx_lo_truncate() library function 7-95
 ifx_lo_writewithseek() library function 7-29, 7-98
 ifx_lo_write() library function 7-29, 7-96
 ifx_putenv() function 12-42, 14-37
 ifx_putenv() library function 1-100
 ifx_release() library function 7-34
 ifx_varlena_t structure 10-25
 See also var binary data type.
 ifx_var_alloc() function 10-33
 ifx_var_dealloc() function 10-35
 ifx_var_flag() function 10-37
 ifx_var_getdata() function 10-39
 ifx_var_getlen() function 10-40
 ifx_var_isnull() function 10-31, 10-41
 ifx_var_setdata() function 10-42
 ifx_var_setlen() function 10-44
 ifx_var_setnull() function 10-29, 10-46
 ILENGTH descriptor field 15-8, 16-36
 ILOGIN sample program 1-111
 ILogin sample program 1-6
 Implicit connection
 default 12-25
 definition of 12-18
 starting 12-19, 12-67
 terminating 12-40
 with sqlexit() 12-64
 Import library 1-95
 include directives, ESQ/L/C preprocessing of 1-9
 Include files
 header files as 1-22
 preprocessor directive for 1-13
 search path 1-63
 specifying search path 1-62
 syntax for 1-14, 1-26
 include preprocessor
 directive 1-12, 1-13, 1-26

#include preprocessor statement 1-26
 incvasc() library function 6-37, 6-65
 incvfmtasc() library function 6-37, 6-68
 INDICATOR descriptor field 15-8, 15-30, 16-30, 16-36
 INDICATOR keyword 1-45, 1-46, 14-44
 Indicator variable
 checking for missing indicator 1-62
 declaring 1-44
 definition of 1-42
 determining null data 1-36, 1-43, 1-44
 in GLS environment 1-45
 in INTO clause of EXECUTE 14-44
 indicating truncation 1-44, 3-11, 3-13, 3-15, 3-17, 6-35, 15-30
 inserting null values 1-43
 main variable 1-42
 specifying in SQL statement 1-45
 valid data types 1-44
 with opaque data type 15-30
 with sqlda structure 15-12, 17-8, 17-10, 17-30, 17-33
 with system-descriptor area 15-8, 16-30, 16-36
 See also Null value; Truncated value.
 Industry standards, compliance with Intro-15
 InetLogin structure
 application example 1-110, 12-11
 AskPassAtConnect field 1-107, 12-13, 12-14
 CC8BitLevel field 1-107
 Client_Loc field 1-105
 CollChar field 1-105
 connection information in 12-7
 ConRetry field 1-106
 ConTime field 1-106
 DbAlsBc field 1-107
 DbAnsiWarn field 1-105
 DbApiCode field 1-107
 DbAsciiBc field 1-108
 DbCentury field 1-108

- DbCodeset field 1-108
- DbConnect field 1-108
- DbCsConv field 1-108
- DbCsOverride field 1-108
- DbCsWidth field 1-108
- DbDate field 1-104
- DbFltMsk field 1-108
- DbLang field 1-105
- DbMoney field 1-104
- DbMoneyScale field 1-109
- DbNls field 1-105
- DbPath field 1-104
- DbSS2 field 1-109
- DbSS3 field 1-109
- DbTemp field 1-104
- DbTime field 1-104
- DB_Loc field 1-105
- DelimIdent field 1-106
- description of 1-103
- determining default database server 1-104
- EsqLMF field 1-107
- FetBuffSize field 1-107
- fields of 1-104
- GlDate field 1-107
- GlDateTime field 1-107
- Host field 1-106, 12-13
- InformixDir field 1-105
- InformixSqlHosts field 1-107
- InfxServer field 1-104, 12-10, 12-13, 12-94
- Lang field 1-105
- Lc_Collate field 1-105
- Lc_CType field 1-106
- Lc_Monetary field 1-106
- Lc_Numeric field 1-106
- Lc_Time field 1-106
- Options field 1-107, 12-14
- Pass field 1-106, 12-13
- precedence 1-111
- Protocol field 1-107, 12-14
- Service field 1-107, 12-14
- setting fields 1-109
- User field 1-106, 12-13
- with HostInfoStruct 12-13
- INFORMIX FIND ERROR utility 1-6

- Informix general library
 - actual name 13-7, 13-8
 - API version 13-14
 - choosing version of 13-5
 - compatibility issues 13-12
 - default version 13-9
 - description of 13-3
 - libasf 1-87, 13-5, 13-13
 - libgen 1-87, 13-5, 13-13
 - libgls 1-87, 13-5, 13-13
 - libos 1-87, 13-5, 13-13
 - libsql 1-87, 13-5, 13-13
 - libtgen 13-13
 - libtos 13-13
 - libtsql 13-13
 - linking 13-7, 13-9, 13-26
 - location of 13-6
 - naming 13-7, 13-8
 - obtaining version of 13-13
 - shared 13-3, 13-8, 13-10
 - static 13-3, 13-7
 - symbolic name 13-8
 - thread-safe 13-3
 - thread-safe shared 13-28
 - thread-safe static 13-28
- Informix library. *See* Informix general library.
- INFORMIX Registry subkey
 - connection information in 1-111
- Informix Server Information dialog box (Setnet32 utility) 12-94
- Informix SQL data types. *See* SQL data types.
- INFORMIXC environment variable 1-77
- INFORMIXCONRETRY environment variable 1-106
- INFORMIXCONTIME environment variable 1-106
- INFORMIXDIR
 - location of demonstration programs 1-6
 - value of in a Windows environment 1-6
- INFORMIXDIR environment variable 1-48, 1-63, 13-6, 13-13, B-2
 - in InetLogin structure 1-105
 - location of DLLs 1-88

- location of esqlauth.c file 12-15
- location of executables 1-6
- location of function libraries 1-6
- location of import library 1-83
- location of include files 1-83
- required 1-10
- INFORMIXDIR/bin directory B-2
- INFORMIX-OnLine Dynamic Server 11-15, 11-23
- INFORMIXSERVER environment variable 1-118, 12-7, 12-49, 12-67, 16-51, B-2
- and GetConnect() 12-94
- in HostInfoStruct structure 12-13
- in InetLogin structure 1-104
- Informix-specific status code. *See* SQLCODE variable.
- INFORMIXSQLHOSTS environment variable 1-107, 12-7, 12-12, B-2
- INFORMIX-Universal Server
 - checking for secondary mode 11-15, 11-23, 11-30, 12-27
- DATASKIP feature 1-43, 11-15, 11-23, 11-31
- determining type 11-15, 11-23
- interrupting 12-36, 12-53
- Input parameter
 - definition of 14-7, 14-51
 - in singleton SELECT 14-44
 - not known at compile time 15-23
 - specifying value in a system-descriptor area 16-10
 - specifying value in an sqlda structure 17-14
 - specifying values for EXECUTE FUNCTION statements 14-52, 14-54, 15-25
 - specifying values for non-SELECT statements 14-52, 15-25
 - specifying values for SELECT statements 14-52, 14-54, 15-25
 - specifying values for user-defined routines 14-45
 - with DESCRIBE 16-6
- Insert buffer 14-18, 14-20

- Insert cursor 17-14
 - definition of 14-18
 - description of 14-20
 - executing with sqlda structure 17-31
 - executing with system-descriptor area 16-35
 - for collection variable 9-21
 - input parameters in VALUES clause 16-11
 - insert buffer 14-20
 - required for 14-18
 - statements that manage 14-18
 - with system-descriptor area 16-11
- INSERT statement
 - and NOT FOUND
 - condition 11-21, 11-36
 - associated with a cursor 14-18
 - collection columns 9-40
 - Collection Derived Table
 - clause 9-18, 9-64
 - defined statement constant 15-16
 - determining estimated cost of 11-30
 - determining number of rows
 - inserted 11-8, 11-30
 - determining rowid 11-30
 - dynamic 14-18, 14-40, 14-52, 15-24
 - executing with sqlda structure 17-31
 - executing with system-descriptor area 16-30
 - failing to access rows 11-29
 - inserting CHAR data 3-12, 3-18
 - inserting collection data 9-18, 9-38, 9-41
 - inserting NCHAR data 3-12
 - inserting NVARCHAR data 3-15
 - inserting opaque-type data 10-13, 10-22, 10-28, 15-28
 - inserting row-type data 9-64, 9-74
 - inserting smart large-object data 7-17
 - inserting VARCHAR data 3-14
 - interrupting 12-33, 12-51
 - known at compile time 14-40, 14-52

- not known at compile time 15-23
- obtaining generated SERIAL
 - value 11-29
- parameterized 14-52
- row-type columns 9-71
- VALUES clause 1-38, 14-52
 - with DESCRIBE 15-15, 16-6, 17-7, 17-8
 - with null values 1-43
- Inserting
 - CHAR values 3-12
 - character data 3-11, 3-18
 - data conversion 2-15
 - DATETIME values 2-18, 6-33
 - from char host variable 3-12, 3-14, 3-17
 - from datetime host variable 6-33
 - from fixchar host variable 3-12, 3-14, 3-17
 - from fixed binary host variable 10-22
 - from fixed-size lvarchar 10-12
 - from ifx_lo_t host variable 7-17
 - from interval host variable 6-34
 - from string host variable 3-12, 3-14, 3-17
 - from var binary host variable 10-28
 - from varchar host variable 3-12, 3-14, 3-17
 - INTERVAL values 2-18, 6-34
 - into collection column 9-38, 9-40, 9-41
 - into collection variable 9-18
 - into row variable 9-64
 - into row-type column 9-71, 9-74
 - opaque-type values 15-28
 - smart large-object data 7-17
 - VARCHAR values 2-19, 3-14
- int data type 2-10
 - converting from decimal 4-44
 - converting from int8 5-26
 - converting to decimal 4-20
 - See also* short int data type; long int data type.
- INT data type. *See* INTEGER data type.

- INT8 data type
 - corresponding ESQL/C data type 2-5, 2-13, 5-4, 5-5
 - data conversion 5-8
 - declaring host variable for 5-7
 - defined constant 2-9
 - ifx_getserial8() 5-10
 - ifx_int8add() 5-12
 - ifx_int8cmp() 5-13
 - ifx_int8copy() 5-14
 - ifx_int8cvasc() 5-15
 - ifx_int8cvdbl() 5-16
 - ifx_int8cvdec() 5-17
 - ifx_int8cvint() 5-18
 - ifx_int8cvlong() 5-19
 - ifx_int8div() 5-20
 - ifx_int8mul() 5-21
 - ifx_int8sub() 5-22
 - ifx_int8toasc() 5-23
 - ifx_int8todbl() 5-24
 - ifx_int8todec() 5-25
 - ifx_int8toint() 5-26
 - ifx_int8tolong() 5-27
 - int8 structure shown 5-7
 - See also* int8 data type.
- int8 data type
 - addition 5-12
 - arithmetic operations 5-8
 - comparing 5-13
 - converting from decimal 5-17
 - converting from double 5-16
 - converting from integer 5-18
 - converting from long int 5-19
 - converting from text 5-15
 - converting to decimal 5-25
 - converting to double 5-24
 - converting to int 5-26
 - converting to long int 5-27
 - converting to text 5-23
 - copying 5-14
 - corresponding SQL data type 2-5
 - data conversion 5-8
 - declaration 5-7
 - defined constant 2-10
 - division 5-20
 - getting SERIAL8 values 5-10
 - multiplication 5-21

- subtraction 5-22
- See also* INT8 data type; SERIAL8 data type.
- int8.h header file 2-13, 5-7
- int8.h header file, definition of 1-24
- INTEGER data type
 - corresponding ESQ/C data type 2-4, 2-12, 5-4
 - data conversion 2-19, 2-20, 2-21
 - defined constant 2-8, 2-9, 15-20
 - See also* int data type.
- INTERVAL data type
 - ANSI-standard qualifiers 6-36
 - classes of 6-34, 6-36
 - corresponding ESQ/C data type 2-4, 2-12
 - data conversion 2-18, 6-33, 6-35, 6-37
 - declaring host variables for 6-30
 - defined constant 2-8
 - dynamically allocating structures for 17-12
 - fetching 2-18, 6-34
 - incvasc() 6-65
 - incvfmtasc() 6-68
 - inserting 2-18, 6-34
 - intoasc() 6-71
 - intofmtasc() 6-74
 - invdivdbl() 6-77
 - invdivinv() 6-80
 - invextend() 6-82
 - invmuldbl() 6-85
 - macros 6-31
 - precision of underlying decimal value 6-29
 - qualifiers 6-30, 6-36
 - role of datetime.h 1-22, 2-12
 - See also* interval data type.
- interval data type
 - arithmetic operations 6-36
 - corresponding SQL data type 2-4
 - data conversion 6-37
 - declaration 6-30
 - defined constant 2-10
 - definition of 6-28
 - fetching into 6-34
 - inserting from 6-34
 - role of datetime.h 2-12

- See also* INTERVAL data type; intrvl_t typedef.
- intoasc() library function 6-37, 6-71
- intofmtasc() library function 6-37, 6-74, 16-62
- intrvl structure 6-30
- intrvl_t typedef
 - defined constant 2-10
 - definition of 6-30, 6-34
 - setting qualifier for INTERVAL 17-12
- invdivdbl() library function 6-36, 6-77
- invdivinv() library function 6-36, 6-80
- invextend() library function 6-34, 6-82
- invmuldbl() library function 6-36, 6-85
- ISAM error code
 - and sqlerrd 11-29
 - and sqlerrd[1] 11-38
 - retrieving message text 11-11, 11-48, 11-51
 - See also* SQLCODE variable.
- ISDISTINCTBOOLEAN distinct-bit macro 15-31
- ISDISTINCTLVARCHAR distinct-bit macro 15-31
- ISDISTINCTTYPE distinct-bit macro 15-31
- ISO 8859-1 code set Intro-4
- ISO standards. *See* ANSI SQL standards.
- Item descriptor
 - DATA field. *See* DATA descriptor field.
 - definition of 15-5
 - EXTYPEID field 15-9
 - EXTYPELENGTH field 15-9
 - EXTYPENAME field 15-9
 - EXTYPEOWNERLENGTH field 15-9
 - EXTYPEOWNERNAME field 15-9
 - getting field values 16-10
 - IDATA field 15-8, 16-36
 - ILENGTH field 15-8, 16-36

- INDICATOR field 15-8, 15-30, 16-30, 16-36
- ITYPE field 15-8, 16-36
- LENGTH field. *See* LENGTH descriptor field.
- NAME field 15-8, 16-6, 16-21
- NULLABLE field 15-8, 16-6, 16-21
- PRECISION field 15-8, 16-6
- SCALE field 15-8, 16-6
- setting fields 16-9
- setting maximum number 16-5
- SOURCEID field 15-9, 15-31
- SOURCETYPE field 15-9, 15-31
- TYPE field. *See* TYPE descriptor field.
- See also individual field names.*
- Iterator function 14-50, 16-29, 17-29
- ITYPE descriptor field 15-8, 16-9, 16-36

L

- l preprocessor option 1-75
- l processor option 1-79, 1-87
- LANG environment variable 1-105
- Large object. *See* Simple large-object data type; Smart large-object data type.
- LC_COLLATE environment variable 1-105
- LC_CTYPE environment variable 1-106
- LC_MONETARY environment variable 1-106
- LC_NUMERIC environment variable 1-106
- LC_TIME environment variable 1-106
- ldchar() library function 3-30, 8-67
- LD_LIBRARY_PATH environment variable 13-9
- LENGTH descriptor field
 - after a DESCRIBE 16-6, 16-21
 - definition of 15-7
 - for varchar data 3-7
 - inserting opaque-type data 15-28
 - setting input parameter length 16-36, 16-41

- Less than (<) sign 5-28
- .lib file extension 1-79
- libasf Informix library 1-87, 13-5, 13-13
- libgen Informix library 1-87, 13-5, 13-13
- libgls Informix library 1-87, 13-5, 13-13
- libos Informix library 1-87, 13-5, 13-13
- Libraries
 - Informix. *See* Informix general library.
 - of ESQL/C functions 1-7
 - runtime search path 13-9
 - shared 13-3, 13-8, 13-10
 - static 13-3, 13-7
 - thread-safe 13-3
 - thread-safe shared 13-28
 - thread-safe static 13-28
- Library
 - creating dynamic thread 13-40
 - ESQL client-interface 1-84, 1-95
 - import 1-95
 - runtime 1-92
 - static-link 1-94, 1-95
 - system 1-79
- Library function. *See* ESQL/C library functions; Function library.
- libs preprocessor option 1-50
- libs processor option 1-54, 1-86
- libsql Informix library 1-87, 13-5, 13-13
- libtgen Informix library 13-13
- libtos Informix library 13-13
- libtsql Informix library 13-13
- Lightweight I/O
 - for smart large objects 7-25
 - specifying for all smart large objects 7-26
 - switching to buffered I/O 7-26
- Limits, Informix product Intro-14
- Line wrapping 1-73
- Linker
 - linking the ESQL client-interface DLL 1-95
 - marking beginning of arguments 1-79

- options invoked implicitly 1-82
- passing arguments to 1-87
- LIST data type
 - accessing 9-4
 - after a DESCRIBE 17-8
 - corresponding ESQL/C data type 2-5, 2-13
 - declaring host variable for 9-6
 - defined constant 2-9
 - definition of 9-4
 - inserting many elements into 9-20
- See also* Collection data type (SQL).
- LO file descriptor
 - deallocating 7-29
 - description of 7-20
 - ESQL/C functions for 7-20
- LO handle, deallocating 7-59
- local preprocessor option 1-52, 1-56, 1-64, 14-6
- Locale Intro-4
- Locating a simple large object
 - in a client-server environment 8-10
 - in a named file 8-26, 16-33, 16-34
 - in an open file 8-21
 - in memory 8-11, 16-55
 - locations for 8-10
 - on optical disc 8-40
 - with user-defined functions 8-31
- Locator structure
 - definition of 8-5, 8-6
 - fields of 8-8
 - lc_union structure 8-8, 8-11, 8-19, 8-32
 - loc_buffer field 8-11, 8-13
 - loc_buysize field 8-11, 8-12, 8-13, 16-55
 - loc_close field 8-32, 8-33, 8-34, 8-39
 - loc_currdata_p field 8-11
 - loc_fd field 8-19, 8-21, 8-25
 - loc_fname field 8-19, 8-26
 - loc_indicator field 8-9, 8-14
 - loc_loctype field. *See* loc_t.loc_loctype field.
 - loc_mflags field 8-11, 8-12
 - loc_mode field 8-19

- loc_oflags field. *See* loc_t.loc_oflags field.
- loc_open field 8-20, 8-31, 8-33, 8-34, 8-35
- loc_position field 8-19, 8-20
- loc_read field 8-31, 8-34, 8-36
- loc_size field. *See* loc_t.loc_size field.
- loc_status field 8-9, 8-13, 8-20, 8-26
- loc_type field 8-9
- loc_user_env field 8-32
- loc_write field 8-31, 8-33, 8-37
- loc_xfercount field 8-32
- memory buffer 8-12
- See also* loc_t data type.
- locator.h header file
 - access-mode constants 7-23
 - create-time constants 7-10
 - data types defined 2-11, 2-12, 2-13, 8-7
 - definition of 1-23
 - description of 7-7
 - field name shortcuts 8-11, 8-20
 - file-location constants 7-41, 7-44
 - file-open mode flags 8-20
 - location constants 8-10
 - LOC_ALLOC constant 8-13
 - LO-pointer structure 7-18
 - LO-specification structure 7-8
 - LO-status structure 7-30
 - whence constants 7-58, 7-62, 7-99
- LOCFILE location constant 8-10, 8-21
- LOCNAME location
 - constant 8-10, 8-26
- Locks, on smart large objects 7-26
- LOCMEMORY location
 - constant 8-10, 8-11, 16-55
- LOCUSER location constant 8-10, 8-31
- LOC_ALLOC locator constant 8-13
- LOC_APPEND locator mask 8-20, 8-22
- LOC_DESCRIPTOR locator
 - mask 8-40
- LOC_RONLY locator mask 8-20, 8-25, 8-34
- loc_t data type

- corresponding SQL data type 2-4, 2-5
- declaration 8-5
- defined constant 2-10
- definition of 8-5
- role of locator.h 2-11, 2-12
- See also* BYTE data type; Locator structure; TEXT data type.
- loc_t structure, with fetch array 15-38
- loc_t.loc_union structure. *See individual field names under* Locator structure.
- loc_t.loc_loctype field
 - assigning values to 8-10
 - definition of 8-8, 8-10
 - LOCFILE value 8-10, 8-21
 - LOCNAME value 8-10, 8-26
 - LOCMEMORY value 8-10, 8-11, 16-55
 - LOCUSER value 8-10, 8-31
- loc_t.loc_oflags field
 - file-open mode flags 8-20
 - setting for memory 8-12
 - setting for named file 8-26
 - setting for open file 8-21
 - setting for optical disc 8-40
 - using LOC_APPEND mask 8-20
 - using LOC_RDONLY mask 8-20, 8-34
 - using LOC_USEALL mask 8-30
 - using LOC_WONLY mask 8-20, 8-33
- loc_t.loc_size field
 - definition of 8-9
 - determining transfer size 8-30
 - indicating simple large-object size 8-13, 8-14
 - inserting a simple large object 8-25, 8-34
- LOC_USEALL locator mask 8-30
- LOC_WONLY locator mask 8-20, 8-33
- log preprocessor option 1-64
- log preprocessor option 1-52, 1-56
- login.h header file 1-25, 1-104, 12-13
- long int data type
 - converting from decimal 4-47
 - converting from int8 5-27
 - converting to decimal 4-22
 - converting to int8 5-19
 - converting to text 5-41
 - corresponding SQL data type 2-12, 5-4
 - data conversion 2-20
 - defined constant 2-10
 - See also* INTEGER data type; int data type.
- longjmp() system call 12-35, 12-63
- LO-pointer structure
 - creating 7-16
 - description of 7-18
 - ESQL/C functions for 7-19
 - in INSERT 7-17
 - in UPDATE 7-17
 - obtaining a valid 7-30
- LO-specification structure
 - allocating 7-7, 7-49
 - allocation extent size 7-66, 7-75
 - create-time flags 7-10, 7-67, 7-77
 - deallocating 7-16, 7-63
 - description of 7-8
 - disk-storage information 7-8
 - ESQL/C functions for 7-11
 - estimated size 7-64, 7-73
 - initializing 7-49
 - maximum size 7-69, 7-79
 - sbospace name 7-71, 7-80
 - setting 7-39, 7-84
 - storage characteristics 7-12
- LO-status structure
 - allocating 7-30
 - deallocating 7-32, 7-87
 - description of 7-30
- LO_APPEND access-mode constant 7-24, 7-25
- LO_APPEND file-location constant 7-44
- LO_BUFFER access-mode flag 7-25
- LO_CLIENT_FILE file-location constant 7-41, 7-44
- LO_KEEP_LASTACCESS_TIME create-time constant 7-10, 7-31
- LO_LOG create-time constant 7-10
- LO_NOBUFFER access-mode flag 7-25
- LO_NOKEEP_LASTACCESS_TIME create-time constant 7-10
- LO_NOLOG create-time constant 7-10
- LO_RDONLY access-mode constant 7-24, 7-25
- LO_RDWR access-mode constant 7-24, 7-25
- LO_SEEK_CUR whence constant 7-58, 7-62, 7-99
- LO_SEEK_END whence constant 7-58, 7-62, 7-99
- LO_SEEK_SET whence constant 7-58, 7-62, 7-99
- LO_SERVER_FILE file-location constant 7-41, 7-44
- LO_WRONLY access-mode constant 7-24, 7-25
- LVARCHAR data type
 - corresponding ESQL/C data type 2-5, 2-12
 - defined constant 2-9
 - distinct-bit constant 15-31
 - distinct-bit macro 15-31
 - See also* lvarchar data type.
- lvarchar data type
 - checking for null 10-14
 - corresponding SQL data type 2-5, 2-12, 2-13
 - CREATE OPAQUE TYPE statement 10-6
 - declaration 10-7
 - declaring 3-9
 - defined constant 2-11
 - definition of 3-4, 3-9
 - description of 3-9
 - fetching from column 3-17
 - fetching into 10-14
 - inserting from 3-18
 - inserting from, fixed size 10-12
 - inserting to column 3-17
 - of a fixed size 3-10
 - of fixed size 10-9
 - opaque type name 10-11

- pointer host variable 3-10, 10-9
 - allocating memory 10-10
 - and ifx_var() functions 10-11
 - functions for 10-32
 - ifx_var_alloc() function 10-33
 - ifx_var_dealloc() function 10-35
 - ifx_var_flag() function 10-37
 - ifx_var_getdata()
 - function 10-39
 - ifx_var_getlen() function 10-40
 - ifx_var_isnull() function 10-41
 - ifx_var_setdata() function 10-42
 - ifx_var_setlen() function 10-44
 - ifx_var_setnull() function 10-46
 - inserting from 10-15
 - selecting into 10-16
 - using 10-14
- selecting into, fixed size 10-14
- setting to null 10-13
- truncation, fixed size 3-10
- using 10-11
- with ANSI-compliant
 - database 3-18
- with DESCRIBE statement 16-7
- See also* LVARCHAR data type.
- lw preprocessor option 1-56

M

- Machine notes Intro-14
- Macro
 - for datetime and interval data
 - types 6-31
 - for distinct bit 15-31
 - for library versions 13-15
 - for thread-safe status
 - variables 13-27
 - for var binary data type 10-32
 - for varchar data type 3-7
- See also* Function library.
- Main variable 1-42
- Major features Intro-5
- malloc() system call 8-14, 17-10, 17-11, 17-25
- MAXVLEN varchar constant 3-7
- mc processor option 1-79
- Memory allocation, LO handle 7-59

- Memory management
 - ESQL/C functions 17-12
 - for sqlda structure 17-6
 - for system-descriptor area 16-5, 16-7
 - freeing resources 14-13, 16-12, 17-15
- Message chaining 12-42
- Message file, error
 - messages Intro-13
- Message request
 - definition of 12-33
 - interrupting 12-33
 - optimizing for cursor 14-22, 14-24, 14-30, 14-36
 - representing 12-89
 - with callback function 12-38
- Message transfers, optimized 12-41
- Microsoft compiler, invoking 1-79
- mi_lo_release() function 7-59
- Module-definition file 1-79
- MONEY data type
 - corresponding ESQL/C data
 - type 2-4, 2-12
 - data conversion 2-20, 5-35, 5-38, 5-41
 - defined constant 2-8
 - role of decimal.h 1-22, 2-12
 - scale and precision 15-8
 - See also* decimal data type.
- mserr preprocessor option 1-56, 1-74
- Multibyte filter 1-6
- MULTISET data type
 - accessing 9-4
 - after a DESCRIBE 17-8
 - corresponding ESQL/C data
 - type 2-4, 2-12
 - declaring host variable for 9-6
 - defined constant 2-9
 - definition of 9-4
 - inserting many elements
 - into 9-20
 - See also* Collection data type (SQL).

N

- N processor option 1-54
- NAME descriptor field 15-8, 16-6, 16-21
- Named row type
 - after a DESCRIBE 17-8
 - declaring host variable for 9-53
 - in a collection-derived table 9-54
 - in typed table 9-72
 - inserting into 9-71
 - literal values 9-75
 - updating 9-71
 - See also* Row type (SQL); Unnamed row type.
- Named row variable 9-53
 - typed 9-53
 - untyped 9-53
- NCHAR data type
 - corresponding ESQL/C data
 - type 2-4, 2-12, 3-3
 - defined constant 2-8
 - fetching 3-12
 - transferring with host
 - variables 3-11
- Network parameter
 - ASKPASSATCONNECT 1-107, 12-13, 12-14
 - HOST 1-106, 12-13
 - INFORMIXSERVER 1-104, 12-13
 - OPTIONS 1-107
 - PASSWORD 1-106, 12-13
 - precedence 1-111
 - PROTOCOL 1-107, 12-14
 - SERVICE 1-107, 12-14
 - setting with InetLogin 1-110
 - USER 1-106, 12-13
- nln preprocessor option 1-52, 1-56, 1-64
- Noncursor function
 - definition of 14-48
 - known at compile time 14-49
 - not known at compile time 16-23, 17-28
 - parameterized 14-52
 - with sqlda structure 17-28
 - with system-descriptor
 - area 16-23

- Non-default compilation of
 - ESQL / C programs
 - options for 1-67
 - Non-default order of compilation
 - for all ESQL / C files 1-67
 - Non-parameterized non-SELECT
 - statements 14-40
 - Non-parameterized SELECT
 - statements 14-43, 15-23, 16-13, 17-17
 - Non-SELECT statements
 - definition of 14-40
 - known at compile time 14-40, 14-41
 - nonparameterized 14-40
 - not known at compile time 15-25
 - parameterized 14-52, 15-25, 16-46, 17-43
 - preparing 14-41
 - with sqlda structure 17-43
 - with system-descriptor area 16-46
 - NOT FOUND condition
 - definition of 11-5
 - using SQLCODE 11-35
 - using SQLSTATE 11-20
 - using the WHENEVER statement 11-45
 - nowarn preprocessor option 1-52, 1-56, 1-65, 1-74
 - np preprocessor option 1-50
 - Null value
 - determining in dynamic SQL 15-8, 17-30
 - for simple large-object values 8-18, 8-25, 8-30
 - for simple-large-object values 8-9
 - ifx_var_isnull() 10-41
 - ifx_var_setnull() 10-29, 10-46
 - in aggregate function 11-15, 11-31
 - in host variables 1-35
 - inserting code for check for 1-62
 - inserting into table 1-43
 - returned in indicator 1-43
 - risnull() 1-36, 2-24
 - rsetnull() 1-36, 2-27
 - setting to 1-36, 2-27, 10-13, 10-22, 10-29, 10-46
 - testing for 1-36, 1-42, 2-24, 10-14, 10-23, 10-31, 10-41
 - See also* Indicator variable.
 - NULLABLE descriptor field 15-8, 16-6, 16-21
 - NUMERIC data type. *See* DECIMAL data type.
 - Numeric expressions
 - example formats 5-29
 - formatting 5-28
 - rfmtdec() function 5-35
 - rfmtdouble() function 5-38
 - rfmtlong() function 5-41
 - valid characters 5-28
 - nup preprocessor option 1-50
 - NVARCHAR data type
 - corresponding ESQL / C data type 2-4, 2-12, 3-3, 3-4, 3-5
 - data conversion 3-12
 - defined constant 2-8
 - fetching 3-15
 - transferring with host variables 3-11
-
- ## O
- .o file extension 1-75, 1-86
 - o preprocessor option 1-50, 1-54, 1-75, 1-76, 1-79, 1-84
 - .obj file extension 1-79, 1-80
 - On-line manuals Intro-13
 - onlycp preprocessor option 1-50
 - onspaces database utility 7-13
 - Opaque data type
 - after a DESCRIBE 17-8
 - as element type of collection 9-6
 - as field type of row 9-50
 - corresponding ESQL / C data type 2-5, 2-13
 - defined constant 2-9
 - definition of 10-3
 - dynamically executing 15-27
 - extended identifier 15-9, 15-12
 - in dynamic SQL 15-12
 - inserting 15-28
 - name of 15-9, 15-13
 - owner of 15-9, 15-13
 - predefined 10-48
 - truncation of data 15-30
 - See also* Fixed-length opaque data type; Varying-length opaque data type.
 - OPEN statement
 - and deferred PREPARE 14-34
 - executing a cursor 14-15, 14-18
 - executing with PREPARE 14-30
 - interrupting 12-33
 - optimizing 14-35
 - role in dynamic SQL 14-10
 - USING DESCRIPTOR
 - clause 17-4, 17-14, 17-34, 17-42
 - USING host_var clause 14-54
 - USING SQL DESCRIPTOR
 - clause 16-4, 16-11, 16-37, 16-44
 - with a SELECT statement 14-15
 - with an EXECUTE FUNCTION statement 14-15, 16-28, 17-28
 - with an INSERT statement 14-18
 - with deferred PREPARE 14-30
 - with OPTOFC and Deferred PREPARE features 14-38
 - open() system call 8-19
 - OPEN, FETCH, and CLOSE
 - (OPTOFC) feature
 - restrictions 14-36
 - Optimized Message Transfers 12-41
 - Optimized message transfers
 - enabling 12-42
 - error handling 12-44
 - reasons to disable 12-43
 - restrictions on 12-41
 - Optimizing
 - OPEN, FETCH, and CLOSE statements 14-35
 - OPTIONS network
 - parameter 1-107
 - OPTMSG environment
 - variable 12-42
 - setting 12-42
 - OptMsg global variable 12-42
 - setting 12-43
 - OPTOFC environment
 - variable 14-37
 - client only 14-37

- OPTOFC feature
 - and static cursors 14-36
 - enabling 14-37
 - with AUTOFREE and Deferred PREPARE features 14-39
 - with Deferred PREPARE feature 14-38

P

- p preprocessor option 1-54
- PARAMETER keyword 1-40, 14-19, 16-62
- Parameterized non-SELECT statements 14-52, 15-25, 16-46, 17-43
- Parameterized SELECT statements 14-54, 15-25, 16-36, 16-43, 17-32, 17-41
- Parenthesis symbol 5-29
- PASSWORD network parameter 1-106, 12-13
- PATH environment variable 1-96, B-2
 - required 1-10
- Period (.) symbol 5-28, 9-74
- Pi processor option 1-92
- Platform icons Intro-9
- Plus (+) sign 5-28
- Pointer, as host variable 1-39
- Pound (#) sign 5-28
- PRECISION descriptor field 15-8, 16-6
- PREPARE statement
 - and sqlca.sqlerrd[0] 11-29, 14-9
 - and sqlca.sqlerrd[3] 11-30, 14-9
 - and sqlca.sqlerrd[4] 11-30, 11-39, 14-9
 - deferring execution of 14-30
 - exceptions returned 14-9
 - for collection variables 14-8
 - in thread-safe application 13-22
 - role in dynamic SQL 14-5
 - SQLCODE value 11-39, 14-9
 - with DATABASE statement 12-18
 - with EXECUTE 14-41

- with EXECUTE
 - FUNCTION 16-22, 16-26, 17-26
- with EXECUTE
 - PROCEDURE 14-47
- with EXECUTE...INTO 14-43, 14-49
- with INSERT 16-29
- with OPTOFC and Deferred PREPARE features 14-38
- with SELECT 16-13
- Preprocessor
 - case sensitivity 1-18
 - conditional compilation 1-16, 8-48
 - definitions 1-14, 1-60
 - directives. *See* Preprocessor directive.
 - generating thread-safe code 13-17
 - header files 1-26
 - include files 1-13
 - line numbers 1-64
 - options. *See* Preprocessor option.
 - redirecting errors 1-64
 - search path for included files 1-63
 - stage 1 1-13, 1-61
 - stage 2 1-16, 1-66
 - syntax 1-48
 - version information 1-54, 1-57
- Preprocessor directive
 - define 1-14, 1-60
 - definition of 1-12
 - elif 1-16
 - else 1-16
 - endif 1-16
 - ifdef 1-16
 - ifndef 1-16
 - include 1-13, 1-26
 - undef 1-14, 1-60
- Preprocessor option
 - ansi 1-51, 1-55, 12-17
 - ccargs 1-50, 1-76
 - cp 1-49
 - e 1-49, 1-53, 1-58
 - ED 1-51, 1-55, 1-60
 - EU 1-51, 1-55, 1-60
 - G 1-51, 1-56, 1-64
 - g 1-51, 1-56, 1-64
 - I 1-52, 1-56, 1-62

- icheck 1-51, 1-56, 1-62
- l 1-75
- l for dynamic thread library 13-51
- libs 1-50
- local 1-52, 1-56, 1-64, 14-6
- log 1-52, 1-56, 1-64
- lw 1-56
- mserr 1-56, 1-74
- nln 1-52, 1-56, 1-64
- nowarn 1-52, 1-56, 1-65, 1-74
- np 1-50
- nup 1-50
- o 1-50, 1-54, 1-75, 1-76, 1-79, 1-84
- onlycp 1-50
- p 1-54
- static 1-50, 1-76, 13-6, 13-7, 13-28
- those affecting linking 1-86
- those affecting preprocessing 1-58
- those for Informix libraries 13-6
- thread 1-50, 1-57, 1-59, 1-76, 13-6, 13-17, 13-27
- ts 1-57
- V 1-50, 1-58
- xopen 1-57, 1-65, 15-19, 15-21
- Preprocessor, ESQL
 - stage 1 1-9
- Preprocessor, ESQL / C
 - stage 2 1-9
- Printed manuals Intro-13
- Process
 - child 12-32, 12-56
 - parent 11-33, 12-32, 12-56
- Processor
 - associating options with files 1-58
 - creating a response file 1-82
 - naming executable file 1-79
 - syntax 1-77, 1-78, 1-90
 - using a project file 1-81
 - Windows options 1-91
- Processor option
 - c 1-80
 - cc 1-78, 1-80
 - cpu 1-91
 - f 1-54, 1-81
 - l 1-79, 1-87
 - libs 1-54, 1-86
 - N 1-54

- Pi 1-92
- placement of 1-58
- r 1-79, 1-88
- rt 1-92
- runtime 1-84, 1-92
- Sc 1-85, 1-93
- ss 1-85, 1-92, 1-93
- subsystem 1-85, 1-93
- Sw 1-85, 1-93
- target 1-83, 1-93, 1-96
- V 1-54
- wd 1-83, 1-93, 1-96
- we 1-83, 1-93
- Windows options 1-90
- Program
 - checking library version 13-14
 - commenting 1-21
 - compiling 1-11, 1-80, 13-12
 - including files 1-13, 1-26
 - linking 1-11, 1-86, 13-7, 13-9, 13-12, 13-26
 - message request 12-33, 12-38, 12-89, 14-22, 14-24, 14-30, 14-36
 - naming the executable file 1-54, 1-76, 1-79
 - preprocessing 1-12, 1-58
 - running 1-11
 - suppressing compilation 1-58
 - suppressing linking 1-80
 - See also* Sample program.
- Project file 1-54, 1-81
- PROTOCOL network
 - parameter 1-107, 12-14
- pthread.h DCE header file 13-25
- pthread_lock_global_np() DCE
 - function 13-25
- pthread_yield() DCE
 - function 13-26
- PUT statement
 - inserting a row 14-18
 - inserting into a collection variable 9-22
- USING DESCRIPTOR
 - clause 17-4, 17-14, 17-31
- USING SQL DESCRIPTOR
 - clause 16-5, 16-11, 16-35
- putenv() system call 12-42, 14-37

Q

- Question mark (?) 14-51
 - wildcard in smart large-object filenames 7-42
- Quotation marks
 - escaping 1-19, 9-43, 9-76
 - See also* Single quotes; Double quotes.

R

- r processor option 1-79, 1-88
- .rc file extension 1-79, 1-86, 1-88
- rdatestr() library function 6-6
- rdayofweek() library function 6-8
- rdefmtdate() library function 6-10, 6-37
- rdownshift() library function 3-32
- REAL data type. *See* SMALLFLOAT data type.
- Reference count 7-31, 7-89
- regcopy utility 1-6
- Registering dynamic thread functions 13-49
- Registry
 - connection information in 12-7
 - InformixServer 12-94
 - in-memory copy 1-111
 - precedence 1-111
- Release notes Intro-14
- ReleaseConnect() library
 - function 12-92, 12-97
- .res file extension 1-79, 1-86, 1-88
- Resource compiler
 - default options 1-83
 - passing arguments to 1-79, 1-88
 - specifying compiled resource file 1-79
 - specifying file to compile 1-79
- Resource file 1-79, 1-86
- Response file 1-54, 1-82
- Restrictions
 - on optimized message transfers 12-41
 - on OPTOFC feature 14-36
- Retrieving an error message 11-48
- rfmtdate() library function 6-14, 6-37, 16-62

- rfmtdec() library function 5-35, 16-62
- rfmtdouble() library function 5-38
- rfmtlong() library function 5-41
- rgetlmsg() library function 11-48
- rgetmsg() library function 11-51
- risnull() library function 1-43, 2-24
- rjulmdy() library function 6-17
- rleapyear() library function 6-19
- rmdyjul() library function 6-21
- ROLLBACK WORK
 - statement 12-29, 15-17
- Row constructor 9-70
- row data type (ESQL/C)
 - accessing a typed table 9-72
 - allocating memory for 9-57
 - client-side 9-56
 - Collection Derived Table clause 9-59
 - corresponding SQL data type 2-5, 2-13
 - deallocating memory for 9-57
 - declaration 9-49
 - defined constant 2-11
 - deleting from 9-68
 - fetching from 9-64
 - fetching into 9-62
 - field names 9-68
 - field values 9-69
 - initializing 9-62
 - inserting into 9-64
 - literal values 9-69
 - named row variable 9-53
 - nested rows 9-66
 - operating on 9-58
 - typed row variable 9-50
 - untyped row variable 9-52
 - updating 9-66
 - See also* Row type (SQL).
- Row data type(ESQL/C)
 - with a collection 9-26
- Row type (SQL)
 - accessing 9-48
 - as element type of collection 9-6
 - as field type of row 9-50
 - constructed rows 9-70
 - corresponding ESQL/C data type 2-5, 2-13
 - declaring host variables for 9-49

- defined constant 2-9
- definition of 9-48
- deleting 9-68, 9-76
- dot notation 9-66, 9-74
- extended identifier 15-9, 15-12
- fetching 9-62, 9-64
- in dynamic SQL 15-12
- inserting into 9-64, 9-71, 9-74
- literal values 9-43, 9-69, 9-74, 9-75
- nested 9-70
- owner of 15-9, 15-13
- selecting from 9-73
- typed table 9-72
- updating 9-66, 9-71, 9-74
- See also* row data type (ESQL/C).
- rsetnull() library function 2-14, 2-27
- rstod() library function 3-34
- rstoi() library function 3-36
- rstol() library function 3-38
- rstodate() library function 6-23
- rt processor option 1-92
- rtdat() library function 6-26
- rtypalign() library function 2-30, 17-12, 17-23
- rtypsize() library function 2-33, 17-12, 17-23
- rtypname() library function 2-36
- rtypwidth() library function 2-39
- Running C preprocessor first
 - options for 1-67
- Runtime environment 1-97
- Runtime errors
 - definition of 11-5
 - in user-defined routines 11-18, 11-25
 - Informix-specific messages 11-25
 - using rgetlmsg() 11-48
 - using rgetmsg() 11-51
 - using sqlca structure 11-38
 - using SQLSTATE variable 11-24
 - using the WHENEVER statement 11-45
 - See also* Error handling; Warnings.
- runtime processor option 1-84, 1-92
- Runtime routines, exporting 1-89
- rupshift() library function 3-40

S

Sample program

- bycmpr 3-22
- bycopy 3-24
- byfill 3-27
- byleng 3-29
- create_clob 7-100
- decadd 4-9
- deccmp 4-11
- deccopy 4-13
- deccvasc 4-6, 4-16
- deccvdbl 4-18
- deccvint 4-20
- deccvlong 4-22
- decdiv 4-24
- dececv 4-27
- decfcvt 4-30
- decml 4-32
- decround 4-35
- decsub 4-37
- dectoasc 4-40
- dectodbl 4-42
- dectoint 4-45
- dectolong 4-47
- dectrunc 4-50
- demo1 1-116
- demo2 14-54
- demo3 16-15, 16-37, 17-18
- demo4 14-54, 16-15, 17-34
- dispcat_pic 8-44, 11-59
- dtaddinv 6-41
- dtcurrent 6-43
- dtcvasc 6-46
- dtcvfmtasc 6-49
- dtextend 6-52
- dtsub 6-55
- dtsubinv 6-58
- dttoasc 6-60
- dttofmtasc 6-63
- dyn_sql 11-59
- ESQLAUTH 12-15
- getcd_me 8-15
- getcd_nf 8-27
- getcd_of 8-22
- ILOGIN 1-111
- incvasc 6-66
- incvfmtasc 6-69
- intoasc 6-73

- intofmtasc 6-75
- invdivdbl 6-78
- invdivinv 6-81
- invextend 6-83
- invmuldbl 6-86
- ldchar 3-31
- rdatestr 6-7
- rdayofweek 6-8
- rdefmtdate 6-12
- rdownshift 3-33
- rfmtdate 6-15
- rfmtdec 5-36
- rfmtdouble 5-39
- rfmtlong 5-42
- rgetlmsg 11-49
- rgetmsg 11-52
- risnull 2-25
- rjulmdy 6-17
- rleapyear 6-19
- rmidyjul 6-22
- rsetnull 2-27
- rstod 3-34
- rstoi 3-36
- rstol 3-38
- rstodate 6-24
- rtdat 6-26
- rtypalign 2-31
- rtypsize 2-34
- rtypname 2-36
- rtypwidth 2-40
- rupshift 3-41
- sqgetdbs 12-49
- sqldetach 12-57, 12-58
- stcat 3-43
- stchar 3-45
- stcmpr 3-47
- stcopy 3-48
- stleng 3-51
- timeout 12-69
- upcdcd_me 8-17
- upcdcd_nf 8-28
- upcdcd_of 8-23
- varchar 3-7
- WDEMO 1-96
- See also* Program.
- Sbospace
 - definition of 7-9
 - getting name of 7-71
 - on optical disc 7-32

- running out of space 7-96, 7-99
- setting 7-80
- storage characteristics for 7-13
- SBSPACENAME configuration
 - parameter 7-13
- Sc processor option 1-85, 1-93
- SCALE descriptor field 15-8, 16-6
- Scope of
 - cursor names 1-64, 13-24
 - host variables 1-29
 - preprocessor definitions 1-14
 - statement identifiers 1-64, 13-22
- Scroll cursor 14-15
- Select cursor
 - definition of 14-16
 - fetch buffer 14-20
 - for collection variable 9-27
 - statements that manage 14-15
 - using 14-45
- SELECT statement
 - and NOT FOUND
 - condition 11-20, 11-35
 - associated with a cursor 14-14, 16-4, 17-4
 - checking for truncation 11-31
 - Collection Derived Table
 - clause 9-24, 9-64
 - defined statement constant 15-16
 - determining estimated cost
 - of 11-30
 - determining rowid 11-30, 11-35
 - executing a singleton
 - SELECT 14-43
 - failing to access rows 11-29
 - fetching CHAR data 3-11, 3-18
 - fetching collection data 9-16, 9-24, 9-37, 9-41
 - fetching DATETIME data 6-33
 - fetching INTERVAL data 6-34
 - fetching opaque-type data 10-14, 10-23, 10-30
 - fetching row-type data 9-62, 9-64, 9-73
 - fetching smart large-object
 - data 7-22
 - fetching VARCHAR data 3-13
 - identifying variable
 - mismatch 11-15, 11-23, 11-31
 - in ANSI-compliant database 3-18

- interrupting 12-33, 12-51
- INTO host_var clause 11-31, 14-6
- INTO TEMP clause 11-21, 11-36
- known at compile time 14-43
- nonparameterized 14-43, 15-23, 16-13, 17-17
- not known at compile time 15-23, 15-25
- parameterized 14-52, 14-54, 15-25, 16-36, 16-43, 17-32, 17-41
- singleton 14-10, 14-16, 14-43, 16-4, 16-12, 17-4
- with aggregate functions 11-31
- with DESCRIBE 15-15, 16-6, 17-7, 17-8
- with sqlda structure 17-17, 17-32
- with system-descriptor
 - area 16-13, 16-36, 16-43, 17-41
- See also* Cursor (database); Fetching.
- Sequential cursor 9-21, 9-27, 14-15, 14-18
- SERIAL data type
 - corresponding ESQ/C data type 2-4, 2-12, 5-4
 - data conversion 2-19
 - defined constant 2-8
 - obtaining value after INSERT 11-29
 - using typedefs 1-38
 - See also* int data type.
- SERIAL8 data type
 - corresponding ESQ/C data type 2-5, 2-13, 5-4
 - declaring host variable for 5-7
 - defined constant 2-9
 - ifx_getserial8() 5-10
 - See also* int8 data type.
- SERVICE network
 - parameter 1-107, 12-14
- SET AUTOFREE statement 14-25, 14-26
- SET AUTOFREE (SQL)
 - statement 14-25
 - setting 14-26
- SET CONNECTION
 - statement 12-92, 12-99
 - and explicit connections 12-17

- determining database server
 - features 12-26
- making connection
 - dormant 13-18, 13-21, 13-23
- managing connections across threads 13-20
- switching to a dormant
 - connection 12-28, 13-18
- with an active transaction 12-29
- SET data type
 - accessing 9-4
 - after a DESCRIBE 17-8
 - corresponding ESQ/C data type 2-5, 2-13
 - declaring host variable for 9-6
 - defined constant 2-9
 - definition of 9-4
 - inserting many elements
 - into 9-20
 - See also* Collection data type (SQL).
- SET DEFERRED PREPARE
 - statement 14-32
- SET DESCRIPTOR statement
 - setting COUNT field 16-9, 16-36
 - setting fields 16-4, 16-9, 16-36
 - VALUE keyword 16-36
 - See also* System-descriptor area.
- SetConnect() library
 - function 12-92, 12-99
- setjmp() system call 12-35
- Setnet32 utility
 - description 1-6
 - determining default database server 12-94
 - use of ixreg.dll 1-88
- Shared library
 - creating for dynamic thread functions 13-50
- short int data type
 - corresponding SQL data type 2-12, 5-4
 - data conversion 2-20
 - defined constant 2-10
 - See also* SMALLINT data type; int data type.
- Signal handling
 - of ESQ/C library 12-65
 - of SIGINT 12-34

Signal-handler function 12-34, 12-35

Signals

SIGCHLD 12-66

SIGINT 12-34

signal() system call 12-35

Simple large objects, and a fetch array 15-33

Simple-large-object data type compared with smart large objects 8-4

declaring host variable for 8-5

definition of 8-3

format on disk 8-40

inserting 8-17, 8-23, 8-28, 8-34, 16-33

locating

See Locating a simple large object.

null values 8-9

on optical disc 8-40

programming with 8-5

selecting 8-15, 8-22, 8-27, 8-33, 16-63

subscripting 8-6

See also BYTE data type; TEXT data type.

Simple-large-object descriptor 8-40

Single quotes (' ')

delimiting strings 1-32

escaping 1-19

in a literal collection 9-43

in a literal row 9-76

in a prepared statement string 14-5

in a quoted string 1-20

.sl file extension 13-13

SMALLFLOAT data type

corresponding ESQL/C data type 2-4, 2-12, 5-5

data conversion 2-19, 2-20, 2-21

defined constant 2-8

number of decimal digits 2-18, 5-6

See also float data type.

SMALLINT data type

corresponding ESQL/C data type 2-4, 2-12, 5-4

creating a typedef for 1-38

data conversion 2-19, 2-20, 2-21

defined constant 2-8, 2-9, 15-20

See also short int data type.

Smart large object, permanent 7-59

Smart-large-object data type

access modes 7-23

accessing 7-22

advantages 8-4

allocation extent size 7-66, 7-75

altering 7-36

closing 7-29, 7-38

compared with simple large objects 8-4

copying from user-defined

buffer 7-52

copying to a file 7-41

copying to a user-defined buffer 7-93

create-time flags 7-67, 7-77

creating 7-21, 7-46

declaring host variable for 7-7

definition of 7-5, 9-3

determining storage

characteristics of 7-47

duration of open 7-27

ESQL/C functions for 7-33

estimated size 7-9, 7-64, 7-73

file position 7-61, 7-92

format on disk 7-32

getting size of 7-31, 7-91

hexadecimal identifier for 7-16, 7-42

ifx_lo_alter() 7-36

ifx_lo_close() 7-38

ifx_lo_col_info() 7-39

ifx_lo_copy_to_file() 7-41, 9-45

ifx_lo_copy_to_lo() 7-44

ifx_lo_create() 7-46

ifx_lo_def_create_spec() 7-49

ifx_lo_filename() 7-50

ifx_lo_from_buffer() 7-52

ifx_lo_open() 7-52, 7-53

ifx_lo_readwithseek() 7-57

ifx_lo_read() 7-55

ifx_lo_release() 7-59

ifx_lo_seek() 7-61

ifx_lo_specget_estbytes() 7-64

ifx_lo_specget_extsz() 7-66

ifx_lo_specget_flags() 7-67

ifx_lo_specget_maxbytes() 7-69

ifx_lo_specget_sbspace() 7-71

ifx_lo_specset_estbytes() 7-73

ifx_lo_specset_extsz() 7-75

ifx_lo_specset_maxbytes() 7-79

ifx_lo_specset_sbspace() 7-80

ifx_lo_spec_free() 7-63

ifx_lo_stat() 7-81

ifx_lo_stat_atime() 7-83

ifx_lo_stat_cspec() 7-84

ifx_lo_stat_ctime() 7-86

ifx_lo_stat_free() 7-87

ifx_lo_stat_mtime_sec() 7-88

ifx_lo_stat_refcnt() 7-89

ifx_lo_stat_size() 7-91

ifx_lo_tell() 7-92

ifx_lo_to_buffer() 7-93

ifx_lo_truncate() 7-95

ifx_lo_writewithseek() 7-98

ifx_lo_write() 7-96

inserting 7-17

lightweight I/O 7-25

LO file descriptor 7-20

locking 7-26, 7-38, 7-47, 7-53

locks 7-26

LO-pointer structure 7-18

LO-specification structure 7-8

LO-status structure 7-30

maximum size 7-9, 7-69, 7-79

modifying 7-28

obtaining status of 7-29, 7-81

on optical disc 7-32

opening 7-23, 7-47, 7-53

programming with 7-6

reading from 7-28, 7-55, 7-57

reference count 7-31, 7-89

sample program 7-100

sbspace 7-71, 7-80

selecting 7-22

storage characteristics 7-8, 7-12

storing 7-17

temporary smart large objects 7-59

temporary, releasing

resources 7-59

time of last access 7-31, 7-83

time of last change in status 7-31, 7-86

- time of last modification 7-31, 7-88
- truncating 7-95
- updating 7-17
- writing to 7-29, 7-96, 7-98
- See also* BLOB data type; CLOB data type; ifx_lo_t data type.
- .so file extension 13-8, 13-13
- Software dependencies Intro-4
- SOURCEID descriptor field 15-9, 15-31
- SOURCETYPE descriptor field 15-9, 15-31
- source.ec file, argument to preprocessor 1-50
- source.ecp file, argument to the preprocessor 1-50
- SPL function
 - cursor function 14-42
 - definition 14-47
 - executing dynamically 14-50, 16-23, 16-29, 17-28, 17-29
 - See also* SPL routine; User-defined function.
- SPL procedure 14-47
 - See also* SPL routine; User-defined procedure.
- SPL routine 12-33, 14-47
- sqgetdb() library function 12-48
- SQL data types
 - BLOB 2-5, 2-11, 7-5
 - BOOLEAN 2-5, 2-13
 - BYTE 2-4, 2-11, 8-3
 - CHAR 2-4, 2-11, 3-3
 - CLOB 2-5, 2-13
 - collections 9-4
 - DATE 2-4, 2-11, 6-3
 - DATETIME 2-4, 2-11, 6-28
 - DECIMAL 2-4, 2-12, 4-3
 - defined constants for 2-8, 15-20
 - distinct 15-27
 - FLOAT 2-4, 2-12, 5-5
 - INT8 2-5, 2-13, 5-4, 5-5, 5-7
 - INTEGER 2-4, 2-12, 5-4
 - INTERVAL 2-4, 2-12, 6-28
 - LIST 2-5, 2-13, 9-4
 - LVARCHAR 2-5, 2-12
 - MONEY 2-4, 2-12
 - MULTISET 2-4, 2-12, 9-4
 - named row type 9-48
 - NCHAR 2-4, 2-12, 3-3
 - NVARCHAR 2-4, 2-12, 3-3, 3-4, 3-5
 - opaque 2-5, 2-13, 10-3
 - row types 2-5, 9-48
 - SERIAL 2-4, 2-12, 5-4
 - SERIAL8 2-5, 2-13, 5-4, 5-7
 - SET 2-5, 2-13, 9-4
 - SMALLFLOAT 2-4, 2-12, 5-5
 - SMALLINT 2-4, 2-12, 5-4
 - TEXT 2-4, 2-12, 8-3
 - unnamed row type 2-13, 9-48
 - VARCHAR 2-4, 2-12, 3-3
 - X/Open defined constants 2-9
 - See also individual data type names.*
- SQL identifier 1-32, 14-51
 - See also* Delimited identifier.
- SQL keyword protection
 - against interpretation by C preprocessor 1-72
 - relation to the dollar (\$) sign 1-72
- SQL statement
 - CLOSE DATABASE 12-98
 - CONNECT 12-11, 12-92
 - DATABASE 12-95
 - DISCONNECT 12-92, 12-97
 - SET CONNECTION 12-92, 12-99
- SQL statements
 - case sensitivity 1-18
 - cursor-management statements 14-15, 14-18
 - defined constants for 11-33, 15-16
 - dynamic. *See* Dynamic SQL.
 - for dynamic SQL 14-11, 14-15, 14-18, 16-4, 17-4
 - interruptable 12-33
 - obtaining diagnostic information 11-4
 - parameterized 14-51
 - static 11-30, 14-3
- sqlauth() authentication function 1-88, 1-107, 12-13
- SQLBOOL data-type constant 2-9
 - See also* BOOLEAN data type.
- sqlbreakcallback() library function 12-31, 12-36, 12-53
- sqlbreak() library function 12-33, 12-51, 12-69
- SQLBYTES data-type constant 2-8
 - See also* BYTE data type.
- sqlca structure 1-115
 - and DESCRIBE 15-22
 - and PREPARE 14-9
 - checking for exceptions 11-27
 - definition of 11-28
 - determining database server features 11-30, 11-37, 12-27
 - in thread-safe code 13-17, 13-27
 - indicating truncation 1-44, 2-16
 - relation to SQLCODE status variable 11-32
 - retrieving error message text 11-47
 - sqlcode 11-29
 - See also* SQLCODE variable.
- sqlcrrd array. *See* sqlca.sqlcrrd array.
- sqlerrm 11-29, 11-38, 11-49
- sqlerrp 11-29
- sqlwarn structure. *See* sqlca.sqlwarn structure.
- using the WHENEVER statement 11-45
- warning values 11-37
- sqlca.h header file
 - constant definitions 11-36
 - definition of 1-23
 - structure definition 11-28, 13-28
 - variable definitions 11-12, 11-32, 13-28
- sqlca.sqlcode value. *See* SQLCODE variable.
- sqlca.sqlcrrd array
 - definition of 11-29, 11-30
- sqlcrrd[0] 11-34, 14-9
- sqlcrrd[1] 11-29, 11-34, 11-38, 11-48, 11-51
- sqlcrrd[2] 11-35, 11-38
- sqlcrrd[3] 11-34, 14-9
- sqlcrrd[4] 11-38, 11-39, 14-9
- sqlcrrd[5] 11-35
- sqlca.sqlwarn structure
 - definition of 11-30, 11-31, 11-37
- sqlwarn0 11-30, 11-31
- sqlwarn1 3-11, 3-17, 6-35, 11-30, 11-31, 12-27
- sqlwarn2 11-30, 11-31, 12-27

- sqlwarn3 11-31, 12-27
- sqlwarn4 11-31, 12-27, 15-22
- sqlwarn5 11-31
- sqlwarn6 11-31, 12-27
- sqlwarn7 11-30, 11-31, 12-27
- SQLCHAR data-type constant 2-8
See also CHAR data type.
- SQLCODE value 1-115, 12-95, 12-98, 12-100
- SQLCODE variable
 - after a DESCRIBE statement 11-33, 15-16
 - after a GET DIAGNOSTICS statement 11-13
 - after a PREPARE statement 11-39
 - after a simple large-object access 8-9
 - after an ALLOCATE COLLECTION 9-12
 - after an ALLOCATE ROW 9-57
 - after an EXECUTE statement 11-40
- and sqlerrd 11-29
- definition of 11-32
- in diagnostics area 11-10, 11-13
- in thread-safe code 13-17, 13-27
- indicating an interrupt 12-51
- indicating runtime errors 11-38
- relation to sqlca structure 11-32
- result codes 11-29, 11-34
- retrieving error message text 11-48, 11-51
- See also* ISAM error code; sqlca structure; SQLSTATE variable.
- sqlc field. *See* sqlc.sqlc field.
- sqlcda
 - and a fetch array 15-34
- sqlcda structure
 - allocating memory for 13-17, 17-6, 17-10
 - assigning values to 17-13
 - data type lengths 3-7
 - declaring 17-5
 - definition of 15-10
 - desc_name field 15-14
 - desc_next field 15-14
 - desc_occ field 15-14

- examples 17-18, 17-34
- fetching rows into 17-14
- fields of 15-10, 15-11, 15-12
- for columns of a SELECT 17-17
- for columns of an INSERT 17-30
- for distinct-type columns 15-31
- for input parameters 17-32, 17-43
- for opaque-type columns 15-28, 15-30
- for return values of a user-defined function 17-26
- freeing memory for 17-15
- getting field values 17-13
- initializing 17-7
- interrupting database server 12-51
- managing 17-4
- obtaining values from 17-13
- setting fields 17-13
- specifying input parameter values for 17-14
- sqlc field. *See* sqlc.sqlc field.
- sqlcvar structure. *See* sqlc.sqlcvar structure.
- uses of 17-16
- using an indicator variable 15-12
- sqlcdata field. *See* sqlc.sqlcvar.sqlcdata field.
- SQLDATE data-type constant 2-8
See also DATE data type.
- sqlcda.h header file 1-23, 1-26, 17-5
- sqlcda.sqlc field
 - after a DESCRIBE 15-15, 17-7, 17-10, 17-21
 - definition of 15-10, 15-11
 - saving 17-17, 17-30
 - setting 17-32, 17-39
- sqlcda.sqlcvar structure
 - after a DESCRIBE 17-7
 - definition of 15-11
 - getting field values 17-13
 - setting fields 17-13
 - sqlcdata field. *See* sqlcda.sqlcvar.sqlcdata field.
 - sqlcflags field 15-13
 - sqlcformat field 15-12
 - sqlcdata field 15-12, 17-33
 - sqlclen field 15-12, 17-33

- sqlcind field 15-12, 15-30, 17-8, 17-10, 17-30, 17-33
- sqlctype field 15-12, 17-33
- sqlclen field. *See* sqlcda.sqlcvar.sqlclen field.
- sqlcname field 15-12, 17-8, 17-10, 17-21, 17-24
- sqlcownerlen field 15-13
- sqlcownername field 15-13
- sqlsourceid field 15-13, 15-31
- sqlsourcetype field 15-13, 15-31
- sqlctype field. *See* sqlcda.sqlcvar.sqlctype field.
- sqlcypelen field 15-13
- sqlcypename field 15-13
- sqlxid field 15-12
- sqlcda.sqlcvar.sqlcdata field
 - after a DESCRIBE 17-8, 17-10
 - after a FETCH 17-14, 17-18, 17-24, 17-29, 17-38
 - allocating memory for 15-11, 17-10, 17-23
 - definition of 15-12
 - freeing memory for 17-16, 17-25
 - setting column value 17-30, 17-31
 - setting input parameter data 17-33, 17-40
- sqlcda.sqlcvar.sqlclen field
 - after a DESCRIBE 17-8, 17-10
 - definition of 15-12
 - determining host variable type 17-15, 17-24
 - for varchar data 3-7
 - inserting opaque-type data 15-28
 - setting input parameter length 17-33, 17-40
 - used to allocate memory 17-12, 17-17, 17-23, 17-27
- sqlcda.sqlcvar.sqlctype field
 - after a DESCRIBE 17-8, 17-10
 - column type values 15-19, 15-31, 17-13
 - definition of 15-12
 - determining host variable type 17-15, 17-24
 - indicating distinct-type data 15-31
 - inserting opaque-type data 15-28

setting input parameter type 17-33, 17-40
 used to allocate memory 17-12, 17-17, 17-23, 17-27
SQLDBOOLEAN distinct-bit constant 15-31, 15-33
SQLDECIMAL data-type constant 2-8
See also DECIMAL data type.
sqldetach() library function 12-32, 12-40, 12-56, 13-25
SQLDISTINCT distinct-bit constant 15-31, 15-33
SQLDLVARCHAR distinct-bit constant 15-31, 15-33
sqldone() library function 12-32, 12-63, 12-69
SQLDTIME data-type constant 2-8
See also DATETIME data type.
sqlerrd array. *See* sqlca.sqlerrd array.
sqllexit() library function 12-40, 12-64
SQLFLOAT data-type constant 2-8
See also FLOAT data type.
sqlhdr.h file
 and FetBufSize global variable 14-23
 FetBufSize variable 14-23
sqlhdr.h header file 1-23, 1-26
 determining product version 13-15
 var binary macros 10-32
sqlhdr.h, and OptMsg global variable 12-43
sqlhosts file 12-6, B-2
sqlapi.h header file, definition of 1-23
SQLINT data-type constant 2-8
See also INTEGER data type.
SQLINT8 data-type constant 2-9
See also INT8 data type.
SQLINTERVAL data-type constant 2-8
See also INTERVAL data type.
SQLKEYWORD_ prefix 1-72
sqlen field. *See* sqlda.sqlvar.sqllen field.
SQLLIST data-type constant 2-9
See also LIST data type.
SQLLVARCHAR data-type constant 2-9
See also LVARCHAR data type.
SQLMONEY data-type constant 2-8
See also MONEY data type.
SQLMULTISET data-type constant 2-9
See also MULTISET data type.
SQLNCHAR data-type constant 2-8
See also NCHAR data type.
SQLNOTFOUND constant
 definition of 11-36
 detecting NOT FOUND condition 11-45
See also NOT FOUND condition.
SQLNULL constant. *See* Null value.
SQLNVCHAR data-type constant 2-8
See also NVARCHAR data type.
sqlproto.h header file 1-23, 1-25
SQLROW data-type constant 2-9
See also Row type (SQL).
SQLSERIAL data-type constant 2-8
See also SERIAL data type.
SQLSERIAL8 data-type constant 2-9
See also SERIAL8 data type.
SQLSET data-type constant 2-9
See also SET data type.
sqlsignal() library function 12-65
SQLSMFLOAT data-type constant 2-8
See also SMALLFLOAT data type.
SQLSMINT data-type constant 2-8
See also SMALLINT data type.
sqlstart() library function 12-19, 12-64, 12-67
SQLSTATE value 1-115, 12-95, 12-98, 12-100
SQLSTATE variable
 after a GET DIAGNOSTICS statement 11-13
 class and subclass codes 11-10, 11-14, 11-15
 determining database server features 11-22, 12-27
 determining number of exceptions 11-8
 determining origin of class portion 11-10, 11-22, 11-24, 11-25
 determining origin of subclass portion 11-10, 11-22, 11-24, 11-25
 in diagnostics area 11-10
 in thread-safe code 13-17, 13-27
 indicating truncation 3-11, 3-17
 result codes 11-19
 using 11-12
 using the WHENEVER statement 11-45
 warning values 11-22
See also SQLCODE variable.
sqltype.h header file
 definition of 1-23
 statement-type constants 11-33, 15-16
SQLTEXT data-type constant 2-8
See also TEXT data type.
sqltype field. *See* sqlda.sqlvar.sqltype field.
sqltypes.h header file
 data-type constants 2-7, 15-19, 16-9
 definition of 1-23
 distinct-bit constants 15-31
 distinct-bit macros 15-31
 simple-large-object data types 8-6, 8-9
 source type for distinct columns 15-9, 15-13
SQLUDTFIXED data-type constant 2-9, 15-33
See also Fixed-length opaque data type.
SQLUDTVAR data-type constant 2-9, 15-33
See also Varying-length opaque data type.
sqlvar structure. *See* sqlda.sqlvar structure.
sqlvar_struct structure
 with fetch array 15-37
sqlvar_struct typedef. *See* sqlda.sqlvar structure.

SQLVCHAR data-type constant 2-8

See also VARCHAR data type.

sqlxtype.h header file

definition of 1-23

X/Open data types 1-65, 2-9,
15-20

-ss processor option 1-85, 1-92, 1-93

Standards. *See* ANSI SQL

standards; X/Open standards.

START DATABASE

statement 12-17, 12-19

Statement identifier

case sensitivity 1-18

creating 14-5

freeing 14-13

scope rules 1-64, 13-22

structure 14-5

using delimited identifiers 1-33

Static cursor

with OPTOFC feature 14-36

static preprocessor option 1-76,

13-6, 13-7, 13-28

-static preprocessor option 1-50

Static-link library 1-94, 1-95

stcat() library function 3-42, 14-7

stchar() library function 3-44

stcmp() library function 3-46

stcopy() library function 3-48, 14-7

stleng() library function 3-50

Storage characteristics

altering 7-36

column-level 7-14, 7-39

create-time flags 7-10

disk-storage information 7-8

inheritance hierarchy 7-13

obtaining 7-12, 7-31, 7-84

sbspace-level 7-13

system default 7-13

system-level 7-49

system-specified 7-13

user-defined 7-15

Stored procedure 14-47

See also EXECUTE PROCEDURE
statement.

stores7 database

creating and populating B-3

string data type

corresponding SQL data type 2-4,
2-12

defined constant 2-10

definition of 3-4, 3-5

fetching into 2-18, 3-11, 3-13, 3-17,
5-6, 6-35

inserting from 3-12, 3-14, 3-17,
3-18

with ANSI-compliant

database 3-18

with empty string 3-6

See also CHAR data type.

strncmp() system call 11-20, 11-24

struct data type. *See* Structure.

Structure

as host variable 1-37

decimal 4-4

dtime 6-29

for dynamic management 15-4

ifx_int8_t 5-7

ifx_lo_create_spec_t 7-8

ifx_lo_stat_t 7-30

ifx_lo_t 7-18

intrvl 6-30

locator. *See* Locator structure.

nesting 1-37

sqlca 11-27

sqlda 15-10

sqlvar_struct 15-11

Subclass value. *See* SQLSTATE

variable, class and subclass
codes.

-subsystem processor option 1-85,
1-93

-Sw processor option 1-85, 1-93

System call

calloc() 17-10

DCE restrictions 13-25

exec() 12-57, 13-25

exit() 11-33, 11-44

fclose() 1-62

fopen() 1-62

fork() 8-62, 12-56, 13-25

fread() 1-62

free() 8-14, 17-25

longjmp() 12-35, 12-63

malloc() 8-14, 17-10, 17-11, 17-25

open() 8-19

setjmp() 12-35

signal() 12-35

strncmp() 11-20, 11-24

vfork() 12-57

wait() 8-63

System-descriptor area

allocating memory for 16-5

assigning values to 16-8

COUNT field. *See* COUNT
descriptor field.

data type lengths 3-7

definition of 15-5, 16-3

examples 16-15, 16-24,
16-30, 16-37

fetching rows into 16-11

fields of 15-5, 15-7

for columns of a SELECT 16-13,
16-29

for columns of an INSERT 16-29

for distinct-type columns 15-31

for input parameters 16-36, 16-43,
16-46, 17-41

for opaque-type columns 15-28,
15-30

for return values of a user-defined
function 16-22

freeing memory for 16-12

getting field values 16-10

initializing 16-6

interrupting database

server 12-51

item descriptor fields 15-7

managing 16-4

obtaining values from 16-8

setting fields 16-9

specifying input parameter values
for 16-10

uses for 16-13

using an indicator variable 15-8,
16-30, 16-36

T

Tabs 1-74

-target processor option 1-83, 1-93,
1-96, 12-15

TEXT data type

corresponding ESQ/C data
type 2-4, 2-12

declaring host variable for 8-5

defined constant 2-8

- inserting 8-17, 8-23, 8-28, 8-34, 16-33
- locator structure shown 8-7
- on optical disc 8-40
- role of locator.h 1-23, 2-12
- selecting 8-15, 8-22, 8-27, 8-33, 16-34
- subscripting 8-6
- See also* loc_t data type; Simple large-object data type.
- thread preprocessor option 1-50, 1-57, 1-59, 1-76, 13-6, 13-17, 13-27
- for dynamic thread library 13-51
- THREADLIB environment variable 13-26, 13-27
- Thread-safe application
 - concurrent connections 13-18
 - connections across threads 13-20
 - creating 13-16
 - cursors across threads 13-24
 - DCE restrictions 13-25
 - decimal values 13-25, 13-30
 - DISCONNECT ALL statement 13-22
 - environment variables 13-25
 - linking 13-26
 - preparing statements 13-22
 - programming hints 13-17
 - resource allocation 13-26
 - sample 13-33
 - SET CONNECTION statement 13-18, 13-20
 - thread-safe code 13-17
- thread-safe DLLs for Windows 1-59
- Time-out interval 12-36, 12-53, 12-69
- timeout sample program 12-69
- Trailing blanks
 - in VARCHAR conversion 3-13, 3-15
 - inserting into database 3-14
 - removing 3-30
 - with ESQL/C data types 3-5
- Transaction
 - committing 12-29
 - determining if used 11-15, 11-22, 11-30, 12-26

- exiting all connections 12-64
- interrupting the database server 12-51
- rolling back 12-29
- switching server connections 12-29
- Truncated value
 - in CHAR conversion 3-11, 3-17
 - in decimal conversion 4-49
 - in VARCHAR conversion 3-13, 3-15
 - indicated by sqlca 11-31
 - indicated by SQLSTATE 11-15
 - of opaque data type 15-30
 - using indicator variable 1-42, 1-44
 - with pointers 1-39
- ts preprocessor option 1-57
- TU_DAY qualifier macro 6-31
- TU_DTENCODE qualifier macro 6-32, 6-42
- TU_ENCODE qualifier macro 6-32
- TU_END qualifier macro 6-32
- TU_FLEN qualifier macro 6-32
- TU_Fn qualifier macro 6-32
- TU_FRAC qualifier macro 6-32
- TU_HOUR qualifier macro 6-31
- TU_IENCODE qualifier macro 6-32
- TU_LEN qualifier macro 6-32
- TU_MINUTE qualifier macro 6-31
- TU_MONTH qualifier macro 6-31
- TU_SECOND qualifier macro 6-32
- TU_START qualifier macro 6-32
- TU_YEAR qualifier macro 6-31
- TYPE descriptor field
 - after a DESCRIBE 16-6, 16-21
 - column type values 15-19, 16-9
 - definition of 15-7
 - indicating distinct-type data 15-31
 - inserting opaque-type data 15-28
 - setting column type 15-31, 16-33
 - setting input parameter type 16-36, 16-41
- Typed collection variable
 - allocating memory for 9-12
 - deallocating memory for 9-12
 - declaring 9-6
 - operating on 9-13

- Typed row variable
 - allocating memory for 9-57
 - deallocating memory for 9-57
 - declaring 9-50
 - operating on 9-58
- typedef
 - as host variable 1-38
 - dec_t 4-4
 - dtime_t 6-29, 6-33
 - intrvl_t 6-30, 6-34

U

- UDR. *See* User-defined routine.
- undef preprocessor directive 1-12, 1-14, 1-60
- Union of structures 1-38, 8-8
- Unnamed row type
 - after a DESCRIBE 17-8
 - inserting into 9-71
 - literal values 9-75
 - updating 9-71
 - See also* Named row type; Row type (SQL).
- Untyped collection variable
 - allocating memory for 9-12
 - deallocating memory for 9-12
 - declaring 9-9
 - operating on 9-13
- Untyped row variable
 - allocating memory for 9-57
 - deallocating memory for 9-57
 - declaring 9-52
 - operating on 9-58
- Update cursor 9-27, 14-15
- UPDATE statement
 - and NOT FOUND condition 11-21, 11-36
 - collection columns 9-40, 9-41
 - collection data 9-29
 - Collection Derived Table clause 9-29, 9-66
 - defined statement constant 15-16, 15-17
 - determining estimated cost of 11-30
 - determining number of rows updated 11-8, 11-30

- determining rowid 11-30
- dynamic 14-40, 14-52, 15-25
- failing to access rows 11-29
- in ANSI-compliant database 3-18
- interrupting 12-33, 12-51
- known at compile time 14-40, 14-52
- not known at compile time 15-25
- parameterized 14-52, 15-26, 16-46, 17-43
- row-type columns 9-71, 9-74
- row-type data 9-66
- SET clause 1-37, 14-52
- updating smart large-object data 7-17
- WHERE CURRENT OF clause 14-19
- with DESCRIBE 15-15, 15-22
- without WHERE clause 11-15, 11-23, 11-31, 15-17, 15-22
- upgcd_me sample program 8-17
- upgcd_nf sample program 8-28
- upgcd_of sample program 8-23
- USER network parameter 1-106, 12-13
- User-defined data type. *See* Distinct data type; Opaque data type.
- User-defined function
 - arguments 14-45
 - compared with procedure 14-46
 - creating 14-46
 - cursor 14-48, 14-50, 16-4, 17-4
 - definition 14-46
 - determining return values dynamically 15-26
 - dropping 14-46
 - executing 14-46, 14-48
 - executing dynamically 14-46, 14-48
 - known at compile time 14-46
 - noncursor 14-45, 14-48, 14-49, 16-4, 16-12, 17-4
 - not known at compile time 15-26
 - parameterized 14-45, 14-52, 14-54, 16-44, 17-42
 - with sqlda structure 17-26
 - with system-descriptor area 16-22

See also Cursor function; External function; Noncursor function; SPL function; User-defined routine.

- User-defined procedure
 - arguments 14-45
 - compared with function 14-46
 - creating 14-46
 - definition 14-46
 - dropping 14-46
 - executing 14-46, 14-47
 - executing dynamically 14-47
 - parameterized 14-45, 14-52, 16-45, 17-42
 - See also* External procedure; SPL procedure; User-defined routine.
- User-defined routine
 - definition 14-46
 - error messages 11-18, 11-25
 - languages supported 14-47
 - warning messages 11-15, 11-22
 - See also* User-defined function; User-defined procedure.
- USING DESCRIPTOR clause
 - of EXECUTE statement 17-4, 17-14, 17-31, 17-41, 17-43
 - of FETCH statement 17-4, 17-14
 - of OPEN statement 17-4, 17-14
 - of PUT statement 17-4, 17-14, 17-31
 - with a fetch array 15-34
- USING host_var clause
 - of EXECUTE statement 14-52
 - of OPEN statement 14-54
- USING SQL DESCRIPTOR clause
 - of DESCRIBE statement 16-4, 16-6, 16-14, 16-29
 - of EXECUTE statement 16-4, 16-11, 16-30
 - of FETCH statement 16-4, 16-11, 16-14
 - of OPEN statement 16-4, 16-11
 - of PUT statement 16-5, 16-11, 16-35
- Utility
 - finderr 1-6
 - ILogin 1-6

- INFORMIX FIND ERROR 1-6
- regcopy 1-6
- Setnet32 1-6

V

- V preprocessor option 1-50, 1-58
- V processor option 1-54
- VALUE descriptor field 16-21
- value.h header file 1-24
- var binary data type
 - checking for null 10-31, 10-33, 10-41
 - corresponding SQL data type 2-5, 2-13
 - deallocating data buffer 10-35
 - declaration 10-24
 - defined constant 2-11
 - fetching into 10-30
 - getting data buffer from 10-39
 - getting size of data buffer 10-40
 - inserting from 10-28
 - setting data buffer 10-42
 - setting size of data buffer 10-44
 - setting to null 10-29, 10-46
 - specifying memory allocation 10-37
 - See also* Varying-length opaque data type.
- VARCHAR data type
 - corresponding ESQ / C data type 2-4, 2-12, 3-3
 - data conversion 2-19, 3-12
 - defined constant 2-8
 - fetching 2-19, 3-13
 - inserting 2-19, 3-14
 - macros 3-7
 - role of varchar.h 1-24
 - truncating values 3-13, 3-15
 - with null-terminated strings 3-4, 3-5
 - See also* varchar data type.
- varchar data type
 - corresponding SQL data type 2-4, 2-12
 - defined constant 2-10
 - definition of 3-4, 3-6
 - fetching into 2-18, 3-11, 3-17, 5-6

inserting from 3-12, 3-14, 3-17, 3-18
 role of varchar.h 1-24
 with ANSI-compliant database 3-18
See also VARCHAR data type; CHAR data type.
 varchar.h header file 1-24, 3-7
 Variable
 host. *See* Host variable.
 indicator. *See* Indicator variable.
 main 1-42
 Varying-length opaque data type
 declaring host variable for 10-24
 inserting 10-28
 selecting 10-30
See also Fixed-length opaque data type; var binary data type.
 VCLENGTH varchar macro 3-7
 VCMAX varchar macro 3-7
 VCMIN varchar macro 3-7
 VCSIZ varchar macro 3-7
 Version independence 1-89
 Version information 1-54, 1-57
 vfork() system call 12-57

W

wait() system call 8-63
 Warnings
 definition of 11-4
 displaying in Microsoft format 1-74
 extensions to X/Open standards 1-65
 in user-defined routines 11-15, 11-22
 Informix-specific messages 11-22, 11-37
 redirecting 1-64
 suppressing 1-65, 1-74
 using sqlca structure 11-37
 using SQLSTATE variable 11-22
 using the WHENEVER statement 11-45
 X/Open messages 11-22
See also Error handling; Exception handling.

-wd processor option 1-83, 1-93, 1-96, 12-15
 WDEMO sample program 1-96
 -we processor option 1-83, 1-93
 WHENEVER statement 11-45
 Wildcard character
 exclamation point (!) 7-42
 question mark (?) 7-42
 with smart large-object filenames 7-42
 Windows syntax option
 -cpu 1-91
 -cpu:i 1-91
 -cpu:i386 1-91
 -Pi 1-92
 -rt:d 1-92
 -rt:m 1-92
 -rt:s 1-92
 -rt:t 1-92
 -runtime 1-92
 -runtime:libc 1-92
 -runtime:libcmt 1-92
 -runtime:msvcrt 1-92
 -Sc 1-93
 -ss:c 1-92
 -ss:w 1-93
 -subsystem:console 1-93
 -Sw 1-93
 -target:dll 1-93
 -target:exe 1-93
 -wd 1-93
 -we 1-93
 WORM optical disc 7-32, 8-40

X

xa.h header file, definition of 1-24
 -xopen preprocessor option 1-57, 1-65, 15-19, 15-21
See also X/Open standards.
 XSQLCHAR data-type constant 2-9
See also CHAR data type.
 XSQLDECIMAL data-type constant 2-9
See also DECIMAL data type.
 XSQLFLOAT data-type constant 2-9
See also FLOAT data type.

XSQLINT data-type constant 2-9
See also INTEGER data type.
 XSQLSMINT data-type constant 2-9
See also SMALLINT data type.
 X/Open compliance level Intro-15
 X/Open standards
 checking for Informix extensions to 1-65
 connecting to a database 12-17
 data type defined constants 1-65, 2-9, 15-20
 getting diagnostic information 11-6
 nonstandard system descriptor fields 15-8
 runtime error values 11-24
 SQLSTATE class values 11-14
 TYPE field values 16-36
 using dynamic SQL statements 15-4, 15-23, 15-24, 15-25, 15-26
 warning values 11-22
 warnings on extensions 1-65
See also xopen preprocessor option.